UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Algorithms II

Academic year 2013–2014

Michaelmas term 2013

http://www.cl.cam.ac.uk/teaching/1314/AlgorithII/

frank.stajano--algs2@cl.cam.ac.uk

Revised 2013 edition

Revision 22 of 2013-10-09 17:04:49 +0100 (Wed, 09 Oct 2013).

© 2005–2013 Frank Stajano

Contents

1 Advanced data structures **7**
1.1 Priority queues . . . 7
1.1.1 Binary heaps . . . 9
1.1.2 Binomial heaps . . . 10
1.2 Amortized analysis . . . 13
1.2.1 Aggregate analysis . . . 14
1.2.2 The potential method . . . 15
1.3 Fibonacci heaps . . . 17
1.3.1 Historical motivation and design criteria . . . 17
1.3.2 Core ideas . . . 18
1.3.3 Implementation . . . 20
1.3.4 Amortized analysis . . . 23
1.3.5 Why Fibonacci? . . . 26
1.4 van Emde Boas trees . . . 27
1.4.1 Array . . . 28
1.4.2 Binary tree . . . 29
1.4.3 Two-level tree . . . 29
1.4.4 Proto-vEB tree . . . 30
1.4.5 vEB tree . . . 34
1.5 Disjoint sets . . . 36
1.5.1 List implementation . . . 38
1.5.2 Forest implementation . . . 39

2 Graph algorithms **41**
2.1 Basics . . . 41
2.1.1 Definitions . . . 42
2.1.2 Graph representations . . . 43
2.1.3 Searching (breadth-first and depth-first) . . . 44
2.2 Topological sort . . . 46
2.3 Minimum spanning tree . . . 48
2.3.1 Kruskal's algorithm . . . 50
2.3.2 Prim's algorithm . . . 51
2.4 Shortest paths from a single source . . . 53
2.4.1 The Bellman-Ford algorithm . . . 54
2.4.2 The Dijkstra algorithm . . . 56
2.5 Shortest paths between any two vertices . . . 58

2.5.1 All-pairs shortest paths via matrix multiplication 58
2.5.2 Johnson's algorithm for all-pairs shortest paths 61
2.6 Maximum flow 62
2.6.1 The Ford-Fulkerson maximum flow method 64
2.6.2 Bipartite graphs and matchings 65

3 Parallel algorithms 67
3.1 Programming model: dynamic multithreading 69
3.2 Performance analysis 70
3.2.1 Definitions 70
3.2.2 Scheduling 71
3.2.3 Race conditions 74
3.3 Case study: chess program 75

4 Geometric algorithms 76
4.1 Intersection of line segments 76
4.1.1 Cross product 77
4.1.2 Intersection algorithm 77
4.2 Convex hull of a set of points 78
4.2.1 Graham's scan 79
4.2.2 Jarvis's march 80

Preliminaries

Course content and textbooks

This course, a continuation of what we did in Algorithms I last year, is about some of the coolest stuff a programmer can do.

Most real-world programming is conceptually pretty simple. The undeniable difficulties come primarily from size: enormous systems with millions of lines of code and complex APIs that won't all comfortably fit in a single brain. But each piece usually does something pretty bland, such as moving data from one place to another and slightly massaging it along the way.

Here, it's different. In our two courses on algorithms and data structures, but especially in this second helping, we look at pretty advanced hacks—those ten-line chunks of code that make you want to take your hat off and bow.

The only way really to understand this material is to program and debug it yourself, and then run your programs step by step on your own examples, visualizing intermediate results along the way. You might think you are fluent in n programming languages but you aren't really a programmer until you've written and debugged some hairy pointer-based code such as that required to cut and splice the doubly-linked lists used in Fibonacci trees. (Once you do, you'll know why.)

However the course itself isn't about programming: like "Algorithms I", it's about designing and analysing algorithms and data structures—the ones that great programmers then write up as tight code and put in libraries for other programmers to reuse. It's about finding smart ways of solving difficult problems, and about measuring different solutions to see which one is actually smarter.

In order to gain a more than superficial understanding of the course material you will also need a full-length textbook, for which this handout is *not* a substitute. If you were diligent last year you probably already have it:

> [CLRS3] Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms, Third edition.* MIT press, 2009. ISBN 978-0-262-53305-8.

A heavyweight book at about 1300 pages, it covers a little more material and at slightly greater depth than most others. It includes careful mathematical treatment of the algorithms it discusses and is a natural candidate for a reference shelf. Despite its bulk and precision this book is written in a fairly friendly style. At some point it was also *the computer science book with the highest number of citations*! You can't properly call yourself a computer scientist if CLRS3 is not on your bookshelf. It's the default text for this course: by all means feel free to refer to other books too, but chapter references in the chapter headings of these notes are to this textbook.

Other algorithms textbooks worth consulting as optional additional references include at least Sedgewick; Kleinberg and Tardos; and of course the legendary Knuth. Full bibliographic details are in the syllabus and on the course web page. However none of these textbooks covers *all* the topics in the syllabus, so you're still better off getting yourself a copy of CLRS3 (which by the way, in spite of its quality, is also the cheapest of the bunch).

Note also that a growing number of usually accurate algorithm and data structure descriptions can be found on Wikipedia. Once you master the material in this course, and especially after you've earned some experience by writing and debugging your own implementation, consider improving any descriptions that you find lacking or unclear.

What is in these notes

Many of the algorithms discussed in this course manipulate graphs. Most of the advanced data structures discussed in this course are useful for efficiently implementing some graph-based algorithm. Whatever topic we choose to start from, there will necessarily be some forward references—either to the algorithms that justify the existence and features of the data structures, or to the data structures that allow the algorithms to achieve the claimed asymptotic performance. Please be patient about that: we have to break the circular references by starting somewhere; and you can always read ahead if you wish.

These notes are meant as a clear and concise reference but they are not a substitute for having your own copy of the recommended textbook. They can help you navigate through the textbook and can help when organizing revision. I recommend that you come to each lecture with this handout *and* a paper notebook, and use the latter to take notes and sketch diagrams. In your own time, use the same notebook to write down your (attempts at) solutions to exercises as you prepare for upcoming supervisions.

These notes contain short exercises, highlighted by boxes, that you would do well to solve as you go along to prove that you are not just reading on autopilot. They tend to be easy (most are meant to take not more than five minutes each) and are therefore insufficient to help you really *own* the material covered here. For that, program the algorithms yourself[1] and solve problems found in your textbook or assigned by your supervisor. There is a copious supply of past exam questions at `http://www.cl.cam.ac.uk/teaching/exams/pastpapers/` under both "Algorithms II" and "Data Structures and Algorithms" but, particularly for the latter, be sure to choose (or ask your supervisor to help you choose) ones that are covered by this year's syllabus, because the selection of topics offered in these courses has evolved throughout the years.

Acknowledgements and history

I wrote my first version of these course notes in 2005, for Data Structures and Algorithms, building on the final 7–8 pages of the excellent notes for that course originally written by Arthur Norman and then enhanced by Roger Needham (my academic grandfather—the PhD supervisor of my PhD supervisor) and Martin Richards. I hereby express my

[1]The more programs you write to recreate what I show you in the lectures, the more you will really own this material.

gratitude to my illustrious predecessors. In 2006 this part of the course split off from the rest and became Algorithms II. I did a major revision and expansion of the notes for the 2007 edition, as the course grew from 6 to 8 lectures, another one in 2008 as it grew to 10 lectures and another one in 2011 as it grew to 12. Naturally I also did minor revisions and corrections in all the other years in which I taught the course. This academic year (2013–2014) is the last time I'll be teaching this Algorithms II course. Following another rearrangement, your successors will study this material in the context of a new and extended first-year Algorithms module, of which I'll teach the first half and Thomas Sauerwald the second.

Although I don't know where they are, from experience I am pretty sure that these notes still contains bugs, as all non-trivial documents do. Consult the course web page for the errata corrige. I am grateful to Sam Staton, Christian Richardt, Long Nguyen, Michael Williamson, Myra VanInwegen, Manfredas Zabarauskas, Ben Thorner, Simon Iremonger, Heidi Howard, Tom Sparrow, Simon Blessenohl, Nick Chambers, Nicholas Ngorok, Miklós András Danka and particularly Alastair Beresford for sending me corrections to previous editions. If you find any more corrections and email them to me, I'll credit you in any future revisions.

Last but not least, many thanks to Tim Griffin for lecturing this course in my stead during my sabbatical in 2009–2010.

Chapter 1

Advanced data structures

Chapter contents

Amortized analysis. Priority queues. Binomial heaps. Fibonacci heaps. van Emde Boas trees. Disjoint sets.
Expected coverage: about 5 lectures.

This chapter discusses some advanced data structures that will be used to improve the performance of some of the graph algorithms to be studied in chapter 2. We also introduce amortized analysis as a more elaborate way of computing the asymptotic complexity in cases where worst-case analysis of each operation taken individually would give a needlessly pessimistic and insufficiently tight bound.

Note that these advanced data structures do not generally offer support for finding an item given its key; it is tacitly assumed that the calling client program will maintain pointers to the nodes of the data structure it uses so that, when the client invokes a method on a specific node, the data structure already knows where that node is, as opposed to having to look for it. Therefore, when a method accepts or returns a parameter such as `Item x` in the Abstract Data Type description, what we implicitly mean is that, at the implementation level, `x` is a pointer to the node holding that item *within the data structure.*

1.1 Priority queues

Textbook

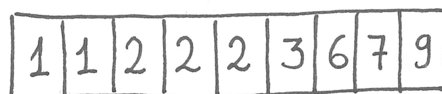
Study section 6.5 in CLRS3.

This section on priority queues should be a review of material already seen in Algorithms I. We keep it here to make this chapter self-contained.

A priority queue is a data structure that holds a dynamic set of items and offers convenient access to the item with highest priority[1], as well as facilities for extracting that item, inserting new items and promoting a given item to a priority higher than its current one.

```
ADT PriorityQueue {
  void insert(Item x);
  // BEHAVIOUR: add item <x> to the queue.

  Item first();
  // BEHAVIOUR: return the item with the smallest key (without
  // removing it from the queue).

  Item extractMin();
  // BEHAVIOUR: return the item with the smallest key and remove it
  // from the queue.

  void decreaseKey(Item x, Key new);
  // PRECONDITION: item <x> is already in the queue.
  // PRECONDITION: new < x.k
  // BEHAVIOUR: change the key of <x> to <new>, thereby increasing the priority
  // of x.
  // POSTCONDITION: x.k == new

  void delete(Item x);
  // PRECONDITION: item <x> is already in the queue.
  // BEHAVIOUR: remove item <x> from the queue.
  // IMPLEMENTATION: make <x> the new minimum by calling decreaseKey with
  // a value (conceptually: minus infinity) smaller than any in the queue;
  // then extract the minimum and discard it.
}

ADT Item {
  // A total order is defined on the keys.
  Key k;
  Data payload;
}
```

As for implementation, you could simply use a sorted array, but you'd have to keep the array sorted at every operation, for example with one pass of bubble-sort, which gives linear time costs for any operations that change the priority queue.



[1]"Highest priority" by convention means "earliest in the sorting order" and therefore "numerically smallest" in case of integers. Priority 1 is higher than priority 3.

Exercise 1
Why do we claim that keeping the sorted-array priority queue sorted using bubble sort has linear costs? Wasn't bubble sort quadratic?

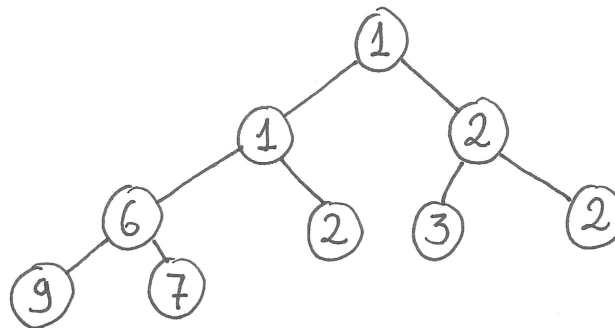
Operation	Cost with sorted array
creation of empty queue	$O(1)$
`first()`	$O(1)$
`insert()`	$O(n)$
`extractMin()`	$O(n)$
`decreaseKey()`	$O(n)$
`delete()`	$O(n)$

But we can do better than this.

1.1.1 Binary heaps

The binary heap is the priority queue data structure implicitly used in the heapsort algorithm[2]. It is a clever yet comparatively simple construction that allows you to read out the highest priority item in constant time cost (without removing it from the queue) and lets you achieve $O(\lg n)$ costs for all other priority queue operations.

A min-heap is a binary tree that satisfies two additional invariants: it is "almost full" (i.e. all its levels except perhaps the lowest have the maximum number of nodes, and the lowest level is filled left-to-right) and it obeys the "heap property" whereby each node has a key less than or equal to those of its children.



As a consequence of the heap property, the root of the tree is the smallest element. Therefore, to read out the highest priority item, just look at the root (constant cost). To insert an item, add it at the end of the heap and let it bubble up (following parent pointers) to a position where it no longer violates the heap property (max number of steps: proportional to the height of the tree). To extract the root, read it out, then replace it with the element at the end of the heap, letting the latter sink down until it no longer violates the heap property (again the max number of steps is proportional to the height of the tree). To reposition an item after decreasing its key, let it bubble up towards the root (again in no more steps than the height of the tree, within a constant factor).

[2] Except that heapsort uses a max-heap and here we use a min-heap.

Since the tree is balanced (by construction, because it is always "almost full"), its height never exceeds $O(\lg n)$, which is therefore the asymptotic complexity bound on all the priority queue operations that alter the tree.

Operation	Cost with binary min-heap
creation of empty heap	$O(1)$
`first()`	$O(1)$
`insert()`	$O(\lg n)$
`extractMin()`	$O(\lg n)$
`decreaseKey()`	$O(\lg n)$
`delete()`	$O(\lg n)$

These logarithmic costs represent good value and the binary heap, which is simple to code and compact to store[3], is therefore a good choice, in many cases, for implementing a priority queue.

1.1.2 Binomial heaps

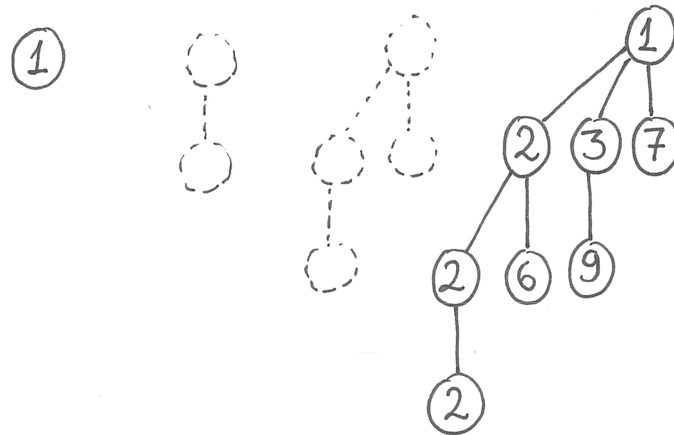
A more complex implementation of the priority queue is the binomial heap, whose main additional advantage is that it allows you to merge two priority queues, still at a time cost not exceeding $O(\lg n)$.

Exercise 2
As a comparison, what is the most efficient algorithm you can think of to merge two binary heaps? What is its complexity?

```
0 ADT BinomialHeap extends PriorityQueue {
1   void merge(BinomialHeap h);
2   // BEHAVIOUR: combine the current heap with the supplied heap h. In
3   // the process, make the supplied heap h empty and incorporate all its
4   // elements into the current heap.
5 }
```

A binomial heap is a forest of binomial trees, with special properties detailed below.

[3] The array representation does not even require any extra space for pointers.



A **binomial tree** (not heap) **of order 0** is a single node, containing one `Item`.

A **binomial tree of order** k is a tree obtained by combining two binomial trees of order $k-1$, by appending one of the trees to the root of the other as the (new) leftmost child[4]. By induction, it contains 2^k nodes (since the number of nodes doubles at each new order). By induction, the number of child subtrees of the root of the tree (known as the *degree* of the tree) is k, the same as the tree's order (since at each new order the tree gains one more child). By induction, the height of the tree is also k, again same as the tree's order (since at each new order the tree grows taller by one level, because the new child is as tall as the previous order's tree and is shifted down by one).

A **binomial heap** is a collection of binomial trees (at most one for each tree order), sorted by increasing size, each obeying the "heap property" by which each node has a key less than or equal to those of its children. If the heap contains n nodes, it contains $O(\lg n)$ binomial trees and the largest of those trees[5] has degree $O(\lg n)$.

Exercise 3
Draw a binomial tree of order 4.

Exercise 4
Give proofs of each of the stated properties of binomial trees (trivial) and heaps (harder until you read the next paragraph—try before doing so).

The following property is neat: since the number of nodes and even the *shape* of the binomial tree of order k is completely determined a priori, and since each binomial heap

[4]Note that a binomial tree is not a binary tree: each node can have an arbitrary number of children. Indeed, by "unrolling" the recursive definition above, you can derive an equivalent one that says that a tree of order k consists of a root node with k children that are, respectively, binomial trees of all the orders from $k-1$ down to 0.

[5]And hence *a fortiori* also each of the trees in the heap.

has at most one binomial tree for any given order, then, given the number n of nodes of a binomial heap, one can immediately deduce the orders of the binomial trees contained in the heap just by looking at the "1" digits in the binary representation of n. For example, if a binomial heap has 13 nodes (binary $1101 = 2^3 + 2^2 + 2^0$), then the heap must contain a binomial tree of order 3, one of order 2 and one of order 0—just so as to be able to hold precisely 13 nodes.

The operations that the binomial heap data structure provides are implemented as follows.

`first()` To find the element with the smallest key in the whole binomial heap, scan the roots of all the binomial trees in the heap, at cost $O(\lg n)$ since there are that many trees.

`extractMin()` To extract the element with the smallest key, which is necessarily a root, first find it, as above, at cost $O(\lg n)$; then cut it out from its tree. Its children now form a forest of binomial trees of smaller orders, already sorted by decreasing size. Reverse this list of trees[6] and you have another binomial heap. Merge this heap with what remains of the original one. Since the merge operation itself (*q.v.*) costs $O(\lg n)$, this is also the total cost of extracting the minimum.

`merge()` To merge two binomial heaps, examine their trees by increasing tree order and combine them following a procedure similar to the one used during binary addition with carry with a chain of full adders.

> "BINARY ADDITION" PROCEDURE. Start from order 0 and go up. At each position, say that for tree order j, consider up to three inputs: the tree of order j of the first heap, if any; the tree of order j of the second heap, if any; and the "carry" from order $j-1$, if any. Produce two outputs: one tree of order j (or none) as the result for order j, and one tree of order $j+1$ (or none) as the carry from order j to order $j+1$. All these inputs and outputs are either empty or they are binomial trees. If all inputs are empty, all outputs are too. If exactly one of the three inputs is non-empty, that tree becomes the result for order j, and the carry is empty. If exactly two inputs are non-empty, combine them to form a tree of order $j+1$ by appending the tree with the larger root to the other; this becomes the carry, and the result for order j is empty. If three inputs are non-empty, two of them are combined as above to become the carry towards order $j+1$ and the third becomes the result for order j.

The number of trees in each of the two binomial heaps to be merged is bounded by $O(\lg n)$ (where by n we indicate the total number of nodes in both heaps together) and the number of elementary operations to be performed for each tree order is bounded by a constant. Therefore, the total cost of the merge operation is $O(\lg n)$.

`insert()` To insert a new element, consider it as a binomial heap with only one tree with only one node and merge it as above, at cost $O(\lg n)$.

[6] An operation linear in the number of child trees of the root that was just cut off. Since the degree of a binomial tree of order k is k, and the number of nodes in the tree is 2^k, the number of child trees of the cut-off root is bounded by $O(\lg n)$.

`decreaseKey()` To decrease the key of an item, proceed as in the case of a normal binary heap within the binomial tree to which the item belongs, at cost no greater than $O(\lg n)$ which bounds the height of that tree.

Operation	Cost with binomial heap
creation of empty heap	$O(1)$
`first()`	$O(\lg n)$
`insert()`	$O(\lg n)$
`extractMin()`	$O(\lg n)$
`decreaseKey()`	$O(\lg n)$
`delete()`	$O(\lg n)$
`merge()`	$O(\lg n)$

Although the programming complexity is greater than for the binary heap, these logarithmic costs represent good value and therefore implementing a priority queue with a binomial heap is a good choice for applications where an efficient merge operation is required. If however there is no need for efficient merging, then the binary heap is less complex and somewhat faster.

Having said that, when the cost of an algorithm is *dominated* by specific priority queue operations, and where very large data sets are involved, as will be the case for some of the graph algorithms of chapter 2 when applied to a country's road network, or to the Web, then the search for even more efficient implementations is justified. We shall describe an even more efficient priority queue implementation in section 1.3 but, before that, we shall introduce a method that will let us analyse its overall performance with greater accuracy.

1.2 Amortized analysis

Textbook

Study chapter 17 in CLRS3.

Some advanced data structures support operations that are ordinarily fast but occasionally very slow, depending on the current state of the data structure. If we performed our usual worst-case complexity analysis, such operations would have to be rated according to their very slow worst case every time they are invoked. However, since we know that they *usually* run much faster than that, we might obtain a correct but needlessly pessimistic bound. Amortized analysis takes the viewpoint that the cost of these occasional expensive operations might in some cases be spread, for accounting purposes, among the more frequent cheap ones, yielding a tighter and more useful bound.

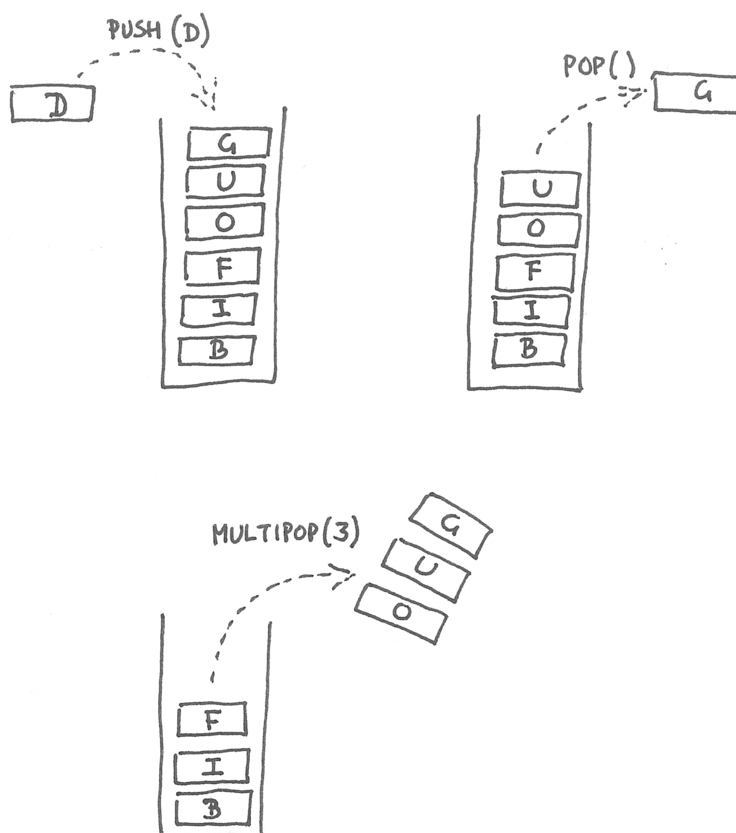
It is important to understand that amortized analysis *still* gives a worst-case bound (it is *not* a probabilistic estimate of the average case), but one that only applies to a sequence of operations over the lifetime of the data structure. Individual operations may exceed the stated amortized cost but the amortized bound will be correct for the whole sequence.

We introduce two ways of performing amortized analysis: aggregate analysis and the potential method.

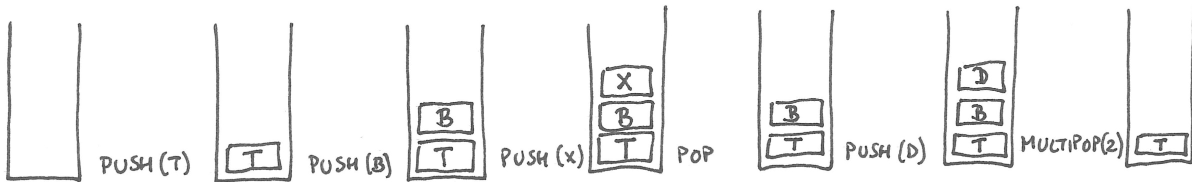
1.2.1 Aggregate analysis

In aggregate analysis the general idea is to take a sequence of operations, compute its total cost and then divide it by the number of operations in the sequence to obtain the amortized cost of each. All operations are rated at the same cost even though in reality some of them were cheaper and some were more expensive. An example will hopefully clarify.

Let's assume you have a stack data structure whose methods are the usual `void push(Item x)` and `Item pop()`, plus a special `void multipop(int n)` that removes and discards the top n elements from the stack (or however many are available if the stack contains fewer than n). What is the cost of a sequence of n operations, each of which could be `push`, `pop` or `multipop`, on an initially empty stack? The "traditional" worst-case analysis would tell us that, while `push` and `pop` have constant cost, a single `multipop` could cost up to n (since there could be up to n elements in the stack) and therefore the worst-case costs are $O(n^2)$.



This asymptotic bound is correct but it is not tight. It is easy to see that, since each item must be pushed before it is popped (whether on its own or as part of a `multipop`), the combined number of all elementary popping operations (whether from `pop` or in bursts from `multipop`) cannot exceed the number of pushing operations, in turn bounded by n which is the total number of operations in the sequence. Therefore aggregate analysis tells us that a sequence of n operations can cost at most $O(n)$ and that therefore the *amortized* cost of each of the n operations is $O(1)$.



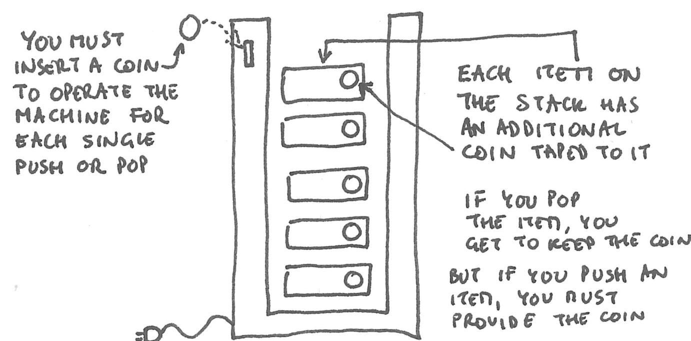
The example should make it clear that, if the amortized cost of an operation is low (e.g. the amortized cost of `multipop` is $O(1)$), this does *not* mean that all individual calls of that operation will be bounded by that low constant cost[7]. It does however mean that, in a sequence of n operations, the *overall* cost cannot build up to more than n times the (low) amortized cost of the individual operations.

1.2.2 The potential method

While aggregate analysis always attributes the same share of the cost to each of the operations in the sequence, the potential method can be more specific: it can give different amortized costs for different operations in the sequence.

The potential method works on the intuitive idea that, in order to compensate for the occasional expensive runs of some operations, if we are going to prove that their amortized cost is in fact lower than their worst-case cost when taken individually, we might "save" some "currency" somewhere, for the purpose of "repaying" the excessive costs when needed and still remain within the budget of the (lower) amortized cost when the overall sequence of operations is taken into account.

The "currency" is a totally fictitious entity that is not actually paid to anyone; it is just a way of accounting for the fact that, if an operation is frugal, that is to say it actually costs less than advertised, the "savings" can be used to "cover up" for a later operation that costs more than advertised. If we do that with savings that accumulate over many frugal operations, we may be able to cover up so well for occasional expensive operations that we lower their asymptotic complexity.



Let's revisit the stack example. Let's say the cost of one elementary `push` or `pop` is one currency unit[8]. Imagine this currency unit as a gold coin, if you wish. To push a plate on the stack in the cafeteria, you must pay a gold coin (insert coin into machine, whatever). Same for popping a plate from the stack. For a `multipop` of 4 plates, you'd

[7] Indeed it is still true that the worst-case non-amortized cost of a single `multipop` is $O(n)$, not $O(1)$.

[8] That's another way to say that these elementary operations have constant cost.

have to pay 4 coins. Now, imagine that at each push you actually sacrifice *two* gold coins: one to operate the machine and the other as a saving—you tape the second gold coin to the plate before pushing it on the stack. If you do that, the cost of a simple push becomes two currency units instead of one. Interestingly, that's still $O(1)$, because the constant factor is absorbed by the big-O notation. But what happens to `pop`? You pay one coin for operating the machine and taking off the plate, but you receive one coin that had been taped to the plate when it was pushed. So you end up doing `pop` at zero cost! And what about `multipop`? Amazingly, there too, even if you pop four plates (and therefore must pay four coins to operate the machine four times), you get back that many coins from the plates themselves, so you end up paying nothing in that case as well. Therefore, with the potential method, the amortized cost of `push` is $O(1)$ while the amortized cost of `pop` and `multipop` is zero[9]. Note that it is still the case, as before, that a sequence of n operations will cost no more than $O(n)$.

The point of the potential method is to devise an amortized cost structure that, despite being somewhat arbitrary, can be proved always to be an upper bound for the actual costs incurred, *for any possible sequence* of operations. To ensure that, it is essential to be always in credit.

If we call $\hat{c}_i$ the amortized cost of operation i, c_i its real cost and Φ_i the potential stored in the data structure before operation i, then the amortized cost (two coins to push one plate) is the same as the real cost (one coin to operate the machine) plus the increase in the potential of the data structure (one more coin deposited in the stack, taped to the plate):

$$\hat{c}_i = c_i + \Delta\Phi_i = c_i + \Phi_{i+1} - \Phi_i.$$

For a sequence of operations for i from 0 to $n-1$, the total amortized cost is:

$$\sum_{i=0}^{n-1} \hat{c}_i = \sum_{i=0}^{n-1} (c_i + \Delta\Phi_i) = \sum_{i=0}^{n-1} c_i + \Phi_n - \Phi_0.$$

If you want to be able to claim that the amortized cost is an upper bound for the real cost, you must be able to say that

$$\sum_{i=0}^{n-1} \hat{c}_i \geq \sum_{i=0}^{n-1} c_i$$

which, when you subtract the previous equation from it, becomes

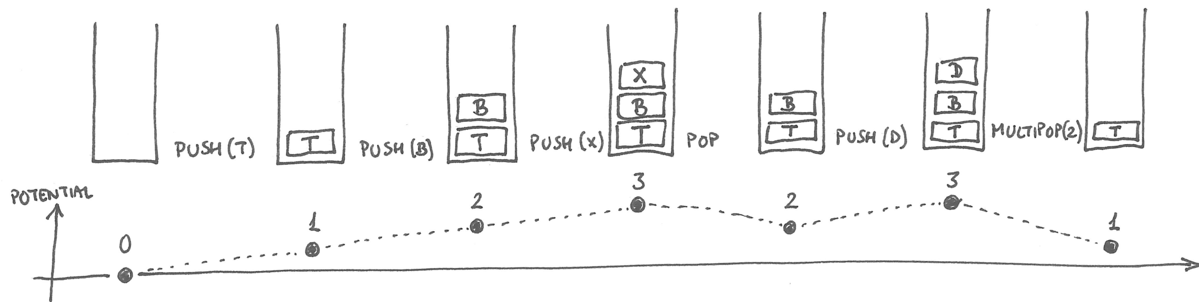
$$0 \geq -\Phi_n + \Phi_0$$

and therefore

$$\Phi_n \geq \Phi_0.$$

In other words, since the above must hold for any sequence and therefore for any n, the potential of the data structure must never at any time go below what it was initially. We may simplify the expression by arbitrarily setting $\Phi_0 = 0$ when the initially empty structure is created and saying that the potential must be nonnegative at all times.

[9] There is a subtlety here related to the cost of doing `pop` or `multipop` on an empty stack; if we wanted to account for the cost of these failed calls, we'd have to be more meticulous in the above, and we'd end up with $O(1)$ amortized costs for all three operations.



In the example, the potential stored in the stack data structure was represented by the coins on the plates—one currency unit per plate. So the "potential function" Φ of the data structure was essentially the number of plates on the stack, which can never be negative. Choosing a potential function that starts at zero and can never become negative will guarantee that your real costs will never exceed your amortized costs for *any* possible sequence of operations. This is what allows you to take the amortized costs as valid asymptotic bounds for the real costs.

1.3 Fibonacci heaps

Textbook

Study chapter 19 in CLRS3.

The Fibonacci heap is yet another implementation of the priority queue, also featuring a merge facility like the binomial heap. In exchange for additional coding complexity and occasional delays, it offers an amazing *constant amortized cost* for all priority queue operations that don't remove nodes from the queue, and logarithmic costs for those that do. It is a "lazy" data structure in which many operations are performed very quickly but in which other less frequently used operations require a major and potentially lengthy clean-up and rearrangement of the internals of the data structure.

Using the Fibonacci heap may be advantageous, compared to a binomial or a binary heap, in cases where we expect the number of cheap operations to be asymptotically dominant over that of expensive operations.

It must however be noted that the additional programming complexity required by Fibonacci heaps means that the big-O notation hides a fairly large constant; this, and the extra implementation effort required, means that the move to a Fibonacci heap is only justified when the priority queues to be processed are really large.

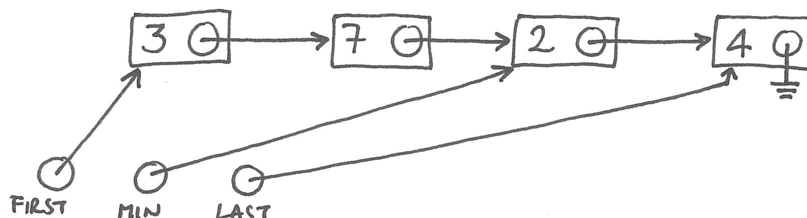
1.3.1 Historical motivation and design criteria

The Fibonacci heap was created out of a desire to improve the asymptotic running time of the Dijkstra algorithm (section 2.4.2) for finding shortest paths from a single source

on a weighted directed graph[10]. The limiting factor on the running time of the Dijkstra algorithm is the efficiency of the priority queue used to hold the vertices. The overall cost of Dijkstra is dominated by $|V|$ times the cost of `extractMin()` plus $|E|$ times the cost of `decreaseKey()`, where V and E are the sets of vertices and edges of the graph[11]. With a standard binary heap implementation for the priority queue, each of these individual costs is $O(\lg V)$; so the overall cost of Dijkstra is $O(V \lg V + E \lg V)$ which, on a connected graph, becomes $O(E \lg V)$ because $|E| \geq |V|$. This highlights that the critical operation is `decreaseKey()`: if we could reduce its cost without increasing that of the other operations, we would reduce the overall cost of Dijkstra. The development of the Fibonacci heap came out of this insight. It led to a data structure where, in amortized terms, `extractMin()` still costs $O(\lg V)$ but `decreaseKey()`, which is performed more frequently, has only constant cost. The overall amortized cost of Dijkstra then becomes $O(V \lg V + E)$ which is a definite improvement over $O(E \lg V)$. The same arguments bring an improvement from $O(E \lg V)$ to $O(V \lg V + E)$ to Prim's algorithm too (section 2.3.2).

1.3.2 Core ideas

One of the main ideas behind the Fibonacci heap is to use a lazy data structure in which many operations have constant cost because they do only what's necessary to compute the answer and then (unlike what happens with binomial heaps) return immediately without performing any tidying up. For example, in a binomial heap, both `insert()` and `merge()` require the elaborate binary-addition-like operation in which trees of the same degree are combined to form a tree of higher degree[12], possibly requiring further recombination with another existing tree and so on. Since each combining step takes constant cost, the `insert()` and `merge()` methods each take time bounded by the number of trees in the longest of the two binomial heaps (roughly speaking), which is $O(\lg n)$. We might instead imagine a lazy data structure (no tidying up after each insertion) in which the nodes are simply held in a linked list, with two extra pointers to indicate respectively the node with the smallest key (`min`) and the last node in the list. There, `insert()` and `merge()` would each only have constant cost: splicing in the new vertex or sublist at the front, and updating the `min` pointer after a single comparison. This gives $O(1)$ instead of $O(\lg n)$ for these two operations.



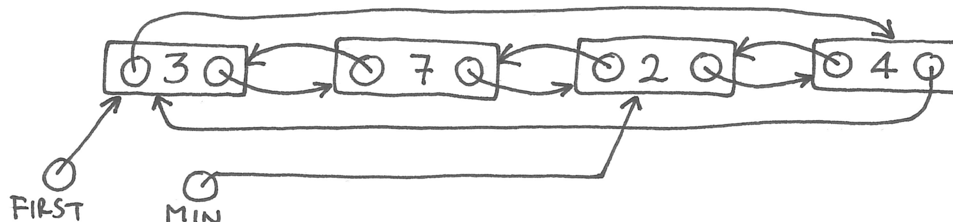
[10] This subsection contains forward references to some graph algorithms to be studied later and may be a little hard to understand fully until then. If so, please don't worry and come back to it later. Or follow the references and read ahead.

[11] See the remark on notation at the end of subsection 2.1.2 on page 43.

[12] In the original 1987 paper by Fredman and Tarjan that introduced Fibonacci trees, the number of children of a node is indicated as its *rank*, not its *degree*. We use *degree* in these notes to avoid confusion with a different definition of the term *rank* that is used in the context of disjoint sets (section 1.5), but be aware that in the literature you may find *rank* to mean "number of children".

In a binomial heap, returning the smallest value without removing it from the heap, as done by `first()`, requires a linear search among the roots of the trees, which costs $O(\lg n)$ since that's the bound on the number of trees. In our linked list data structure we just return the item pointed by `min`, at cost $O(1)$. The `decreaseKey()` method is equally cheap: you decrease the key of the designated node and, after one comparison, update the `min` pointer if necessary. Cost is $O(1)$ as opposed to the $O(\lg n)$ of the binomial heap where the key might have to navigate up the tree towards the root.

There is a catch, of course: in our lazy data structure as described so far, we end up paying $O(n)$ for `extractMin()`. The extraction itself is not the real problem: yes, it would require $O(n)$ to find, via sequential traversal, the predecessor of the node to be removed, but this can easily be brought back to $O(1)$ by using a doubly-linked list instead of a regular list, at no increase in asymptotic cost for any of the other previously described operations.



The real, unavoidable cost is finding the second-smallest item (the new minimum after the removal) so as to be able to update the `min` pointer; for the lazy data structure this requires scanning the whole list, which costs $O(n)$. This negates all the benefits of making the other operations $O(1)$ because for example Dijkstra would end up costing $O(V^2 + E)$. On sparse graphs where $|E| = O(V)$, the performance of Dijkstra using the list-based lazy structure is $O(V^2)$ which is much worse than the $O(V \lg V)$ achieved with a binomial or binary heap[13].

The trick, then, is to perform some moderate amount of tidying up along the way so that searching for the new minimum will cost less than $O(n)$. The strategy used by the Fibonacci heap is in some sense a halfway-house between the flat list and the binomial heap. As far as `insert()`, `merge()` and `first()` are concerned, we operate similarly to what happens with the flat list[14]. Whenever we perform `extractMin()`, though, after having extracted the minimum we rearrange all the remaining vertices into something very similar to a binomial heap; and only then we look for the minimum, now at reduced cost. This may sound suspicious at first, given the expected cost of building an entire binomial heap; but we are only claiming a logarithmic cost for `extractMin()` *in amortized terms*: the actual cost will be higher.

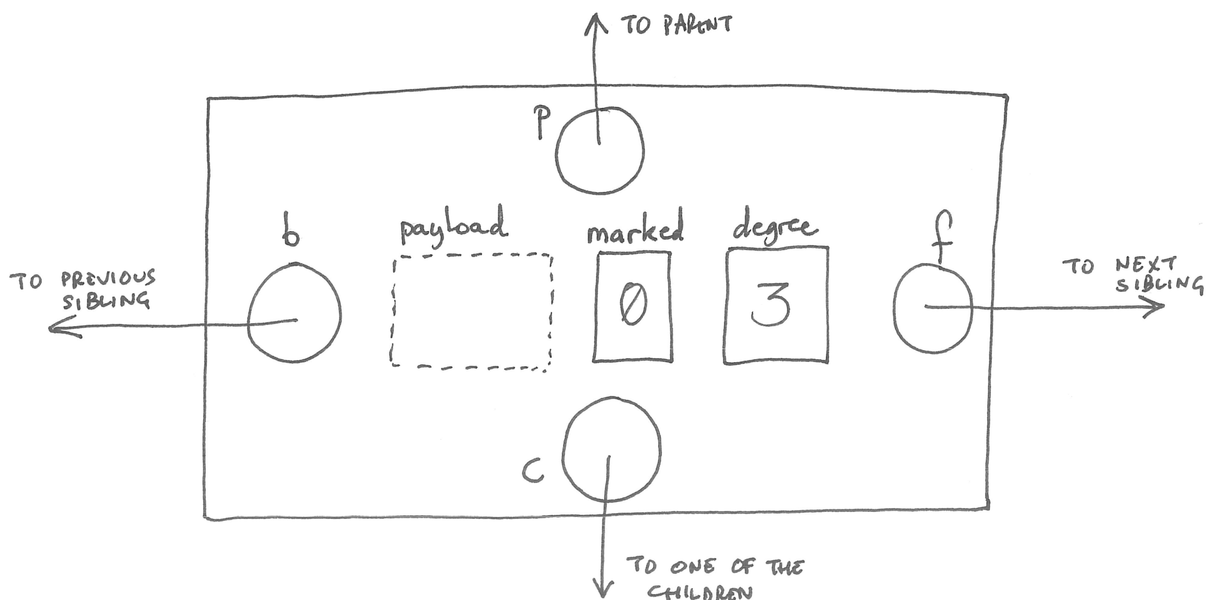
Having conveyed these general intuitions, there is no point in discussing things any further without first explaining precisely how Fibonacci heaps are implemented. So let's now do that. Once that is understood, we'll get back to the analysis and compute precise amortized costs for all operations.

[13]It is however true that $O(V^2 + E)$ is better than $O(E \lg V)$ on dense graphs where $|E| = O(V^2)$: there, it becomes $O(V^2)$ versus $O(V^2 \lg V)$.

[14]Indeed a Fibonacci heap is indistinguishable from a flat doubly-linked list with `min` pointer so long as only `insert()`, `merge()` and `first()` are performed on it.

1.3.3 Implementation

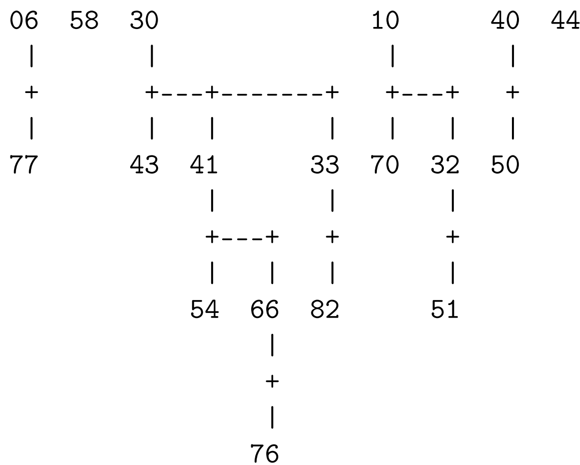
A Fibonacci heap is implemented as a forest of min-heap-ordered[15] trees. The heap also maintains a `min` pointer to the root with the smallest key among all the roots. The roots are linked together in a circular doubly-linked list, in no particular order. For each node, the children of the node are also linked together in a circular doubly-linked list, in no particular order. Each node has a pointer to its unique parent (`None` if the node is a root) and a pointer to one of its children (`None` if the node is a leaf; the other children are reachable via the circular list). Here is the layout of the `FibNode` data structure:



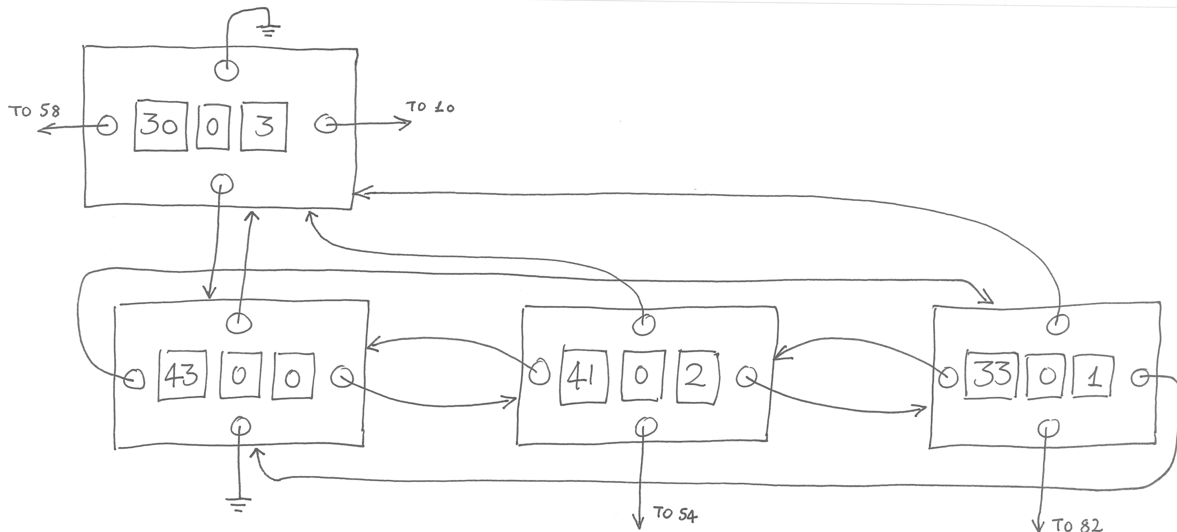

```
0 ADT FibNode extends Item {
1   FibNode b; // back pointer; previous item in sibling list
2   FibNode f; // forward pointer; next item in sibling list
3   FibNode p; // parent pointer; direct ancestor
4   FibNode c; // child pointer; other children reachable via sibling list
5   Boolean marked; // set iff this node is not a root and has lost a child
6   int degree; // number of children
7 }
```

Next is an ASCII-art example of a Fibonacci heap with 6 trees (not showing all the criss-crossing pointers, though). Each two-digit integer in the diagram stands for a full `FibNode`, of which it displays the payload. Horizontally-connected nodes, such as 43, 41 and 33, are siblings: they are actually members of a doubly-linked list.

[15] Each node with a parent has a key not smaller than that of the parent.



The following diagram magnifies a four-node portion of the previous Fibonacci heap, showing all the fields and pointers of each `FibNode`.



A **peculiar constraint** that the Fibonacci heap imposes on its trees, beyond that of being min-heap-ordered, is the following: after a node has been made a child of another node, it may lose at most one child before having to be made a root. In other words: while a root node (such as the one with key 30 in the above example) may lose arbitrarily many children without batting an eyelash, a non-root node (such as 41) may only lose at most *one* child node; after that, if it is to lose any other child nodes, it must be cut off from its tree and promoted to being a standalone root. The reason for this constraint will be explained during the asymptotic analysis: its ultimate purpose is to bound the time of `extractMin()` by ensuring that a node with n descendants (children, grandchildren etc) has a degree (number of children) bounded by $O(\lg n)$. The way the constraint is enforced is to have a boolean flag (the `marked` field) in each node, which is set as soon as the non-root node[16] loses its first child. When attempting to cut a child node from a non-root parent, if the parent is `marked` then the parent must be cut off as well and made a root (and unmarked), and so on recursively. This maintains the required invariant.

[16]Note that, by line 5 of the `FibNode` ADT definition on page 20, a root node is never marked. And if a marked node is ever made into a root, it must then lose its mark.

Exercise 5
Explain with an example why, if the "peculiar constraint" were not enforced, it would be possible for a node with n descendants to have more than $O(\lg n)$ children.

As is typical with priority queues, there is no efficient support for finding the node that holds a given key. Therefore the methods of a Fibonacci heap take already-created `FibNode`s as parameters and return `FibNode`s where appropriate. The calling code is responsible for allocating and deallocating such nodes and maintaining pointers to them.

`FibHeap constructor()`
Behaviour: Create an empty heap.
Implementation: Set `min` to `None` (constant cost).

`FibHeap constructor(FibNode n)`
Behaviour: Create a heap containing just node `n`.
Implementation: Make `min` point to this node and adjust `n`'s own pointers consistently with the fact that `n` is now the only node in the circular doubly-linked list of roots (constant cost)[17].

`void insert(FibNode n)`
Behaviour: Add node `n` to the heap.
Precondition: `n` is not in the heap.
Postcondition: The nodes now in this heap are those it had before plus `n`.
Implementation: create a heap with just `n` (constant cost) and merge it into the current heap (constant cost, *cfr* `merge()`).

`void merge(FibHeap h)`
Behaviour: Take another heap `h` and move all its nodes to this heap.
Postcondition: `h` is empty. The nodes now in this heap are those it had before plus those that used to be in `h`.
Implementation: Splice the circular list of roots of `h` into the circular list of roots of this heap (constant cost). Compare the two `min` and point to the winner (constant cost).

`FibNode first()`
Behaviour: Return (a pointer to) the node containing the minimum key, without removing that node from the heap.
Precondition: This heap is not empty.
Postcondition: This heap is exactly the same as it was before invoking the method.
Implementation: return `min` (constant cost).

`FibNode extractMin()`
Behaviour: Return (a pointer to) the node containing the minimum key, but also remove that node from the heap.

[17]Conceptually this is a private constructor, used only by the `insert()` method and not accessible to external clients.

Precondition: This heap is not empty.
Postcondition: This heap contains the same nodes it had previously (though not necessarily in the same layout), minus the one that contained the minimum key.
Implementation: Cut off the root pointed by `min` from the circular list of roots of this heap (constant cost). Cut off this root from its children (cost proportional to the number of children, because you must reset the `p` pointer of each child). Splice the circular list of children into the circular list of roots (constant cost). Repeatedly perform the "linking step" while it is possible to do so: if any two trees in the circular list of roots have the same degree k, i.e. the same number of child subtrees, combine them into a single tree of degree $k + 1$ by making the tree with the larger root key become a child of the other (cost to be discussed later). Set `min` to point to the root with the smallest key (cost proportional to the total number of trees now in the root list, also to be discussed later). Finally, at constant cost, return the (previous) minimum node that was cut off from the root list and from its children at the start of the operation. (Total cost to be discussed later.)

`void decreaseKey(FibNode n, int newKey)`
Behaviour: Decrease the key of node `n` to `newKey`.
Precondition: `n` is in this heap and `newKey` < `n.key`.
Postcondition: This heap contains the same nodes as it did previously (though not necessarily in the same layout), except that `n` now has a new key.
Implementation: Decrease `n.key` to the new value, then check if `n.key` is now smaller than the parent node's key. If it hasn't, the job is completed, at constant cost. If it has become smaller, the min-heap ordering has been violated. Restore it at constant cost by cutting off `n`, ensuring its `marked` flag is reset and making `n` into a new root, where it can have as low a key as it wants without disturbing the min-heap property. As you cut off the node, you must however obey the "peculiar constraint": if the node's parent is marked, it must also be cut off and unmarked (and its ancestors too, recursively, if necessary, causing a series of so-called *cascading cuts*). If the node's parent is not marked, and is not a root, it must be marked. (Cost of cascading cuts to be discussed later.)

`void delete(FibNode n)`
Behaviour: Remove the given node from the heap.
Precondition: `n` is in this heap.
Postcondition: This heap contains the same nodes as before this method was invoked, except for `n` which has been removed.
Implementation: As with any priority queue, you may implement `delete()` as a combination of `decreaseKey()` and `extractMin()`. Invoke `decreaseKey()` on node `n` with a `newKey` of $-\infty$, making that node the one with the smallest key in the heap. Then perform `extractMin()` to remove it. (Cost: the sum of the costs of `decreaseKey()` and `extractMin()`.)

1.3.4 Amortized analysis

Using the potential method we shall prove that `extractMin()` and `delete()` run in amortized $O(\lg n)$ cost and that all other operations, particularly `decreaseKey()`, run in

amortized constant cost.

We define the potential function Φ of the Fibonacci heap as $t + 2m$ where t is the number of trees and m is the number of marked nodes. This means, using the previous metaphor, that we must pay, or rather deposit in the data structure, an extra gold coin whenever we create a new tree at the root level and we must deposit two gold coins whenever we mark a node; however we do get back one gold coin whenever we remove a tree and we get back two coins whenever we unmark a node. The value of a gold coin is taken to be sufficient to cover the highest of the constant costs identified in the preliminary analysis in the previous section.

Let's now review each of the methods and assess its amortized cost.

`FibHeap constructor()`
: No trees and no nodes are created, so $\Phi = 0$. Amortized cost is equal to real cost, i.e. constant.

`FibHeap constructor(FibNode n)`
: We make a new (singleton) tree, so we deposit an extra gold coin in it on top of the real cost. The amortized cost is still constant; it's actually higher than the real cost, but the difference is hidden by the big-O notation.

`void merge(FibHeap h)`
: We rewrite a fixed number of pointers to splice together the two doubly-linked lists of roots. The roots of `h` already carry one gold coin each so there is no change in overall potential when we transfer them to this heap. The amortized cost is the same as the real cost, i.e. constant.

`void insert(FibNode n)`
: Creating the singleton heap forces us to pay a gold coin but that's still only a constant cost. We just saw that merging has amortized constant cost and therefore here too the overall amortized cost is constant (though slightly higher than the real cost).

`FibNode first()`
: The potential is not affected so the amortized cost is the same as the real cost: constant.

`FibNode extractMin()`
: Now things start to get interesting. Cutting off the node pointed by `min` from the root list yields a gold coin which repays us of any one-off work done. If c is the degree (= number of children) of that minimum node, for each of the c children we must perform a constant amount of work to reset the `p` (parent) pointer to `None`, plus we must deposit a gold coin into the child node because we are creating a new root-level tree. So we are paying the equivalent of $2c$ gold coins for this part. Then we must do the linking step: if any two trees have the same degree, we join them into a single tree of degree one higher by making one the child of the other. This action may have to be repeated many times, depending on how many trees there are (however many there were before, minus one, plus c) *but* we don't worry about it because each join (whose real cost is a constant since it's a fixed number

of pointer manipulations) pays for itself with the gold coin released by the tree that becomes a child of the other. So, in amortized terms, the whole linking step is free of charge. Next, finding the smallest root among the remaining trees after the linking step requires a linear search of all of the remaining trees. How many are there? heap with n nodes, call $d(n)$ the highest degree of any node in that heap; then, since the outcome of the linking step is that no two trees have the same degree, the number of trees is necessarily bounded by $d(n) + 1$ (the case in which we have the trees of all possible degrees from 0 to $d(n)$). Since $c \leq d(n)$ by definition of d, the overall amortized cost of `extractMin()`, including also the $2c$ mentioned previously, is $O(d(n))$. All that remains to do in order to support the original claim is therefore to prove that $d(n) = O(\lg n)$, which we shall do in section 1.3.5. The intuition is that we don't want any tree in a Fibonacci heap to grow "wide and shallow", otherwise the root will have too many children compared to the total number of nodes of the tree and the number of children will exceed $O(\lg n)$; and the "peculiar constraint" has precisely the effect of preventing "wide and shallow" trees. In summary, the amortized cost of `extractMin()` is $O(\lg n)$ even though the real cost may be much higher because of the hidden work involved in the linking step (which gets repaid by spending some of the gold coins that were stored in the data structure).

`void decreaseKey(FibNode n, int newKey)`

If the node doesn't need to move as a consequence of its key having been decreased, the potential is unchanged by the operation and therefore the amortized cost is the same as the real cost, i.e. constant. In the case where the node must be cut (because its key is now smaller than that of its parent, which would violate the min-heap property), let's assume that there are $p \geq 0$ marked ancestors above it that also need to be made into roots with cascading cuts. Each *marked* node that is cut releases two gold coins as it gets unmarked (remember roots are all unmarked). One coin covers for the cost of cutting and splicing, while the other coin is stored in the newly-created root. Therefore, any arbitrarily long chain of cascading cuts *of marked nodes* will exactly repay for itself in amortized terms. The only possibly non-marked node among those to be cut is `n` itself, the one whose key has been decreased. If `n` is unmarked (and it may or may not be) we must pay a constant cost for cutting and splicing it and then we must store a gold coin in the new tree root; furthermore, if the parent of `n` was unmarked and was not a root, we must mark it and store two coins in it, at a total worst-case cost of up to four coins (one stored in the new root, two in the newly-marked node, and one coin's worth to cover all the actual work of cutting, splicing and marking). This is still a constant with respect to n, therefore the overall amortized cost still won't exceed $O(1)$, even though the real cost may be much higher because of the hidden work involved in the cascading cuts.

`void delete(FibNode n)`

As previously noted, this is simply a two-step sequence of `decreaseKey()`, which costs $O(1)$ amortized, and `extractMin()`, which costs $O(\lg n)$ amortized; therefore in total this operation costs $O(\lg n)$ amortized.

The following table summarizes the amortized costs of all the methods of a Fibonacci

heap: all methods have constant amortized cost except those that remove nodes from the heap, which have logarithmic cost.

Operation	Cost (amortized) with Fibonacci heap
creation of empty heap	$O(1)$
`first()`	$O(1)$
`insert()`	$O(1)$
`extractMin()`	$O(\lg n)$
`decreaseKey()`	$O(1)$
`delete()`	$O(\lg n)$
`merge()`	$O(1)$

1.3.5 Why Fibonacci?

To complete the complexity analysis of `extractMin()` we must prove that the maximum degree of any node in an n-node Fibonacci heap is bounded by $O(\lg n)$; if we do so, then we have shown that the whole `extractMin()` operation is $O(\lg n)$. The full story will also explain the reason for the Fibonacci name assigned to the data structure.

We want to prove that, by building Fibonacci heaps in the way we described (including in particular the "linking step" during `extractMin()` and the "peculiar constraint" enforced by node marking in `decreaseKey()`), if a node has k children then the number of nodes in the subtree rooted at that node is at least exponential in k. In other words we shall prove that, if a subtree contains n nodes (root of subtree included), the root of that subtree has at most $O(\lg n)$ children.

Consider a node x, with children, and consider its child nodes in the order in which they were attached to x, from earliest to latest. They could have only been attached during a linking step, which happens between two trees of the same degree. Call y the k-th child of x. When y was appended to x, x must have had at least $k-1$ children (even though it may have had more, since deleted) and so y itself must have had at least $k-1$ children itself at that time, because it had to be of x's degree in order to be linked to x. Since then, y may have lost at most one child but not more—otherwise, thanks to the peculiar constraint, it would have been cut off and made into a root. Therefore (thesis) the k-th child of x has at least $k-2$ children.

Now let's define the function $N(j)$ as the minimum possible number of nodes of a subtree rooted at a node x of degree j and let's seek a closed expression for it. If node x has degree 0, meaning no children, then the whole subtree contains just the original node x: $N(0) = 1$. If node x has degree 1 (one child), the minimum tree is one where that child has no children, so the tree only has these two nodes (x and its child) and $N(1) = 2$. The thesis above tells us how many children each of the children of x from the second onwards must have at least: the second node must have at least 0, the third node at least 1, the fourth node at least 2, the k-th node at least $k-2$. Therefore, assuming that each of these children is itself root of a subtree with the minimum possible number of nodes, the overall minimum number of nodes in the subtree rooted at x will be the sum of all these contributions:

$$N(j) = \overset{\text{root}}{1} + \overset{\text{child 1}}{1} + \overset{\text{child 2}}{N(2-2)} + \overset{\text{child 3}}{N(3-2)} + \ldots + \overset{\text{child } j}{N(j-2)} = 2 + \sum_{l=0}^{j-2} N(l). \tag{1.1}$$

From this we can prove inductively that

$$N(j) = F(j+2) \tag{1.2}$$

where $F(k)$ is the k-th Fibonacci number[18]. For the base of the induction, it's easy to verify that the relation holds for $j = 0$ and $j = 1$; for the inductive step, let's assume it works for j and check whether it does for $j+1$:

$$\begin{aligned}
\text{LHS} &= N(j+1) = \\
&= 2 + \sum_{l=0}^{j-1} N(l) = \\
&= 2 + \sum_{l=0}^{j-2} N(l) + N(j-1) \stackrel{(1.1)}{=} \\
&= N(j) + N(j-1) \stackrel{(1.2)}{=} \\
&= F(j+2) + F(j+1) = \\
&= F(j+3) = \\
&= \text{RHS}
\end{aligned}$$

QED.

It works! So the number of nodes in a subtree of degree j is at least equal to the $(j+2)$-th Fibonacci number. Since the Fibonacci numbers grow exponentially (it can be proved that $F(j+2) \geq \phi^j$), we have that the total number of nodes in a subtree of degree j is exponential in j:

$$N(j) \geq \phi^j$$

and therefore, taking the log of both sides,

$$\log_\phi N(j) \geq j$$

which means that $j \leq K \lg N(j)$ for some constant K. So we have finally proved that, in a Fibonacci heap, if a subtree has n nodes, the root of the subtree has degree $O(\lg n)$.

1.4 van Emde Boas trees

Textbook

Study chapter 20 in CLRS3.

[18] The well-known Fibonacci sequence is defined as $F(0) = 0$, $F(1) = 1$ and $F(i+2) = F(i) + F(i+1)$. Each number is the sum of its two predecessors: 0, 1, 1, 2, 3, 5, 8, 13, 21... The sequence occurs in many contexts in nature and art and the ratio of two successive numbers converges to the irrational number $\phi = \frac{1+\sqrt{5}}{2}$, known as the golden ratio. Note that this ϕ is totally unrelated to the potential Φ introduced in section 1.2.2.

Just when you thought that the Fibonacci heaps were the asymptotically fastest and most elaborate way of implementing a priority queue, here comes another amazing data structure that goes even further—the van Emde Boas tree, or "vEB tree".

The Fibonacci heap already has amortized constant cost for most of its operations and it's hard to improve on that; but is still has amortized $O(\lg n)$ costs for `extractMin()` and `delete()`. The vEB tree manages to bring those costs down to an amazing $O(\lg \lg n)$! The three main trade-offs are:

1. *all* operations cost $O(\lg \lg n)$, even the ones that were $O(1)$ with the Fibonacci heap;
2. the vEB tree requires the keys to be (unique) integers of a specified maximum size in bits;
3. unlike the Fibonacci heap, the vEB tree does not offer an efficient merge operation.

The interface exposed by the vEB tree is essentially that of the ordered dynamic set[19]: `member()`, `insert()`, `delete()`, `min()`, `max()`, `pred()`, `succ()`. All these operations are performed in $O(\lg \lg n)$ time and, by combining them in the obvious way, one can also perform in $O(\lg \lg n)$ time the typical priority queue operations of `extractMin()`, `first()`, `decreaseKey()`.

> **Exercise 6**
> Assume that `insert()`, `delete()`, `min()`, `max()`, `pred()`, `succ()` all have complexity $O(f(n))$. Define `extractMin()`, `first()` and `decreaseKey()` in terms of the previous primitives, without exceeding $O(f(n))$ complexity.

But the vEB tree is a pretty elaborate data structure so we'll get to it in stages rather than describing the final product straight away. First, though, instead of speaking of n, the number of keys held in the priority queue, we'll speak of u, the size of the universe of keys, or in other words the number of *possible* keys in the data structure (for example $u = 2^{64}$, if each key is a 64-bit integer). The costs for the vEB tree, unlike those for the Fibonacci heap, are in fact a function of u (the number of distinct keys that the data structure could potentially hold), not of n (the number of keys actually stored in the data structure); so, to be more precise, with this definition we should have said in the above that the costs of the dynamic set operations supported by the vEB tree are $O(\lg \lg u)$.

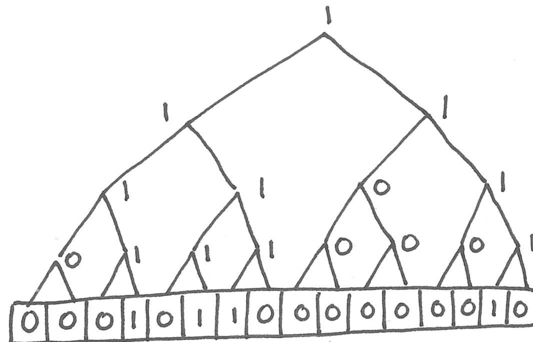
1.4.1 Array

The first attempt to speed up operations such as `insert()` and `delete()` is to use an array of bits of size u, with `A[k] == 1` if and only if the item whose key is k is in the data structure; then `insert()` and `delete()` will cost $O(1)$. But `min()`, `max()`, `pred()`, `succ()` will cost an unacceptable $O(u)$, because we'll have to scan the array linearly, skipping all the 0s, until we find the next 1.

[19]Except that we disregard satellite data because all that matters here is the keys themselves. Indeed, we don't even have a `search` method but just a `member` method that says whether a given key is in the data structure or not.

1.4.2 Binary tree

We can augment the array with a binary tree whose leaves are the array entries. Each node of the tree is labelled with one "summary" bit for its subtree, which is 1 iff the subtree rooted there has any nonzero leaves. In other words, the bit value of each node is the OR of the values of its two children. This allows us to skip empty contiguous regions more efficiently. For example, to find the minimum, start from the root and follow the leftmost "1" pointers until you get to a leaf. The cost is the height of the tree, i.e. $O(\lg u)$.



Finding the successor (the next cell with a "1") of a designated value[20] is slightly more elaborate. You go up until the up-arrow goes right and the other (right) child has a "1" summary[21]. Then you go down that right child of that ancestor and find the minimum of that subtree. The worst possible cost is twice the height of the tree, so again $O(\lg u)$.

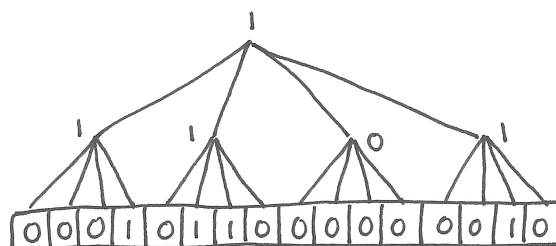
Inserting a new leaf is easy: set it and all its ancestors to 1, again at cost $O(\lg u)$. Deleting is not quite symmetrical, though: you set the leaf to 0 but then, as you go up the tree, the value of each ancestor must be recomputed as the OR of its two children. The cost remains $O(\lg u)$, since the work per node is constant.

1.4.3 Two-level tree

In an attempt to speed up operations even further, we force the tree to have a constant height of just two levels (instead of a variable height of $\lg u$ levels). Each node of the tree, root included, must therefore have $\sqrt{u}$ children. Again, each node stores one bit which is the OR of all its children. The procedures to find the minimum (maximum) or the successor (predecessor) are conceptually the same as before, but because the tree has constant height the total cost is $O(\sqrt{u})$.

[20] Note that the successor of x can be sought regardless of whether x is or isn't a member of the data structure, i.e. without cell x necessarily having to be set to 1 itself.

[21] If you go up and reach the root without the up-arrow ever going right, you were already at the rightmost cell of the array so you have no successor. If the up-arrow eventually goes right but the right child has a "0" summary, then there are no bits set after your position and therefore you have no successor either.



The procedures for inserting and deleting are also conceptually the same as before, but deletion is more expensive.

Exercise 7
How much do insertion and deletion cost for the two-level tree? Why?

1.4.4 Proto-vEB tree

We continue with the idea of a tree of degree $\sqrt{u}$, but this time we make the data structure recursive: if u is the size of the universe of keys at a given level in the structure, at the next level down the structure shrinks by a factor of $\sqrt{u}$. Imagine for example that, at the top level, $u = 2^{64}$. Then the root node, to cater for $u = 2^{64}$ leaves, will have $\sqrt{u} = 2^{32}$ children, each with $\sqrt{u} = 2^{32}$ leaves. From the point of view of one of these children, the size of the universe of keys will be $u = 2^{32}$ (that's a new u for this level), so that node will in turn have $\sqrt{u} = 2^{16}$ children, each with $\sqrt{u} = 2^{16}$ leaves. And so forth until the base level where $u = 2$.

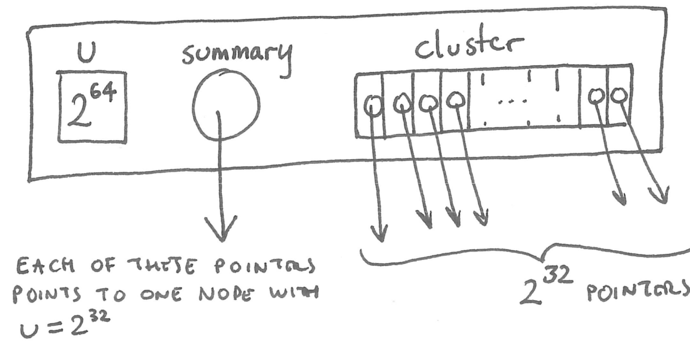
We also rearrange the tree slightly: previously, each node contained one summary bit with the OR of all its children. Now, although the same information is still there, we store it somewhere else. Node (cluster) n, of size u, has $\sqrt{u}$ children of size $\sqrt{u}$. It also has another (anomalous) child, called "summary", which itself is a proto-vEB node of size $\sqrt{u}$ (structurally identical to all the other children of n) but whose values are the indices of the children of n (clusters of size $\sqrt{u}$) that are not empty. And each node restarts numbering its own children from 0.

Each proto-vEB node with $u > 2$ thus contains the following three fields:

`u`: an integer giving the size of the universe of keys for this node; this node can store all integers from 0 to $u - 1$.

`summary`: a pointer to a proto-vEB node of size $\sqrt{u}$ containing one summary bit for each child of the current node.

`cluster[]`: an array with $\sqrt{u}$ cells, each being a pointer to a proto-vEB node of size $\sqrt{u}$.



At the bottom level, the proto-vEB node with $u = 2$ does not have the summary pointer and stores actual bits, rather than pointers, in the two cells of its cluster.

We assume for simplicity that $u = 2^{2^k}$ for some integer k, so that the square root of u is always an integer, at all levels in the tree. Another way of saying the same thing is that the number of bits of u is a power of 2 (at all levels) and that therefore it is always possible to split the bit string of a key in two halves of equal length. Given any key $x \in [0, u-1]$, if we call `high(x)`, or h, the top half of the binary representation of x and `low(x)`, or l, the bottom half[22], then x is stored in the h-th child of the root node and it is key number l within that node. (We use the function `index(h, l)` to join back the two halves: `index(high(x), low(x)) == x`.)

For example, if a proto-vEB tree of size $u = 2^4$ stores the key 13 (binary 1101), the root node (which has $\sqrt{u} = 2^2$ clusters with $\sqrt{u} = 2^2$ keys each) stores it in its cluster number 3 (binary 11, the most significant half of 1101) and at position 1 (binary 01) in that cluster. Note well how cluster number 3 has a size of 4, not 16, and thus only holds keys from 0 to 3. The key it holds is indeed 1, not 13, which wouldn't fit. You can only reconstruct the full value of the key if you look at the whole path: you can't read it off from the leaf. (Remember that as you do the following exercise.)

Exercise 8
Sketch a picture of a proto-vEB tree of size $u = 16$ representing the set 2, 3, 4, 5, 7, 14, 15.

Where did these weird ideas come from? Why are we doing this at all? The objective is to attain the $O(\lg \lg u)$ complexity bound. We can show that the recurrence

$$T(u) = T(\sqrt{u}) + O(1)$$

is solved by the desired

$$T(u) = O(\lg \lg u)$$

and this suggests that, to attain that recurrence, we should build a recursive structure where the size of each node is the square root of the size of the parent. Then if a recursive

[22]Note also that, in reality, the functions `high()` and `low()` must take u as a second parameter, in order to know how many bits to chop off.

procedure for, say, finding the minimum, did on each level an amount of work proportional to the size of a node at that level, it would be described by the above recurrence and would achieve the desired running time.

Let's first show, informally, that the above recurrence is indeed solved by the given function. We use the substitution $u = 2^m$, implying $m = \lg u$.

$$\begin{aligned}
T(u) &= T(\sqrt{u}) + O(1) \\
T(2^m) &= T(2^{\frac{m}{2}}) + O(1) \\
&= T(2^{\frac{m}{4}}) + 2O(1) \\
&= T(2^{\frac{m}{8}}) + 3O(1) \\
&= T(2^{\frac{m}{16}}) + 4O(1) \\
&= T(2^{\frac{m}{2^l}}) + lO(1) \\
&= T(2^{\frac{m}{m}}) + \lg m \cdot O(1) \\
&= T(2) + O(1)\lg m \\
&= O(\lg m) \\
&= O(\lg\lg u)
\end{aligned}$$

QED.

Let's now look specifically at finding the minimum.

```
0  def min(self):
1      """In this class method for a protoVEB object,
2      self is a protovEB node of size u.
3      Return the minimum key in self, or None if self holds no keys."""
4
5      if self.u == 2:
6          handle base case by brute force at constant cost
7      else:
8          minCluster = self.summary.min()
9          if minCluster == None:
10             return None
11         else:
12             h = minCluster
13             l = self.cluster[minCluster].min()
14             return index(h, l)
```

We see that the worst-case costs involve *two* recursive calls of `min()` from within itself (lines 8 and 13), on proto-vEB nodes of size $\sqrt{u}$. This yields the recurrence

$$T(u) = 2T(\sqrt{u}) + O(1)$$

which unfortunately solves to $T(u) = O(\lg u)$, not $O(\lg\lg u)$.

Exercise 9
Prove that the recurrence

$$T(u) = 2T(\sqrt{u}) + O(1)$$

has the solution

$$T(u) = O(\lg u).$$

Some of the other data structure methods have even worse performance. Consider for example the case of looking for the successor of a given key. As with the two-level tree, we go up until the ancestor node has a child to the right of us, and that child's summary bit is 1. Then we go down that child, all the way to the leaves, looking for the minimum. To obtain a recurrence for this operation, let's rephrase it as a top-down recursive procedure that matches the hierarchy of the data structure: the successor of k, if it exists, is either k's successor within the cluster that contains k (in case k is not the maximum of its own cluster), or it's the minimum of the next cluster that contains anything.

```
0  def successor(self, k):
1      """In this method, self is a protovEB node of size u.
2      k is a key in the range 0..u-1.
3      Return the key that is the successor of k in self,
4      or None if there is none."""
5
6      if self.u == 2:
7          handle base case by brute force at constant cost
8      else:
9          h = high(k, self.u)
10         l = low(k, self.u)
11         successorInCluster = self.cluster[h].successor(l)
12         if successorInCluster == None:
13             nextCluster = self.summary.successor(h)
14             if nextCluster == None:
15                 return None
16             else:
17                 return nextCluster.min()
18         else:
19             return successorInCluster
```

Here we see that, in the worst case, the `successor()` method calls itself twice, in lines 11 and 13, each time on a node of size $\sqrt{u}$, but then also invokes `min()` on a node of size $\sqrt{u}$, in line 17. This gives us a recurrence of $T(u) = 2T(\sqrt{u}) + O(\lg \sqrt{u})$, where the factor of 2 in front of the T term comes from the two recursive invocations of `successor()` from within itself, while the $O(\lg \sqrt{u})$ is the cost of `min()` as derived above.

Exercise 10
Prove that the recurrence

$$T(u) = 2T(\sqrt{u}) + O(\lg \sqrt{u})$$

has the solution

$$T(u) = O(\lg u \lg \lg u).$$

Therefore, with the proto-vEB tree, computing the successor is asymptotically slower than computing the minimum.

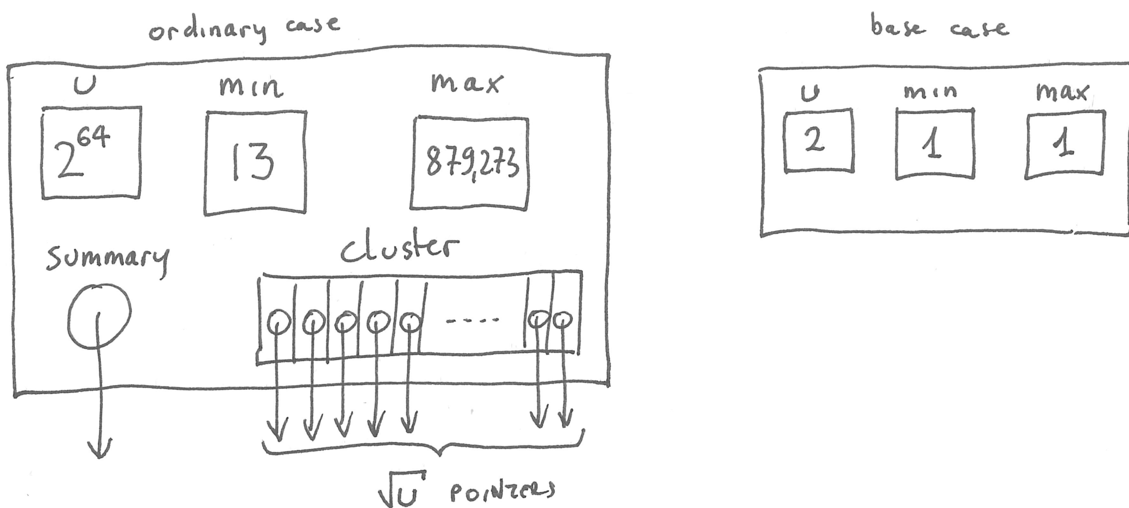
But, despite these disappointments, the proto-vEB tree still gives us hints of what to do next.

1.4.5 vEB tree

The vEB tree can be thought of as a proto-vEB tree optimized so that its methods have to make at most one recursive call. This is done to ensure that the recurrence does not gain a multiplying factor in front of the inner $T(\sqrt{u})$.

The variations to achieve this result are as follows.

- The vEB node contains all the fields of the proto-vEB node, plus two more that store the minimum and maximum key held in the node and its descendants.
- The minimum key for the node is actually only stored in the `min` field; it is *not* also stored in any of the clusters. The maximum, instead, is. (Can be confusing.)
- If a node contains no values, both the `min` and the `max` are set to `None`.
- In the base case of $u = 2$, the node does not have clusters nor summary. Its content can be deduced simply by observing its `min` and `max`.



Exercise 11
Deceptively difficult. Do not skip.
Sketch a picture of a vEB tree of size $u = 16$ representing the set 2, 3, 4, 5, 7, 14, 15. OK to study the textbook and handout first, but keep them closed while doing this exercise. No matter how good you are, you will almost certainly get some details wrong. That's OK. Check the textbook *after* having drawn your solution and mark in red all the items that are different, understanding why. Then *the next day* do the exercise again, with textbook closed. Iterate until you get no errors. Will take several days (no shame in that).
You may think you understand vEB trees but you actually don't until you successfully complete this exercise.

CLRS3 also describes another trick that allows $\lg u$ to be odd instead of even, but we are not going to worry about that detail in this course. It's just formal rules to decide which of the two "halves", h and l, gets the extra bit (the most significant part, h, is the one that ends up with one more bit if there is an odd number of them in $\lg u$).

What happens to the class methods and their costs with these customizations?

The minimum and maximum are trivially read off the corresponding fields of the node, so the `min()` and `max()` methods now only cost $O(1)$. Of course, whenever a data structure caches some information, we must be suspicious and wonder how much it costs to keep this cache up to date. So, what happens with `insert()`? If the node is empty, insert the value directly (into both the `min` and `max` fields, which previously held `None`, but without touching `summary` or `cluster` because the minimum is not stored in the clusters) at constant cost. Otherwise, insert the value `h|l` into both `cluster` and `summary`. However only one of these two calls is recursive: if `cluster[h]` was empty, insertion of `l` into `cluster[h]` has constant cost[23] and insertion of `h` into `summary` is recursive. If, on the other hand, `cluster[h]` wasn't empty, insertion of `l` into `cluster[h]` is recursive but insertion of `h` into `summary` is not even needed because `summary` necessarily already contained `h`. The resulting recurrence is $T(u) = T(\sqrt{u}) + O(1)$, which is $O(\lg \lg u)$.

What about finding the successor?

[23] It's just a matter of setting the `min` and `max` to `l`.

```
0  def successor(self, k):
1      """In this method, self is a vEB node of size u.
2      k is a key in the range 0..u-1.
3      Return the key that is the successor of k in self,
4      or None if there is none."""
5
6      if self.u == 2:
7          handle base case by brute force at constant cost
8      elif self.min != None and k < self.min:
9          return self.min
10         # NB: OK to ask for successor of a key not in self
11     else:
12         h = high(k, self.u)
13         l = low(k, self.u)
14         maxInCluster = self.cluster[h].max
15         if maxInCluster != None and l < maxInCluster:
16             # the successor of k is in k's own cluster
17             return index(h, self.cluster[h].successor(l))
18         else:
19             # the successor of k is in another cluster
20             nextCluster = self.summary.successor(h)
21             if nextCluster == None:
22                 return None
23             else:
24                 return index(nextCluster, self.cluster[nextCluster].min)
```

Does cluster h (the one containing k) also contain k's successor? With a proper vBE we can tell without a recursive call, just by checking with a single lookup (line 14) whether k is the maximum of its cluster (line 15). We then call `successor()` recursively only once per invocation, either on the key's own cluster in the "then" branch (line 17) or on the summary in the "else" branch (line 20). No path through this method has more than one recursive call of `successor()`, so the recurrence is again $T(u) = T(\sqrt{u}) + O(1)$, which is $O(\lg \lg u)$.

Deletion is more complicated (we won't reproduce a listing here but there is one in CLRS3), and so is its analysis because there is a possible path through the procedure in which two recursive calls are made. However it is possible to prove that, when this happens, one of the two calls takes constant time. Therefore the recurrence that applies is still $T(u) = T(\sqrt{u}) + O(1)$, which is $O(\lg \lg u)$.

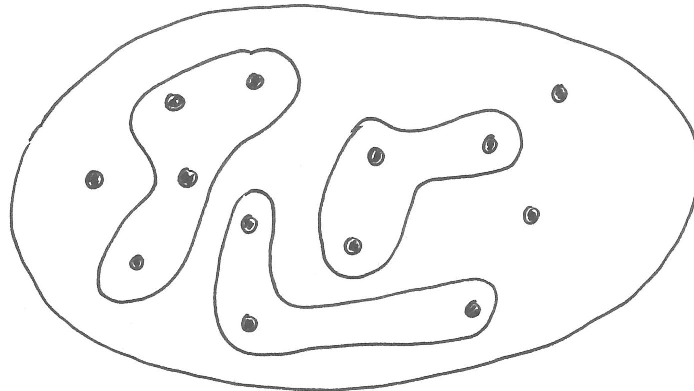
1.5 Disjoint sets

Textbook

Study chapter 21 in CLRS3.

The disjoint set data structure, sometimes also known as union-find or merge-find, is

used to keep track of a dynamic collection of disjoint sets (sets with no common elements) over a given universe of elements.



An example application can be seen in Kruskal's minimum spanning tree algorithm (section 2.3.1), where this data structure is used to keep track of the connected components of a forest: every connected component is stored as the set of its vertices and checking whether two vertices belong to the same disjoint set in the collection or to two different ones tells us whether adding an edge between them would introduce a cycle in the forest or not. Most other uses of this data structure are still generally related to keeping track of equivalence relations[24].

The available operations on a `DisjointSet` object, which despite its name is actually a *collection* of disjoint sets, allow us to add a new singleton set to the collection by supplying the element it contains (`makeSet()`), find the set that contains a given element (`findSet()`), and finally form the union of two sets in the collection (`union()`).

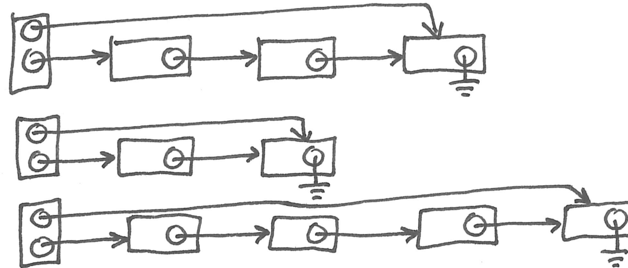
Conceptually, the `makeSet()` and `findSet()` methods return a "handle", which is anything that can act as a unique identifier for a set. If you invoke `findSet()` twice without modifying the set between the two calls, you will get the same handle. In practice, an implementer might choose to use one of the set elements as the handle.

```
0  ADT DisjointSet {
1    Handle makeSet(Item x);
2    // PRECONDITION: none of the existing sets already contains x.
3    // BEHAVIOUR: create a new set {x} and return its handle.
4
5    Handle findSet(Item x);
6    // PRECONDITION: there exists a set that contains x.
7    // BEHAVIOUR: return the handle of the set that contains x.
8
9    Handle union(Handle x, Handle y);
10   // PRECONDITION: x != y.
11   // BEHAVIOUR: merge those two sets into one and return handle of new set.
12   // Nothing is said about whether the returned handle matches either
13   // or none of the supplied ones (i.e. whether set x absorbs y,
14   // set y absorbs x or both sets get destroyed and a new one created).
15 }
```

[24]An equivalence relation is one that is reflexive, symmetric and transitive, such as "is of the same color as" or "has the same mother as" or "costs the same as".

1.5.1 List implementation

A relatively simple implementation of this data structure uses linked lists to store the sets. The `makeSet()` method creates a new one-item list and takes constant time. The `union()` method, too, can be made to take constant time if the `DisjointSet` object maintains, for each list in the collection, a pointer to the last element.



The `findSet()` method requires finding the head of the list that contains the given element[25]; however, in a singly-linked list, this cannot be done by following the list pointers because they go *away* from the list head, not towards it. If we add a further pointer from each list node to the head of the corresponding list, then we can implement `findSet()` in $O(1)$, but then `union()` is no longer $O(1)$ because we must update all the pointers-to-head of all the nodes of the list being appended. Indeed, one can easily construct pessimal input sequences for which the cost of $O(n)$ operations is $\Theta(n^2)$, yielding an amortized cost of $O(n)$ for each disjoint set operation. The following example does. Perform a sequence of $2n-1$ operations, of which n are `makeSet()` and $n-1$ are `union()`, according to the following pattern: keep alternating between making a new set and appending the long list with all the previous elements to the short one of the newly-made singleton set[26].

```
d = DisjointSet()
h0 = d.makeSet(x0)

h1 = d.makeSet(x1)
h0 = d.union(h0, h1)
h2 = d.makeSet(x2)
h0 = d.union(h0, h2)
h3 = d.makeSet(x3)
h0 = d.union(h0, h3)
h4 = d.makeSet(h4)
h0 = d.union(h0, h4)
  .
  .
  .
```

[25]Remember (see the comments that open this chapter on page 7) that the `Item x` passed by the caller to the `findSet()` method is actually already a pointer to the list node that contains the set element of interest—you don't have to worry about finding the required set element in the data structure.

[26]For any given implementation following the above naïve strategy, you can always achieve this pessimal append by appropriately choosing the order of the parameters in the call to `union()`.

As the example implicitly suggests, the smart thing to do is instead to append the shorter list to the longer one; doing this requires us to keep track of the length of each list, but this is not a major overhead. With such a **weighted union** heuristic, it is possible to prove that the cost of a sequence of m operations on n elements[27] goes down to $O(m + n \lg n)$ time.

1.5.2 Forest implementation

A more elaborate representation stores each set in a separate tree (rather than list), with each node pointing to its parent. The `makeSet()` method creates a new root-only tree at cost $O(1)$, while `findSet()` returns the root of the tree by navigating up the edges of the tree, at cost $O(h)$ where h is the height of the tree. The `union()` operation appends the first tree to the second by making the root of the first tree a child of the root of the second[28], at cost $O(1)$.

With the same kind of reasoning that suggested the weighted union heuristic, we may improve the performance of `union()` by ensuring that the operation won't generate unnecessarily tall trees: we do this by keeping track of the **rank** of each tree (an upper bound on the height of the tree) and always appending the tree of smaller[29] rank to the other. This **union by rank** heuristic ensures that the rank of the resulting tree is either the same as that of the taller of the two trees or, at worst, one greater, if the two original trees had the same rank. In other words, the maximum rank of the trees in the collection grows as slowly as possible.

Another speedup is obtained by adopting the **path compression** heuristic, which at every `findSet(x)` "flattens" the path from `x` to the root of the tree. In other words, `x` and all the intermediate nodes between it and the root are reparented to become direct children of the root. Stored ranks are not adjusted (which is why they end up being only upper bounds on the heights of the respective trees, rather than exact values). This operation costs no more than the $O(h)$ of the original `findSet()`, asymptotically, since all these nodes had to be visited in order to find the root anyway.

Exercise 12
If we are so obsessed with keeping down the height of all these trees, why don't we just maintain all trees at height ≤ 1 all along?

It can be shown with some effort that, if we adopt both "union by rank" and "path compression", the cost of a sequence of m operations on n items is $O(m \cdot \alpha(n))$, where $\alpha()$ is an integer-valued monotonically increasing function (related to the Ackermann function) that grows extremely slowly. Since $\alpha(n)$ is still only equal to 4 for n equal to billions of billions of times the number of atoms in the observable universe, it can be assumed that, for practical applications of a disjoint set, the value $\alpha(n)$ is bounded by the constant

[27] In other words, a sequence of m operations, with $m > n$, of which n are `makeSet()` and $m - n$ are `findSet()` or `union()`.

[28] These trees are not binary.

[29] Or equal.

value 5 and may be ignored in the O notation. Therefore, “for all practical purposes”, the performance of the forest implementation of the disjoint set with these two heuristics on a sequence of m operations is $O(m)$, meaning that the *amortized* cost per operation is constant.

Chapter 2

Graph algorithms

Chapter contents

Graph representations. Breadth-first and depth-first search. Topological sort. Minimum spanning tree. Kruskal and Prim algorithms. Shortest paths. Bellman-Ford and Dijkstra algorithms. Maximum flow. Ford-Fulkerson algorithm. Matchings in bipartite graphs.
Expected coverage: about 5 lectures.

Graph theory can be said to have originated in 1736, when Euler proved the impossibility of constructing a closed path that would cross exactly once each of the 7 bridges of the city of Königsberg (Prussia).

Exercise 13
Build a "7-bridge" graph with this property. Then look up Euler and Königsberg and check whether your graph is or isn't isomorphic to the historical one. (Hint: you don't *have* to reconstruct the historical layout but note for your information that the river Pregel, which traversed the city of Königsberg, included two islands, which were connected by some of the 7 bridges.) Finally, build the *smallest* graph you can find that has this property.

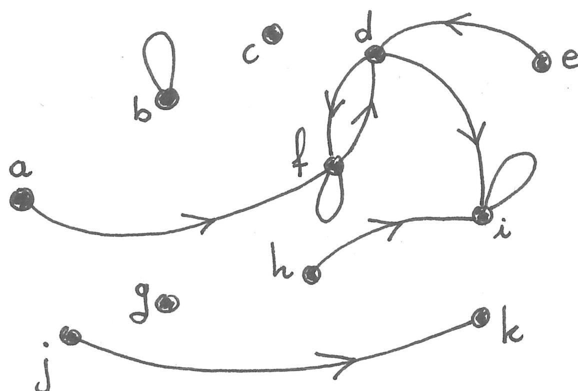
2.1 Basics

Textbook

Study chapter 22 in CLRS3.

2.1.1 Definitions

The general heading "graphs" covers a number of useful variations. Perhaps the simplest case is that of a general **directed** graph: this has a set V of vertices and a set E of ordered pairs of vertices that are taken to stand for directed edges. Such a graph is isomorphic to, and is therefore a possible representation for, a relation $R : V \to V$. Note that it is common to demand that the ordered pairs of vertices be all distinct, and this rules out having parallel edges. In some cases it may also be useful either to assume that for each vertex v, the edge (v, v) is present (in which case the relation is **reflexive**), or to demand that no edges joining any vertex to itself can appear (in which case the relation is **antireflexive**). The graph is called **undirected** when the edges have no arrows, i.e. when (v_1, v_2) is the same edge as (v_2, v_1), as is the case for the graph of the Königsberg bridges problem.



A sequence of zero or more edges[1][2] from vertex u to vertex v in a graph, indicated as $u \rightsquigarrow v$, forms a **path**. If each pair of vertices in the entire graph has a path linking them, then the graph is **connected**. A non-trivial[3] path from a vertex back to itself is called a **cycle**. Graphs without cycles have special importance, and the abbreviation **DAG** stands for Directed Acyclic Graph. An undirected graph without cycles is a **tree**[4], but not vice versa (because some trees are directed graphs). A directed tree is always a DAG, but not vice versa (think of the frequently-occurring diamond-shaped DAG). If the set of vertices V of a graph can be partitioned into two sets, L and R say, and each edge of the graph has one end in L and the other in R, then the graph is said to be **bipartite**.

[1] Zero when $u \equiv v$. Conversely, when a path $u \rightsquigarrow v$ consists of precisely one edge, we iron out the squiggles in the arrow and indicate it as $u \to v$.

[2] If the graph is directed, then all the edges of a path must go in the same direction. In other words, $a \to b \leftarrow c$ is not a path between a and c.

[3] Here, by "non-trivial" we mean a path with more than one edge; when a cycle has only one edge we call it a **loop**.

[4] Or a **forest** if made of several disconnected components.

Exercise 14
Draw in the margin an example of each of the following:

1. An anti-reflexive directed graph with 5 vertices and 7 edges.
2. A reflexive directed graph with 5 vertices and 12 edges.
3. A DAG with 8 vertices and 10 edges.
4. An undirected tree with 8 vertices and 10 edges.
5. A tree that is *not* "an undirected graph without cycles".
6. A graph without cycles that is *not* a tree.

Actually, I cheated. Some of these can't actually exist. Which ones? Why?

The definition of a graph can be extended to allow values to be associated with each edge—these will usually be called **weights** even though they may represent other concepts such as distances, carrying capacities, costs, impedances and so on. Graphs can be used to represent a great many things, from road networks to register use in optimizing compilers, to databases, to electrical circuits, to timetable constraints, to web pages and hyperlinks between them. The algorithms using them discussed here are only the tip of an important iceberg.

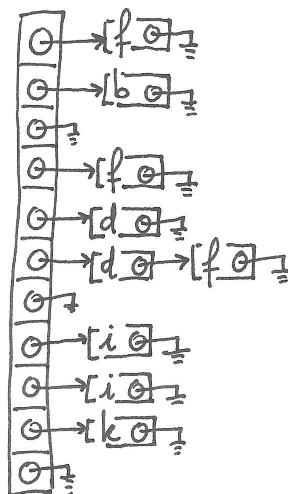
2.1.2 Graph representations

Barring specialized representations for particular applications, the two most obvious ways to represent a graph inside a computer are with an adjacency matrix or with adjacency lists.

	a	b	c	d	e	f	g	h	i	j	k
a						1					
b		1									
c											
d						1					
e				1							
f				1		1					
g									1		
h									1		
i											1
j											
k											

The **adjacency matrix**, especially suitable for **dense** graphs[5], is a square $|V| \times |V|$ matrix W in which w_{ij} gives the weight of the edge from vertex i to vertex j (or, for unweighted graphs, a boolean value indicating the presence or absence of an edge). This format makes manipulation easy but obviously has $O(|V|^2)$ storage costs.

Therefore, for **sparse** graphs[5], a more economical solution is often preferred: for each vertex v, an **adjacency list** (usually unsorted) contains the vertices that can be reached from v in one hop. For weighted graphs, the weight of the corresponding edge is recorded alongside each destination vertex.



A comment on notation: the computational complexity of a graph algorithm will usually be a function of the cardinality of the sets V and E; however, given that E and V on their own as sets have no meaning in a big-O formula, writing down the cardinality bars has no disambiguating function and only makes the formulae longer and less legible, as in $O(|E| + |V| \lg |V|)$. Therefore the convention is to omit the bars within the big-O notation and write instead things like $O(E + V \lg V)$.

2.1.3 Searching (breadth-first and depth-first)

Many problems involving graphs are solved by systematically searching. This usually means following graph edges so as to visit all the vertices. Note that we assume that we have access to a list of all the vertices of the graph and that we can access it regardless of the arrangement of the graph's edges; this way, when a search algorithm stops before visiting all the vertices, we can restart it from one of the vertices still to be visited. This "backdoor" does not, as some might think, defeat the point of graph searching: in general, the objective is not merely to access the vertices but to visit them according to the structure of the graph—for example to establish what is the maximum distance between any two vertices in the graph[6], which by the way is known as the **diameter** of the graph.

[5] A dense graph is one with many edges ($|E| \approx |V|^2$) and a sparse graph is one with few ($|E| \ll |V|^2$).

[6] I should be more precise when saying something like this. In this instance I am referring to a "maximum of minimums"; in other words, imagine to compute, for every two vertices in the graph, the shortest path between them; then I want the longest of these shortest paths. Otherwise the definition of *maximum distance* might be inconsistent, as one might be able to take arbitrarily many detours round

The two main strategies for inspecting a graph are *depth-first* and *breadth-first*. In both cases we may describe the search algorithm as a vertex colouring procedure. All vertices start out as white, the virginal colour of vertices that have never been visited. One vertex is chosen as source of the search: from there, other vertices are explored following graph edges. Each vertex is coloured grey as soon as it is first visited, and then black once we have visited all its adjacent vertices. Depending on the structure of the graph, for example if a graph has several disconnected components, it may be necessary to select another source if the original one has been made black but there still exist white vertices.

```
0  def bfs(G, s):
1      """Run the breadth-first search algorithm on the given graph G
2      starting from the given source vertex s."""
3
4      assert(s in G.vertices())
5
6      # Initialize graph and queue:
7      for v in G.vertices():
8          v.predecessor = None
9          v.d = Infinity # .d = distance from source
10         v.colour = "white"
11     Q = Queue()
12
13     # Visit source vertex
14     s.d = 0
15     s.colour = "grey"
16     Q.insert(s)
17
18     # Visit the adjacents of each vertex in the queue
19     while not Q.isEmpty():
20         u = Q.extract()
21         assert (u.colour == "grey")
22         for v in u.adjacent():
23             if v.colour == "white":
24                 v.colour = "grey"
25                 v.d = u.d + 1
26                 v.predecessor = u
27                 Q.insert(v)
28         u.colour = "black"
```

In the case of **breadth-first** search, from any given vertex we visit all the adjacent vertices in turn before going any deeper. This is achieved by inserting each vertex into a queue as soon as it gets visited and, for each vertex extracted from the queue, by visiting all its unvisited (white) adjacent vertices before considering it done (black). Only white vertices are considered for insertion in the queue and they get painted grey on insertion and black on extraction; as a result, the queue contains only grey vertices and there are

the cycles of the graph while navigating from one vertex to the other. I should also specify whether the length of a path is considered to be the number of edges on it, as is the case for the graph diameter; or—as indeed makes more sense for many other applications—the sum of the weights of the edges on it.

no grey vertices other than those in the queue. Consider the colour grey as indicating that a vertex has been visited by the procedure, but not completed.

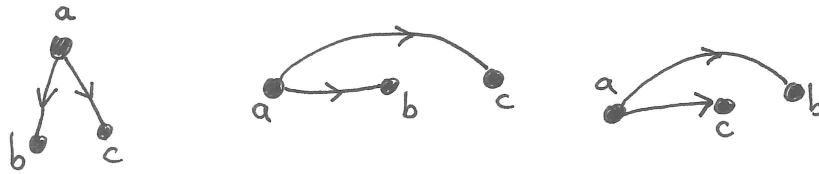
Depth-first search, instead, corresponds to the most natural recursive procedure for walking over a tree: from any particular vertex, the whole of one sub-tree is investigated before any others are looked at at all. In the case of a generic graph rather than a tree, a little extra care is necessary to avoid infinite loops—but the colouring, otherwise unnecessary with trees, helps with that.

```
0  def dfs(G, s):
1      """Run the depth-first search algorithm on the given graph G
2      starting from the given source vertex s."""
3
4      assert(s in G.vertices())
5
6      # Initialize graph:
7      for v in G.vertices():
8          v.predecessor = None
9          v.colour = "white"
10
11     dfsRecurse(G, s)
12
13 def dfsRecurse(G, s):
14     s.colour = "grey"
15     s.d = time() # .d = discovery time
16     for v in s.adjacent():
17         if v.colour == "white":
18             v.predecessor = s
19             dfsRecurse(G, v)
20     s.colour = "black"
21     s.f = time() # .f = finish time
```

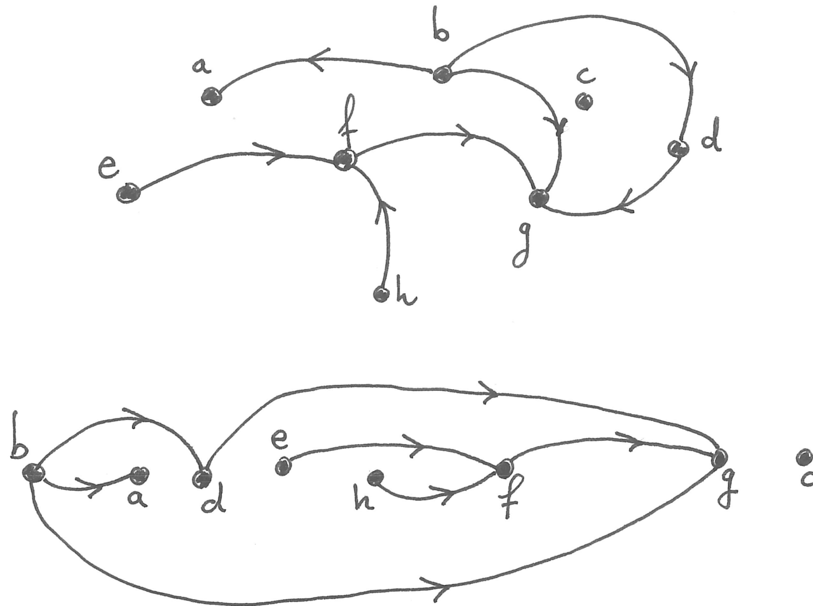
Breadth-first search can often avoid getting lost in fruitless scanning of deep parts of the tree, but the queue that it uses often requires more memory than the stack implicitly used by depth-first search.

2.2 Topological sort

The problem of topological sort is that of linearizing a directed acyclic graph (DAG): you have a DAG and you wish to output its vertices in such an order that "all the arrows go forward", i.e. no edge goes from a vertex v to a vertex u that was output before v. In a project management scenario you might think of the vertices of the DAG as actions and of the edges as dependencies between actions; then the topological sort provides a sequence in which the actions can actually be performed. It ought to be clear that, for a generic DAG, there will usually be more than one valid linearizing sequence for its vertices—a Λ-shaped three-vertex tree provides a trivial example.



Below is a less trivial DAG and one possible linearization for it.



A deceptively simple algorithm, due to Knuth, solves the problem in $O(V+E)$ using a depth-first search. The strategy is to perform an exhaustive depth-first search that visits all the vertices in the graph (restarting from any leftover white vertex if the recursive search returns before having painted all the vertices black) and then to output the vertices in reverse order of finishing time, where the finishing time of a vertex is defined as the time at which it gets coloured black. But why should this work at all?

Exercise 15
Draw a random DAG in the margin, with 9 vertices and 9 edges, and then perform a depth-first search on it, marking each vertex with two numbers as you proceed—the discovery time (the time the vertex went grey), then a slash and the finishing time (the time it went black). Then draw the linearized DAG by arranging the vertices on a line in reverse order of their finishing time and reproducing the appropriate arrows between them. Do the arrows all go forward?

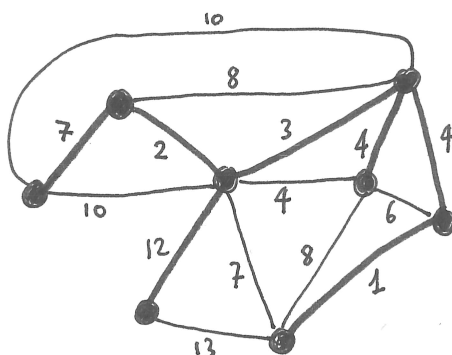
The insight is that, in order to blacken vertex u, I must have already blackened all of its descendents. It is easy to see that this is true for the vertices in u's adjacency list but a little more care is needed to deal with those further down.

Exercise 16
Develop a proof of the correctness of the topological sort algorithm. *(Requires some thought.)*

2.3 Minimum spanning tree

Textbook

Study chapter 23 in CLRS3.



Given a connected undirected graph where the edges have all been labelled with non-negative weights, the problem of finding a **minimum spanning tree** is that of finding the sub-graph of minimum total weight[7] that links all vertices. This must necessarily be a tree. Suppose it isn't: then either it is a forest, in which case it fails to connect all vertices, which contradicts the hypothesis; or it contains a cycle. Removing any one edge from the cycle would leave us with a graph with fewer edges and therefore (since no edges have negative weight) of lower or equal weight, but still connecting all the vertices, again contradicting the hypothesis, QED. Note that a graph may have more than one minimum spanning tree—these would be distinct trees with the same total weight.

Exercise 17
Find, by hand, a minimum spanning tree for the graph drawn in CLRS3 figure 23.4.(a) (trying not to look at nearby figures). Then see if you can find any others.

A generic algorithm that finds minimal spanning subtrees involves growing a subgraph A by adding, at each step, a **safe** edge[8] of the full graph, defined as one that ensures that the resulting subgraph is still a subset of some minimum spanning tree.

[7]Not, obviously, "of minimum number of edges", as that is a trivial problem—*any* tree touching all the vertices is one.

[8]With respect to A.

```
0 def minimumSpanningTree(G):
1     A = empty set of edges
2     while A does not span all vertices yet:
3         add a safe edge to A
```

At this level of abstraction it is still relatively easy to prove that the algorithm is correct, because we are handwaving away the comparatively difficult task of choosing a safe edge.

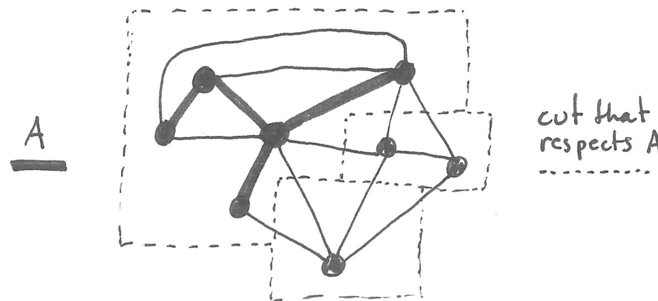
We shall soon describe two instantiations of this generic algorithm that differ in the criterion they use to *choose* a safe edge at each iteration. Before that, though, let's develop a criterion for *recognizing* safe edges.

We start with a couple of definitions. Given a graph $G = (V, E)$, a **cut** is a partition of G's vertices into at least two sets. Given a (possibly non-connected) subgraph A of G, a cut of G **respects** A if and only if no edge of A goes across the cut[9].

Now the theorem: given a graph G and a subgraph A that is a subset of a minimum spanning tree of G, for any cut that respects A, the lightest edge of G that goes across the cut is safe for A.

To prove this, imagine the full minimum spanning tree T of which A is by hypothesis a subgraph and call e_l the lightest edge across the chosen cut. If $e_l \in T$, the theorem is satisfied. Are there any alternatives? If $e_l \notin T$, then adding it to T introduces a loop (because the two endpoints of e_l were already connected to each other through edges of T, by hypothesis of T being a spanning tree of G). For topological reasons, this loop must contain at least one other edge going across the cut and distinct from e_l: call it (or any one of them if there are several) e_x.

I recommend you work out the theorem as you go along on the partial graph below, figuring out where T is and so on. With a pre-completed graph you would learn less than with one you reconstruct by yourself. The highlighted edges are in A, whereas the dotted lines represent the cut.



Now call T_l the spanning tree[10] you get from T by substituting e_l for e_x (note that $e_x \in T$, $e_l \in T_l$). Since e_l is the lightest edge across the cut by hypothesis, either it weighs the same as e_x (in which case T_l is another equally good minimum spanning tree,

[9] We say that an edge "goes across the cut" if and only if the two endpoints of the edge belong to different sets of the partition.

[10] T_l is indeed a tree because we started from a tree T, we added an edge between two distinct vertices, thus introducing a loop, then removed another edge of that loop, yielding another tree over the same vertices. It is a spanning tree because it still connects the same set of vertices as T, which was a spanning tree.

and therefore e_l is safe since it belongs to an MST and the theorem is satisfied), or e_l weighs strictly less than e_x (but then T_l is a spanning tree of smaller weight than T, which contradicts the hypothesis that T is a *minimum* spanning tree, so this case can't happen). QED.

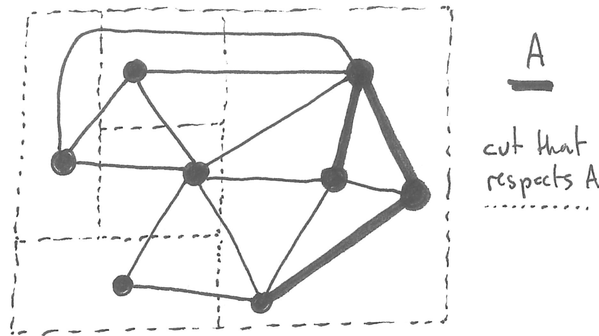
Note that, although for a given cut there is only one safe *edge* (namely the lightest, unless there are several *ex aequo* minimum weight edges across the cut), one still has a wide choice of possible *cuts* that respect the subgraph A formed so far. The two algorithms that follow, as announced, use this freedom in different ways.

2.3.1 Kruskal's algorithm

Kruskal's algorithm allows A to be a forest during execution. The algorithm maintains a set A of edges, initially empty, that will eventually become the MST. The initial cut is one that partitions each vertex into a separate set. The edge to be added at each step is the lightest one that does not add a cycle to A. Once an edge is added to A, the sets of its endpoints edge are merged, so that the partition contains one fewer set. In other words the chosen cut is, at all times, the one that partitions each connected[11] cluster of vertices into its own set.

```
0  def kruskal(G):
1      """Apply Kruskal's algorithm to graph G.
2      Return a set of edges from G that form an MST."""
3
4      A = Set() # The edges of the MST so far; initially empty.
5      D = DisjointSet()
6      for v in G.vertices():
7          D.makeSet(v)
8      E = G.edges()
9      E.sort(key=weight, direction=ascending)
10
11     for edge in E:
12         startSet = D.findSet(edge.start)
13         endSet = D.findSet(edge.end)
14         if startSet != endSet:
15             A.append(edge)
16             D.union(startSet, endSet)
17     return A
```

[11]Connected via edges in $A \subset E$, that is—not merely connected via any edges in E.



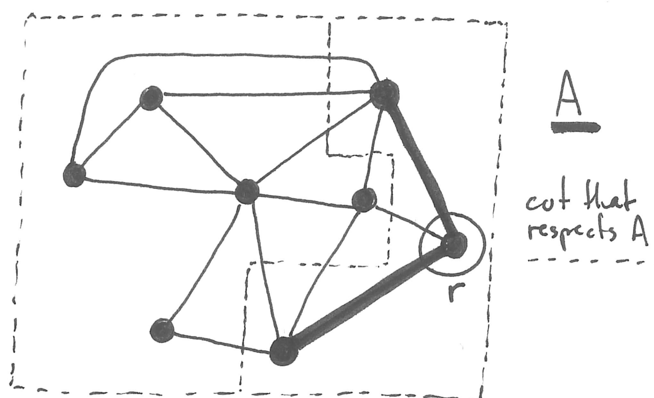
In the main `for` loop starting on line 11, all the edges of the graph are analyzed in increasing order of weight. Any that don't add a loop to the ones already chosen are selected and brought into A, the MST-so-far. To find out whether a candidate edge would form a cycle when added to the edges already in set A, we maintain a disjoint-set D (see section 1.5) which has a set for each connected component of the forest being formed in A. If the candidate edge has its endpoints in two separate components, then it won't add a cycle and it is safe to take it as an edge of the MST-so-far (with appropriate updating of D, since it will connect the two previously disconnected components); otherwise that edge would form a cycle and so it must be rejected.

The cost of this main loop (lines 11–16) is easily estimated: $|E|$ times the cost of two `findSet()` and one `union()`. With the asymptotically-fastest disjoint-set implementation known, a sequence of any m `makeSet()`, `findSet()` and `union()` operations will take $O(m\alpha(n))$, or in practice[12] just $O(m)$ which in our case is $O(V + 3E) = O(E)$. So the main loop actually costs even less than the setting-up operations in the first part of the algorithm (lines 4–9), where the dominant cost is a standard $O(E \lg E)$ for the sorting of the set of edges (line 9). Noting that $|E| \leq |V|^2$ and therefore $\lg |E| \leq 2 \lg |V|$ and therefore $O(\lg E) = O(\lg V)$, we can also write the cost of Kruskal's algorithm as $O(E \lg V)$.

2.3.2 Prim's algorithm

Prim's algorithm, starting from a designated root vertex r, forces A to be a tree throughout the whole operation. The cut is the one that partitions the vertices into just two sets, those touched by A and the others. The edge to be added is the lightest one that joins a new vertex to the tree built so far.

[12] Again, see section 1.5 for more on the disjoint-set data structure, on $\alpha(n)$ and on why it is OK to treat $\alpha(n)$ as a constant in this formula.




```
0  def prim(G, r):
1      """Apply Prim's algorithm to graph G starting from root vertex r.
2      Return the result implicitly by modifying G in place: the MST is the
3      tree defined by the .predecessor fields of the vertices of G."""
4
5      Q = MinPriorityQueue()
6      for v in G.vertices():
7          v.predecessor = None
8          if v == r:
9              v.key = 0
10         else:
11             v.key = Infinity
12         Q.insert(v)
13
14     while not Q.isEmpty():
15         u = Q.extractMin()
16         for v in u.adjacent():
17             w = G.weightOfEdge(u,v)
18             if Q.hasItem(v) and w < v.key:
19                v.predecessor = u
20                Q.decreaseKey(item=v, newKey=w)
```

Each vertex is added to a priority queue, keyed by its distance to the MST-so-far. The main `while` loop on lines 14–20 is executed once for each vertex. At each step we pay for an `extractMin()` from the priority queue to extract the closest vertex `u` and add it to the MST-so-far. All its adjacent vertices are then examined and, if appropriate (that is to say: if reaching them via `u` makes them closer to the MST-so-far than they previously were), we perform `decreaseKey()` on them. How many times does this happen? Since no adjacency list is longer than $|V|$, the answer is at worst $O(V^2)$. But this may be an overestimate on sparse graphs: a tighter bound is obtained through aggregate analysis by observing that the sum of the lengths of all the adjacency lists is $2|E|$ (each edge counted twice, once by each of its endpoints); so, during a full run of the algorithm, the inner `for` loop on lines 16–20 is executed a number of times bounded by $O(E)$. This $O(E)$ is equal to $O(V^2)$ in dense graphs, but can go down to $O(V)$ in sparse graphs[13]. So the cost of Prim is

[13] Or even less for graphs that are not connected, but this is not the case here by hypothesis.

$O(V)$ times the cost of `extractMin()` plus $O(E)$ times the cost of `decreaseKey()`. With a regular binary heap implementation for the priority queue, these two operations cost $O(\lg V)$ each, yielding an overall cost for Prim of $O(V \lg V + E \lg V) = O(E \lg V)$—same as the best we achieved with Kruskal.

If the priority queue is implemented with a Fibonacci heap (see section 1.3), where the cost of `extractMin()` stays at $O(\lg V)$ but the cost of `decreaseKey()` goes down to (amortized) constant, the cost of Prim's algorithm goes down to $O(V \lg V + E)$, improving on Kruskal.

Exercise 18
Starting with fresh copies of the graph you used earlier, run these two algorithms on it by hand and see what you get. Note how, even when you reach the same end result, you may get to it via wildly different intermediate stages.

2.4 Shortest paths from a single source

Textbook

Study chapter 24 in CLRS3.

The problem is easily stated: we have a weighted directed graph; two vertices are identified and the challenge is to find the shortest route through the graph from one to the other.

An amazing fact is that, for sparse graphs, the best ways so far discovered of solving this problem may do as much work as a procedure that sets out to find distances from the source to *all* the other vertices in the entire graph. This illustrates that, if we think in terms of particular applications (for instance distances in a road atlas in this case) but then try to make general statements, our intuition on graph problems may be misleading.

While we are considering algorithms to discover the shortest paths from a designated source vertex s to any others, let us indicate as $v.\delta$ the length[14] of the shortest path from source s to vertex v, which we may not know yet, and as $v.d$ the length of the shortest path *so far discovered* from s to v. Initially, $v.d = +\infty$ for all vertices except s (for which $s.d = s.\delta = 0$); all along, $v.d \geq v.\delta$; and finally, on completion of the algorithm, $v.d = v.\delta$.

[14]Or weight—we use these terms more or less interchangeably in this section, because so does the literature. Do not get confused. In particular, we usually do not mean "number of edges" when we speak of the "length" of a path, but rather the sum of the weights of all its edges.

Exercise 19
Give a formal proof of the intuitive "triangle inequality"

$$v.\delta \leq u.\delta + w(u, v)$$

(where $w(u, v)$ is the weight of the edge from vertex u to vertex v) but covering also the case in which there is actually no path between s and v.

In some graphs it may be meaningful to have edges with negative weights. This complicates the analysis. In particular, if a negative-weight *cycle* exists on a path from s to v, then there is no finite-length shortest path between s and v—because it is always possible to find an even "shorter" path (i.e. one with a lower total weight, despite it having more edges) by running through the negative-weight cycle one more time. On the other hand, in graphs with negative-weight edges but no negative-weight cycles, shortest paths are well defined, but some algorithms may not work because their proof of correctness relies on the assumption that edge weights are never negative. For example the Dijkstra algorithm presented in section 2.4.2, although very efficient, gives incorrect results in presence of negative weigths. There exist, though, other algorithms capable of dealing with non-degenerate negative weight cases: one of them is the Bellman-Ford algorithm introduced in section 2.4.1.

An important step in most shortest-path algorithms is called **relaxation**: given two vertices u and v, each with its current "best guess" $u.d$ and $v.d$, and joined by an edge (u, v) of weight $w(u, v)$, we consider whether we would discover a shorter path to v by going through (u, v). The test is simple: if $u.d + w(u, v) < v.d$, then we can improve on the current estimate $v.d$ by going through u and (u, v). Doing so is indicated as "relaxing the edge (u, v)".

A variety of useful lemmas (microtheorems) can be proved about the properties of $.d$ and $.\delta$: one of them is the **triangle inequality** mentioned in the preceding exercise.

Another one, which we'll use later to prove the correctness of Dijkstra's algorithm, is the **convergence lemma**: if $s \leadsto u \to v$ is a shortest path from s to v, and at some time $u.d = u.\delta$, and at some time after that the edge (u, v) is relaxed, then, from then on, $v.d = v.\delta$.

Another one, which is helpful in proving the correctness of the Bellman-Ford algorithm to be examined next, is the **path relaxation lemma**: if $p = (s, v_1, v_2, \ldots, v_k)$ is a shortest path from s to v_k, and the edges of p are relaxed in the order $(s, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even if other relaxation steps elsewhere in the graph are intermixed with those on the edges of p, then after these relaxations we have that $v_k.d = v_k.\delta$.

Consult your textbook for proofs, and for more lemmas.

2.4.1 The Bellman-Ford algorithm

Bellman-Ford computes shortest paths to all vertices from a designated source s, even in the presence of negative edge weights, so long as no negative weight *cycles* are reachable from s. If any are, it reports their existence and refuses to give any other results, because in such a graph shortest paths cannot be defined.

```
0  def bellmanFord(G, s):
1      """Apply Bellman-Ford's algorithm to graph G, starting from
2      source vertex s, to find single-source shortest paths to all vertices.
3      Return a boolean indicating whether the graph is free from
4      negative cycles reachable from s.
5
6      Return the single-source-shortest-paths (only valid if the
7      above-mentioned boolean is true) by modifying G in place: for each
8      vertex v, the length of the shortest path from source to it is
9      left in v.d and the shortest paths themselves are indicated by the
10     .predecessor fields of the vertices."""
11
12     assert(s in G.vertices())
13     for v in G.vertices():
14         v.predecessor = None
15         v.d = Infinity
16     s.d = 0
17
18     repeat |V|-1 times:
19         for e in G.edges():
20             # Relax edge e.
21             if e.start.d + e.weight < e.end.d:
22                 e.end.d = e.start.d + e.weight
23                 e.end.predecessor = e.start
24
25     # If, after all this, further relaxations are possible,
26     # then we have a negative cycle somewhere
27     # and all the previous .d and .predecessor results are worthless.
28     for e in G.edges():
29         if e.start.d + e.weight < e.end.d:
30             return False
31     return True
```

Bellman-Ford works by considering each edge in turn and relaxing it if possible. This full pass on all the edges of the graph (lines 19–23) is repeated a number of times equal to the maximum possible length of a shortest path, $|V| - 1$ (lines 18–23). The algorithm therefore has a time complexity of $O(VE)$. It is not hard to show that, after that many iterations, in the absence of negative-weight cycles, no further relaxations are possible and that the distances thus discovered for each vertex are indeed the smallest possible ones.

The core of the correctness proof is based on the path relaxation lemma (page 54). For any vertex v for which a shortest path p of finite length exists from s to it, that path may contain at most $|V|$ vertices. As far as this path is concerned, the purpose of the i-th round of the outer loop in line 18 is to relax the i-th edge of the path. This guarantees that, regardless of any other relaxations that the algorithm also performs in between, it will eventually relax all the edges of p in the right order and therefore, by the lemma, it will achieve $v.d = v.\delta$.

An $O(E)$ post-processing phase (lines 28–31) then detects any negative-weight cycles. If none are found, the distances and paths discovered in the main phase are returned as valid.

2.4.2 The Dijkstra algorithm

The Dijkstra[15] algorithm only works on graphs without negative-weight edges but, on those, it is more efficient than Bellman-Ford.

Starting from the source vertex s, the algorithm maintains a set S of vertices to which shortest paths have already been discovered. The vertex $u \notin S$ with the smallest $u.d$ is then added to S (to justify this move one should prove that, at that time, $u.d = u.\delta$) and then all the edges starting from u are relaxed. This sequence of operations is repeated until all vertices are in S, in a strategy reminiscent of Prim's algorithm.

```
0  def dijkstra(G, s):
1      """Apply Dijkstra's algorithm to graph G, starting from
2      source vertex s, to find single-source shortest paths to all vertices.
3
4      Return the single-source-shortest-paths (only valid if the graph
5      has no negative-weight edges) by modifying G in place: for each
6      vertex v, the length of the shortest path from source to it is
7      left in v.d and the shortest paths themselves are indicated by the
8      .predecessor fields of the vertices."""
9
10     assert(s in G.vertices())
11     assert(no edges of G have negative weight)
12     Q = MinPriorityQueue() # using .d as each item's key
13     for v in G.vertices():
14         v.predecessor = None
15         if v == s:
16             v.d = 0
17         else:
18             v.d = Infinity
19         Q.insert(v)
20
21     # S (initially empty) is the set of vertices of G no longer in Q.
22     while not Q.isEmpty():
23         u = Q.extractMin()
24         assert(u.d == u.delta) # NB we can't actually _compute_ u.delta.
25         for v in u.adjacent():
26             # Relax edge (u,v).
27             if Q.hasItem(v) and u.d + G.weightOfEdge(u,v) < v.d:
28                 v.predecessor = u
29                 Q.decreaseKey(item = v, newKey = u.d + G.weightOfEdge(u,v))
```

As can be seen, if any vertex v is unreachable from s, its $v.d$ will stay at $+\infty$ throughout and it will be processed, as will any others in that situation, in the final round(s) of the `while` loop of lines 22–29; but the algorithm will still terminate without problems. Conversely, any vertex v whose $v.d$ is still $+\infty$ after completion is indeed unreachable from s.

[15] It's a Dutch name: pronounce as [ˈdɛɪkstra].

Computational complexity

Performance-wise, to speed up the extraction of the vertex with the smallest $u.d$, we keep the unused vertices in a min-priority queue Q where they are keyed by their $.d$ attribute. Then, for each vertex, we must pay for one `extractMin()` plus as many `decreaseKey()` as there are vertices in the adjacency list of the original vertex. We don't know the length of each individual adjacency list, but we know that all of them added together have $|E|$ edges (since it's a directed graph). So in aggregate we pay for $|V|$ times the cost of `extractMin()` and $|E|$ times the cost of `decreaseKey()`. With a regular binary heap implementation for the priority queue, these two operations cost $O(\lg V)$ each, yielding an overall cost for Dijkstra of $O(V \lg V + E \lg V) = O(E \lg V)$. Same as Prim.

As with Prim, though, we can however do better if we make the priority queue faster: you will recall that the development of Fibonacci heaps (section 1.3) was originally motivated by the desire to speed up the Dijkstra algorithm. Using a Fibonacci heap to hold the remaining vertices, the `decreaseKey()` cost is reduced to amortized $O(1)$, so the overall cost of the algorithm goes down to $O(V \lg V + E)$.

Correctness proof

To prove the correctness of Dijkstra's algorithm the crucial point, as already noted, is to prove that, for the vertex u that we extract from the queue in line 23, it is indeed the case that the assertion on line 24, i.e. $u.d = u.\delta$, holds. We prove this by contradiction.

Since $u.d$ can never be $< u.\delta$ by definition of $.d$ and $.\delta$, the only two cases left are "$u.d = u.\delta$" and "$u.d > u.\delta$". Let's imagine for the sake of argument that there is a vertex u that, when extracted from Q in line 23, has

$$u.d > u.\delta. \tag{2.1}$$

Consider the *first* such vertex we encounter while running the algorithm and consider the shortest path[16] from s to u.

Represent this shortest path as $s \rightsquigarrow x \to y \rightsquigarrow u$, where (x, y) is the first edge along this path for which $x \in S$ and $y \notin S$. At the time (line 23) of adding u to S, since u was the chosen vertex and thus the one with the smallest $.d$ among all the vertices outside S,

$$u.d \leq y.d. \tag{2.2}$$

[16]Detour: we should also prove that a shortest path exists, for which it is sufficient to prove that u is reachable—and it is, otherwise $u.d = u.\delta = +\infty$, contradicting hypothesis (2.1)

Since the path $s \rightsquigarrow x \to y$ matches all the conditions of the convergence lemma (the path is a subpath of a shortest path and hence a shortest path itself; $x.d = x.\delta$ because $x \in S$ and u, found after x, is the first vertex to violate that property; and (x, y) was relaxed when x was added to S), we have that, at the time of adding u to S,

$$y.d = y.\delta. \tag{2.3}$$

Combining equations (2.1), (2.2) and (2.3) we have $u.\delta < u.d \leq y.d = y.\delta$, which implies

$$u.\delta < y.\delta. \tag{2.4}$$

However, since $s \rightsquigarrow x \to y \rightsquigarrow u$ is a shortest path by hypothesis, on the assumption that all edges have non-negative weights then reaching u via y must cost at least as much as reaching y and therefore $u.\delta \geq y.\delta$. This directly contradicts (2.4), proving that there cannot be any vertex u satisfying (2.1) when extracted from Q at line 23, QED.

2.5 Shortest paths between any two vertices

Textbook

Study chapter 25 in CLRS3.

It is of course possible to compute all-pairs shortest paths simply by running the single-source shortest path algorithm on each vertex. If the graph contains negative edges, running Bellman-Ford on all vertices will cost $O(V^2E)$, which is $O(V^4)$ for dense graphs and $O(V^3)$ for sparse graphs. For a graph without negative edges, using Dijkstra and Fibonacci heaps we can compute all-pairs shortest paths in $O(V^2 \lg V + VE)$. Interestingly, there are ways in which we can do better.

Note that, in this section, we compute the weight of the shortest path between two vertices without worrying about keeping track of the edges that form that path. You might consider addressing that issue but I am not making this a boxed exercise as it may take you a while. The most interesting way to find out about it is clearly to write an actual program.

2.5.1 All-pairs shortest paths via matrix multiplication

Let's first look at a method inspired by matrix multiplication, which works even in the presence of negative-weight edges, though necessarily we require the absence of negative-weight *cycles*.

Represent the graph with an adjacency matrix W whose elements are defined as:

$$w_{i,j} = \begin{cases} \text{the weight of edge } (i,j) & \text{for an ordinary edge;} \\ 0 & \text{if } i = j; \\ \infty & \text{if there is no edge from } i \text{ to } j. \end{cases}$$

We want to obtain a matrix of shortest paths L in which $l_{i,j}$ is the weight of the shortest path from i to j (∞ if there is no path).

Let $L^{(m)}$ be the matrix of shortest paths that contain no more than m edges. Then $W = L^{(1)}$.

Exercise 20
Write out explicitly the elements of matrix $L^{(0)}$.

Let's build $L^{(2)}$:

$$l_{i,j}^{(2)} = \min\left(l_{i,j}^{(1)}, \quad \min_{k=1,n}(l_{i,k}^{(1)} + w_{k,j}) \right).$$

In other words, the shortest path consisting of at most two steps from i to j is either the shortest one-step path or the shortest two-step path, the latter obtained by fixing the endpoints at i and j and trying all the possible choices for the intermediate vertex k. Similarly, we build $L^{(3)}$ from $L^{(2)}$ by adding a further step: it's either the shortest path of at most two steps, or the shortest path of at most three steps; the latter obtained by trying all possible combinations for an additional last edge, with the first two steps given by a shortest two-step path.

$$l_{i,j}^{(3)} = \min\left(l_{i,j}^{(2)}, \quad \min_{k=1,n}(l_{i,k}^{(2)} + w_{k,j}) \right).$$

We proceed using the same strategy for $L^{(4)}$, $L^{(5)}$ and so on.

We can also simplify the formula by noting that $w_{j,j} = 0 \quad \forall j$ and that therefore the term with $k = j$ reduces from $(l_{i,k}^{(2)} + w_{k,j})$ to $(l_{i,j}^{(2)} + w_{j,j}) = (l_{i,j}^{(2)} + 0)$ and is thus equal to the first $l_{i,j}^{(2)}$ inside the big brackets; thus we don't need to have two "rounds" of minimum to include it and we can simply write, in the general case:

$$l_{i,j}^{(m+1)} = \min\left(l_{i,j}^{(m)}, \quad \min_{k=1,n}(l_{i,k}^{(m)} + w_{k,j}) \right) = \min_{k=1,n}(l_{i,k}^{(m)} + w_{k,j}).$$

It is easy to prove that this sequence of matrices $L^{(m)}$ converges to L after a finite number of steps, namely $n-1$. Any path with more than $n-1$ steps must revisit a vertex and therefore include a cycle. Under the assumption that there are no negative-weight cycles, the path cannot be shorter than the one with fewer edges obtained from it by removing the cycle. Therefore

$$L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots = L.$$

The operation through which we compute $L^{(m+1)}$ from $L^{(m)}$ and W is reminiscent of the matrix multiplication

$$P = L^{(m)} \cdot W.$$

Compare

$$l_{i,j}^{(m+1)} = \min_{k=1,n} (l_{i,k}^{(m)} + w_{k,j})$$

and

$$p_{i,j} = \sum_{k=1,n} l_{i,k}^{(m)} \cdot w_{k,j}$$

to discover the mapping:

$$\begin{array}{rcl} \min & \leftrightarrow & + \\ + & \leftrightarrow & \cdot \\ \infty & \leftrightarrow & 0 \\ 0 & \leftrightarrow & 1 \end{array}$$

> **Exercise 21**
> To what matrix would $L^{(0)}$ map? What is the role of that matrix in the corresponding algebraic structure?

It is possible to prove that this mapping is self-consistent. Let's therefore use a "matrix product" notation and say that $L^{(m+1)} = L^{(m)} \cdot W$. One matrix "product" takes $O(n^3)$ and therefore one may compute the all-pairs shortest path matrix $L^{(n-1)}$ with $O(n^4)$ operations. Note however that we are not actually interested in the intermediate $L^{(m)}$ matrices! If n is, say, 738, then instead of iterating $L^{(m+1)} = L^{(m)} \cdot W$ to compute

$$L^{(1)}, L^{(2)}, L^{(3)}, L^{(4)}, \ldots, L^{(736)}, L^{(737)}$$

we can just repeatedly square W and obtain the sequence

$$L^{(1)}, L^{(2)}, L^{(4)}, L^{(8)}, \ldots, L^{(512)}, L^{(1024)}.$$

Since all $L^{(m)}$ are equal to L for $m = n-1$ onwards, we have that $L^{(1024)} = L^{(737)} = L$ and we don't have to hit the target exactly—it's OK to keep squaring until we exceed it. Since the number of steps in the second sequence is only $\lceil \lg n \rceil$ instead of $n-1$, this method brings the cost down to $O(n^3 \lg n)$—also known as $O(V^3 \lg V)$ since n, the order of the matrix, is the number of vertices in the graph.

This repeated matrix squaring algorithm for solving the all-pairs shortest paths problem with negative edges is easy to understand and implement. At $O(V^3 \lg V)$ it beats the iterated Bellman-Ford whenever $O(E) > O(V \lg V)$, although it loses on very sparse graphs where the reverse inequality holds. Another option for the case with negative edges, not discussed in these notes, is the Floyd-Warshall algorithm, which is $O(V^3)$ (see textbook). Even better, however, is Johnson's algorithm, discussed next.

2.5.2 Johnson's algorithm for all-pairs shortest paths

The Johnson algorithm, like the iterated matrix squaring, the iterated Bellman-Ford and the Floyd-Warshall algorithms, runs on graphs with negative edges but without negative cycles. The clever idea behind Johnson's algorithm is to turn the original graph into a new graph that only has non-negative edges *but the same shortest paths*, and then to run Dijkstra on every vertex.

The Johnson algorithm includes one pass of Bellman-Ford at a cost of $O(VE)$; but since the final step of iterating Dijkstra on all vertices costs, as we said, at least $O(V^2 \lg V + VE)$ even in the most favourable case of using Fibonacci heaps, we are not increasing the overall complexity with this single pass of Bellman-Ford. Note that $O(VE)$ is $O(V^3)$ for dense graphs but it may go down towards $O(V^2)$ for sparse graphs; in the latter case, therefore, Johnson's algorithm reduces to $O(V^2 \lg V)$, offering a definite improvement in asymptotic complexity over Floyd-Warshall's $O(V^3)$.

The most interesting part of the Johnson algorithm is the **reweighting** phase in which we derive a new graph without negative edges but with the same shortest paths as the original. This is done by assigning a new weight to each edge; however, as should be readily apparent, the simple-minded strategy of adding the same large constant to the weight of each edge would not work, as that would change the shortest paths[17].

We introduce a fake source vertex s, and fake edges (s, v) of zero weight from that source to each vertex $v \in V$. Then we run Bellman-Ford on this augmented graph, computing distances from s for each vertex. As a side effect, this tells us whether the original graph contains any negative weight cycles[18]. If there are no negative cycles, we proceed. If we indicate as $v.\delta$ the shortest distance from s to v as computed by Bellman-Ford (and left in $v.d$ at the end of the procedure), we assign to each edge (u, v) of weight $w(u, v)$ a new weight $\hat{w}(u, v) = u.\delta + w(u, v) - v.\delta$. Adding the above equality to the triangle inequality $u.\delta + w(u, v) \geq v.\delta$, we get

$$\hat{w}(u, v) + u.\delta + w(u, v) \geq u.\delta + w(u, v) - v.\delta + v.\delta$$

and, simplifying,

$$\hat{w}(u, v) \geq 0,$$

meaning that the reweighted edges are indeed all non-negative, as intended.

Let's now show that this reweighting strategy also preserves the shortest paths of the original graph. Let $p = v_0 \to v_1 \to \ldots \to v_k$ be any path in the original graph, and let $w(p) = w(v_0, v_1) + w(v_1, v_2) + \ldots + w(v_{k-1}, v_k)$ be its weight. The weight of p in the reweighted graph is $\hat{w}(p) = w(p) + v_0.\delta - v_k.\delta$, since all the $v.\delta$ of the intermediate vertices cancel out. Therefore, once we fix the start and end vertices v_0 and v_k, whatever path we choose from one to the other (including ones that do not visit those intermediate vertices v_1, v_2 etc, and ones that visit other vertices altogether), the weight of the path in the old and in the reweighted graph differ only by a constant (namely $v_0.\delta - v_k.\delta$), which implies that the path from v_0 to v_k that was shortest in the original graph is still the shortest in the new graph, QED.

[17] Just imagine adding such a large constant that the original weights become irrelevant.

[18] It does if and only if the augmented graph does, since the vertex and edges we introduced do not add or remove any cycles to the original graph.

So, after reweighting, we can run Dijkstra from each vertex on the reweighted graph, at the stated costs, and obtain shortest paths. The actual lengths of these shortest paths in the original graph are recovered in constant cost for each path: $w(p) = \hat{w}(p) - v_0.\delta + v_k.\delta$. The total cost of this final phase is therefore proportional to the number of shortest paths, $O(V^2)$, and it does not affect the overall asymptotic cost.

2.6 Maximum flow

Textbook

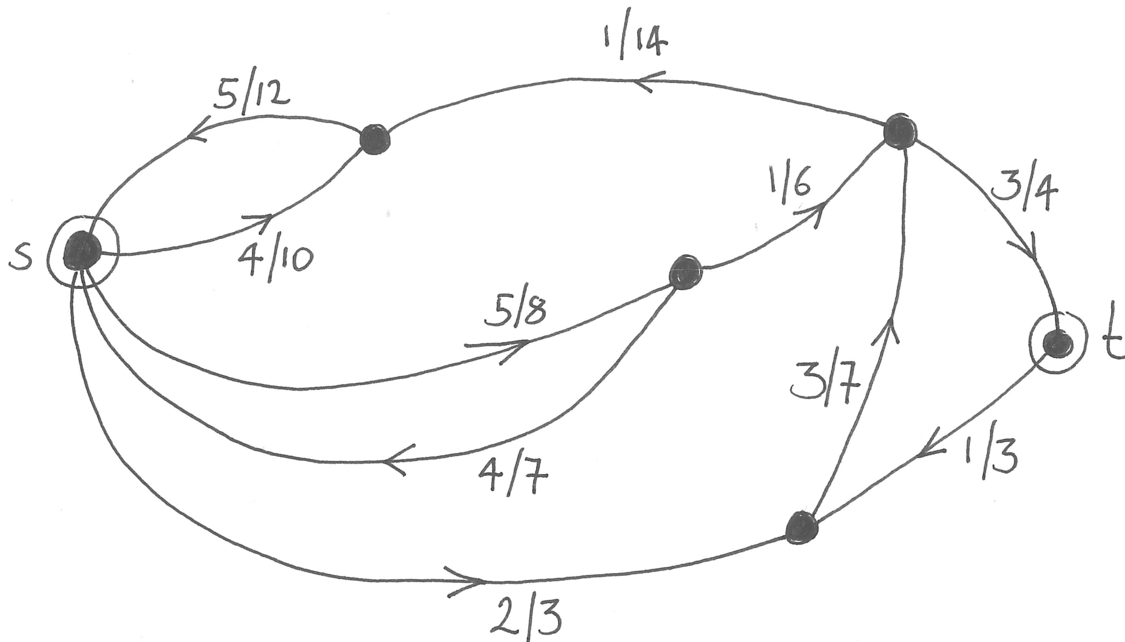
Study chapter 26 in CLRS3.

If the edge weights are taken to represent capacities (how many cubic metres per second, or cars per hour, or milliamperes, can flow through this link?) an interesting question is to determine the maximum flow that can be established between the two designated source and sink vertices[19]. Whatever the entity that is flowing, be it vodka, IP packets or lemmings, we assume in this model that it can't accumulate in the intermediate vertices[20], that the source can generate an infinite supply of it, and that the sink can absorb an infinite amout of it, with the only constraints to the flow being the capacities of the edges. Note that the actual flow through an edge at any given instant is quite distinct from the maximum flow that that edge can support. The two values are usually separated by a slash[21]: for example an edge labelled 3/16 is currently carrying a flow of 3 and has a maximum capacity of 16.

[19] The multiple-sources and multiple-sinks problem is trivially reduced to the single-source and single-sink setting: add a dummy super-source linked to every actual source by edges of infinite capacity and proceed similarly at the sink side with a dummy super-sink.

[20] Therefore the total flow into an intermediate vertex is equal to the total flow out of it.

[21] Not to be taken as meaning division but rather "out of".



In order to deal with flow problems, we introduce a few useful concepts and definitions.

A **flow network** is a directed graph $G = (V, E)$, with two vertices designated respectively as source and sink, and with a **capacity** function $c : V \times V \to \mathbb{R}^+$ that associates a non-negative real value to each pair of vertices u and v. The capacity $c(u, v)$ is 0 if and only if there is no (u, v) edge; otherwise it is noted as the number after the slash on the label of the (u, v) edge, representing the maximum possible amount that can flow directly from u to v through that edge. A **flow** is a function $f : V \times V \to \mathbb{R}$ that associates a (possibly negative) real value to each pair of vertices, subject to the following constraints:

- $f(u, v) \leq c(u, v)$;
- $f(u, v) = -f(v, u)$;
- and "flow in = flow out" for every vertex except source and sink.

Note that the flow $f(x, y)$ can be nonzero even if there is no edge from x to y: for that it is enough to have an edge from y to x carrying the opposite amount. Conversely, when an (x, y) edge and a (y, x) edge both exist and each carry some amount, say a and b respectively, then the flow between x and y is meant as the "net" flow: $f(x, y) = a - b$ and $f(y, x) = b - a$. This is consistent with the constraints presented above.

The **value** of a given flow f is a scalar value $|f| \in \mathbb{R}$ equal to the total flow out of the source: $|f| = \sum_{v \in V} f(s, v)$. Owing to the properties above, it is also equal to the total flow into the sink.

The **residual capacity** $c_f : V \times V \to \mathbb{R}^+$ is a function that, given a flow network G and a flow f, indicates for each pair of vertices the extra amount of flow that can be pushed between the vertices of the pair through the edge(s) that directly join them, if any, on top of what is currently flowing, without exceeding the capacity constraint of the 0, 1 or 2 edges that directly join the two vertices. So, for any two vertices u and v, we

have that $c_f(u,v) = c(u,v) - f(u,v)$. Note that we do not require the original graph to have an (u,v) edge in order for $c_f(u,v)$ to be greater than zero: if the graph has a (v,u) edge that is currently carrying, say, 4/16, then one can, with respect to that situation, increase the flow from u to v by up to 4 units by *cancelling* some of the flow that is already being carried in the opposite direction—meaning that there is, in that situation, a positive residual capacity from u to v of $c_f(u,v) = 4$ (the latter not to be confused with $c(u,v)$, which instead stays at zero throughout in the absence of a (u,v) edge). Note also that the residual capacity $c_f(u,v)$ can exceed the capacity $c(u,v)$ if the current net flow from u to v is negative: first, by increasing the flow in the direction from u to v, you reduce the absolute value of the negative flow, eventually bringing $f(u,v)$ to zero; then you may keep on increasing the flow until you reach $c(u,v)$.

Exercise 22
Draw suitable pictures to illustrate and explain the previous comments. *Very easy; but failure to do this, or merely seeing it done by someone else, will make it unnecessarily hard to understand what follows.*

The **residual network** $G_f = (V, E_f)$ is the graph obtained by taking all the edges with a strictly positive residual capacity c_f and labelling them with that capacity. It is itself a flow network, because the residual capacity is also non-negative.

An **augmenting path** for the original network is a sequence of edges from source to sink in the residual network. The residual capacity of that path is the maximum amount we can push through it, equal of course to the residual capacity of its lightest edge.

We are now ready to describe how to find the maximum flow—or, making use of the above definitions for greater accuracy, to describe how to find, among all the possible flows on the given flow network G, the one whose value $|f|$ is the highest.

2.6.1 The Ford-Fulkerson maximum flow method

We call Ford-Fulkerson a “method” rather than an “algorithm” because it is in fact an algorithm blueprint whose several possible instantiations may have different properties and different asymptotic complexity. The general pattern is simply to compute the residual network, find an augmenting path on it, push the residual capacity through that path (thereby increasing the original flow) and repeat while possible. The algorithm terminates when there is no augmenting path in the residual network.

```
0 def fordFulkerson(G):
1     initialize flow to 0 on all edges
2     while an augmenting path can be found:
3         push as much extra flow as possible through it
```

Contrary to most other graph algorithms examined in this course, the running time does not just depend on the number of vertices and edges of the input graph but on the *values* of its edge labels.

At each pass through the loop, the value of the flow increases. Since all edge capacities are finite, the maximum flow is finite. The method might fail to terminate if it took an

infinite number of steps for the flow to converge to its maximum value. For this to happen, the increments must become infinitesimal. Since the increment in line 3 is the greatest possible given the chosen augmenting path, its value is obtained as a difference of previously computed capacities (since we start from a null flow in line 1). Therefore, if all capacities are integers, every increment will also be an integer and the flow will increase by at least 1 at each pass through the loop, thereby bringing the algorithm to completion in a finite number of steps.

So, integer capacities for all edges are a sufficient condition for termination. Conversely, it is indeed possible[22] to construct elaborate graphs with irrational capacities where Ford-Fulkerson does not terminate.

As for running time in the case of integer capacities, line 3 changes the flow of each edge along the augmenting path, so its running time is proportional to the length of the path and is therefore bounded by the length in edges of the longest path without cycles, $|V| - 1$. On the other hand, finding such a path, if it exists, for example with Breadth First Search or Depth First Search, will cost up to $O(V + E)$. Each iteration thus costs $O(E)$. In the worst case, each pass increases the flow only by 1, so the loop may be executed up to $|f^*|$ times (the value of the maximum flow). So, in total, the running time is bounded by $O(E \cdot |f^*|)$. Note how this growth rate is independent and therefore potentially higher than any polynomial function of $|V|$ and $|E|$, since the capacity values on the edge labels are independent of the shape or size of the graph.

With a maximum flow of high numerical value and a consistently pessimal choice of augmenting path, Ford-Fulkerson might take billions of operations to compute a solution even on a tiny graph with only 5 edges. Specific implementations of Ford-Fulkerson, however, can improve on this sad situation by defining some useful criterion for selecting an augmenting path in line 2, as opposed to leaving the choice entirely open. The so-called **Edmonds-Karp** algorithm, for example, derived from Ford-Fulkerson by choosing as augmenting path through a breadth-first search, thereby picking one with the smallest number of edges, can be proved to converge in $O(VE^2)$ time.

Another possibility, also suggested by Edmonds and Karp, is to select the augmenting path based on the greatest residual capacity. There are still others.

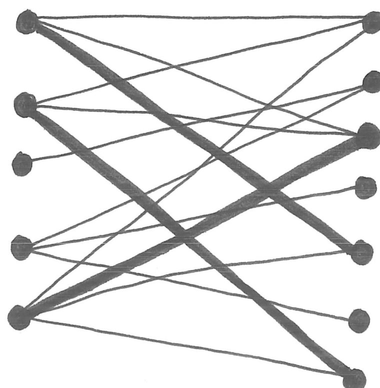
2.6.2 Bipartite graphs and matchings

A **matching** in a bipartite graph[23] is a collection of edges such that each vertex of the graph is the endpoint of at most one of the selected edges. A maximal matching is then obviously as large a subset of the edges that has this property as is possible. Why might one want to find a matching? Well, bipartite graphs and matchings can be used to represent many resource allocation problems. For example, vertices on the left might be final-year students and vertices on the right might be final year project offers. An edge would represent the interest of a student in a particular project. The maximum cardinality matching would be the one satisfying the greatest number of student requests,

[22] Possible, but far from trivial or intuitive: Ford and Fulkerson's own original example had 10 vertices and 48 edges. The minimal example appeared years later in Uri Zwick, "The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate", *Theoretical Computer Science* 148, 165-170 (1995).

[23] As you may recall, the definition of *bipartite graph* was given in section 2.1.1 on page 42.

assuming that no two students can choose the same project and that no student can do more than one project.



A simple direct search through all possible combinations of edges would provide a direct way of finding maximal matchings, but would have costs growing exponentially with the number of edges in the graph. This, even for small graphs, is not a feasible strategy.

A particularly smart way of solving the bipartite matching problem is to recast it as a maximum flow problem in the following way. Introduce a fake super-source to the left of the bipartite graph and a fake super-sink to the right. Add edges of unit capacity from the super-source to all the left side vertices and from all the right side vertices to the super-sink. This is very similar to the construction mentioned in footnote 19 (section 2.6, page 62) to transform a multiple-sources and multiple-sinks flow problem into one with a single source and a single sink. Here, however, we give all edges (including the ones touching the super-source and super-sink) unit capacity. If we constrain all flows to be integers, assigning unit capacities to the edges ensures that no two edges will start or end on the same internal vertex and therefore that all flows will also be valid matchings. All that's left is then to find the maximum flow (itself a valid matching, and necessarily the one with the greatest number of edges) using the Ford-Fulkerson method.

A more complicated variant, which we won't discuss, is that of the *weighted* matching problem: this is where a bipartite graph has the edges labelled with values and it is necessary to find the matching that maximizes the sum of weights on the edges selected.

Chapter 3

Parallel algorithms

Chapter contents

Dynamic multithreading. Work and span. Greedy scheduler. Determinacy races.
Expected coverage: about 1 lecture.

Textbook

Study chapter 27 in CLRS3.

Moore's law, about transistor count in state-of-the-art ICs doubling every two years, surprisingly continues to hold after decades; but clock speeds no longer increase at the same rate. To what extent can the abundance of silicon gates be used to compensate for lack of progress in processor speed? While it's obvious that a processor with 4× the clock frequency can be expected to execute a CPU-bound task four times faster, what can be said of a processor with 4× the number of cores?

Imagine we have to perform a CPU-bound job that requires 1 trillion (10^{12}) machine instructions. A processor that executes 1 billion instructions per second will take 1000 seconds (about 17 minutes) to complete it. If a 10× faster processor existed[1], it would only take 100 seconds, or just over one and a half minute. What if instead we had 10 processors (or 10 cores), each with the same power as the original? For the whole 1-trillion-instruction job to take only 100 seconds, each of the 10 processors would have to run solidly for the whole 100 seconds, in order to execute its quota of 100 billion instructions. If any of the 10 processors were idle for part of that time, 100 seconds would not be sufficient to execute an aggregate 1 trillion instructions[2]. In order to keep each processor busy we need to:

- partition the work equally among all the available processors; and

[1] But maybe it doesn't exist—and, even if it did, it would probably cost too much.
[2] We ignore overheads for scheduling etc.

- ensure that no processor is ever tied up waiting for a result that another processor still has to produce.

Whether we are able to achieve this or not is a function of both the problem itself and of the algorithm we write to solve it. Only in the most favourable cases will we be able to achieve a speedup equal to the number of processors; in general, that will only be an upper bound. We may not be able to achieve the bound if:

- we cannot split the problem into as many independent pieces as there are processors; if we only manage to split the problem into 4 pieces, and there are 10 processors, at least 6 of them will be idle at any time;

- we can split the problem into as many pieces as there are processors, but the pieces are of uneven size; in that case some processors will finish before others and will have to sit there twiddling their thumbs;

- we can split the work into many pieces, but those pieces depend on each other's results; so a processor, despite having an allocated piece of problem it could work on, must wait idly until another processor finishes its piece and supplies a result that is needed as input.

Any time one of the 10 processors is idle, for any of the above reasons, it does not contribute towards the quota of 1 trillion instructions that need to be executed to finish the job. Clearly, then, it will take more than 100 seconds to get to completion, and the speedup factor gained by having $10\times$ more processors will be less than $10\times$.

Taking an example from cryptography, brute-force key-search is easy to parallelize: you can partition the key space into arbitrarily many regions and give each region to a single processor. Trying the keys in one region is completely independent from trying the keys in another region, so each processor can work fully independently of the others. So long as the regions are of equal size, the speedup factor to search the whole key space[3] is equal to the number of processors assigned to the job.

Taking another example from cryptography that goes in the opposite direction, sometimes we build on purpose a function that is hard to parallelize. If, as you should, you salt and hash your passwords in the back-end, you want the function that goes from password to hash to take a long time to execute, so that attackers who learn the hash and salt still won't be able to find the original password by brute force. To do that, you might actually compute the hash of the hash of the hash ... a million times. And you like the fact that there is no easy way to parallelize the computation of $h^{1,000,000}(salt, password)$, in which each of the million individual hash computations in the chain requires the result of the previous one.

So what do we do in this chapter? Well, we will barely have time to scratch the surface, but our aim is to introduce some intellectual tools to reason about parallel algorithms and evaluate their performance in a quantitative way.

[3]Of course, as soon as the sought key is found, the whole process is terminated early—and there's no telling whether the time to actually find the key is decreased by the same factor. But it all makes sense again when talking of *worst case* execution times (search whole keyspace to find that the key was the very last one to be tried).

3.1 Programming model: dynamic multithreading

There are various possible architectures for parallel computing: we won't explore them all. We shall model the one that is common in the mainstream microprocessors that are popular at the time of writing, where several processor cores have access to the same memory (a *shared memory* architecture, as opposed to the *distributed memory* case in which each processor has its own local memory not accessible to the others).

As for the programming model, with *static threading* there is a fixed pool of threads, each representing a (possibly virtual) processor, all having access to a common memory. The programmer is responsible for partitioning the work among the available threads, and doing so in a balanced way is difficult and error-prone. A middleware layer, the *concurrency platform*, can take care of this task and give the programmer the illusion of having as many processors as necessary. Behind the scenes, the concurrency platform allocates jobs to threads in a way that balances the load. This is *dynamic multithreading*, the model we are going to discuss.

In the formulation of the CLRS3 textbook, we describe parallel algorithms by adding three special concurrency keywords to the pseudocode: `spawn`, `sync`, `parallel`, with the following meanings.

`spawn`: An optional prefix to a procedure call statement. Call the indicated procedure, but in a separate thread. In the current thread, just continue. (If you were assigning the result of the procedure call to a variable, as in `y = spawn f(x)`, then the variable will retain the old value until the spawned procedure returns.)

`sync`: Block execution until all the threads previously spawned from within the current procedure[4] have completed.

`parallel`: An optional prefix to the standard looping keyword `for` that causes each iteration of the loop to take place in its own thread.

Note that the `spawn` and `parallel` keywords *allow* the concurrency platform to execute pieces of code in parallel, but they do not require it. Programs with those keywords can generate arbitrarily many threads and it is up to the scheduler to assign them to actual processors.

Removing all occurrences of these three keywords from the pseudocode yields the serial version of the algorithm.

A *strand* is a sequence of statements that does not contain concurrency keywords. We can visualize the execution of the algorithm as a graph (a DAG) in which each strand is a vertex[5] and the edges between vertices represent procedure calls, returns from calls, procedure spawns, return from spawns, as well as the act of continuing sequentially within the same procedure instance.

[4]Including threads spawned by spawned children down to arbitrary depths.

[5]In case of recursive calls, a given strand in the source code may appear in the graph as several vertices, one per instance of the procedure.

3.2 Performance analysis

3.2.1 Definitions

$P =$ number of processors.

$T_P =$ running time of a computation on P processors.

$T_1 =$ **work** of a computation: time to execute that computation on a single processor (hence T_P with $P = 1$).

$T_\infty =$ **span** of a computation: execution time of the critical path in the DAG[6]. Indicated as the time for $P = \infty$ because, if we had infinite processors available, the time to execute the computation would be determined by the critical path.

From these definitions come two seemingly obvious (but actually deceptively subtle) laws that give lower bounds on the time it takes to execute the computation on P processors.

> **Work law:** $T_P \geq \frac{T_1}{P}$
>
> The running time on P processors can't be any shorter than if all of them work all the time, with no wasted cycles.

> **Span law:** $T_P \geq T_\infty$
>
> The running time on P processors can't be any shorter than the time it would take on arbitrarily many processors.

Further definitions:

- The **speedup** of a computation, when run on P processors, is the multiplying factor that says how much faster the computation goes on P processors compared to when it runs on just one processor: speedup $= T_1/T_P$.
- We call it **linear speedup** iff the speedup is proportional to the number of processors used, i.e. iff $\frac{T_1}{T_P} = \Theta(P)$.
- We call it **perfect linear speedup** iff the speedup is exactly equal to the number of processors used, i.e. iff $\frac{T_1}{T_P} = P$.
- The **parallelism** of a computation is defined as the work divided by the span: T_1/T_∞. Given the definition of speedup, we can intuitively interpret the parallelism of a computation as the maximum speedup factor one can possibly obtain by providing arbitrarily many processors.

  We can prove that the parallelism is also:

[6] Equivalent to the maximum among the execution times of all the paths in the DAG.

  - The average amount of work (in terms of number of active processors) that can be performed in parallel at each step of the critical path.
  - A limit on the number of processors that you can add before losing the ability to achieve perfect linear speedup.

- The **parallel slackness** of a computation, when run on P processors, is the factor by which the parallelism of the computation exceeds the parallelism of the machine (i.e. the number of available processors). By definition it's therefore $\frac{T_1/T_\infty}{P} = \frac{T_1}{P \cdot T_\infty}$. A high parallel slackness is good for achieving high speedup factors, i.e. "not wasting processor time".

In the next section we shall see that, with high parallel slackness (i.e. if you give it some margin, meaning a computation that is highly parallelizable), a good scheduler can achieve nearly perfect linear speedup.

Exercise 23
Prove that T_1/T_∞ is the average amount of work that can be performed in parallel at each step of the critical path.

Exercise 24
Prove that, if $P > T_1/T_\infty$, the computation cannot achieve perfect linear speedup.

3.2.2 Scheduling

In this programming model, the programmer doesn't say which processor executes what strand: the scheduler decides. The scheduler sees requests for parallelism as they come but doesn't know in advance what procedure is about to be spawned or when a spawned procedure will return. What policy should the scheduler follow? How do we know if it's any good?

A simple but effective strategy is that of the *greedy scheduler*: in each step, assign as many strands to processors as possible. We are going to prove that it's reasonably good.

First, assume that each strand takes unit time[7]. At each time unit there will be zero or more strands ready to be executed and zero or more strands blocked in a `sync` operation. We call each step **complete** if the number of strands ready to be executed is at least P; **incomplete** otherwise.

[7] If not, you could always chop up longer strands into chains of unit-time strands.

Theorem

With a greedy scheduler, the time T_P to run a computation on P processors is bounded by

$$T_P \quad \leq \quad \frac{T_1}{P} + T_\infty.$$

Proof: Each step is either complete or incomplete. The total number of complete steps (in which all processors are assigned a work unit) cannot exceed the total number of work units divided by the number of processors[8]:

$$\text{complete steps} \quad \leq \quad T_1/P.$$

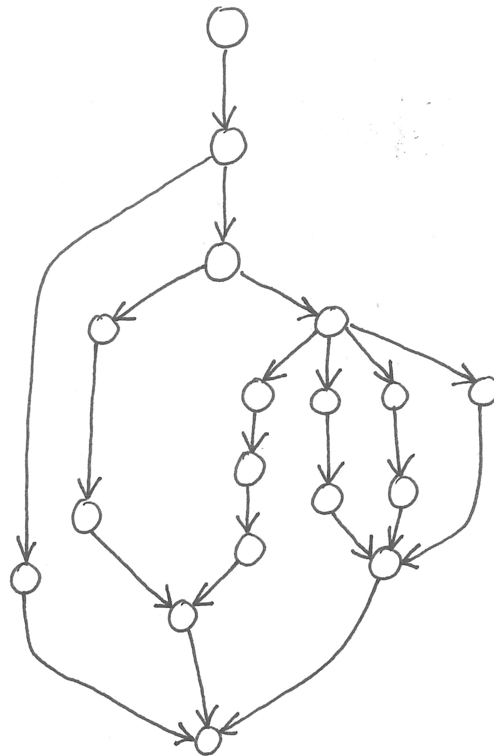
On the other hand, consider an incomplete step. There are fewer than P strands ready to execute. Each strand ready to execute is a source vertex in the execution DAG[9]. Take the longest path from a source to a sink in the DAG: after execution of the incomplete step, that path will be shorter by at least one, because the source vertex will have been executed (and the next vertex will be a source at that point). Therefore each incomplete step shortens the longest path from source to sink by one. Since the longest path is the critical path, the number of incomplete steps is bounded by the span:

$$\text{incomplete steps} \quad \leq \quad T_\infty.$$

Each step is either complete or incomplete, hence by adding the two inequalities we obtain the thesis.

[8]This is a kind of dual of the work law.

[9]Because, if it had any incoming arrows, it would be waiting for some other strand to complete and it thus wouldn't be ready to execute.



Theorem

The performance of a greedy scheduler is within 2× of optimal.

Proof: given the bounds imposed by the work law and the span law, even the optimal scheduler cannot take less time than $T_P^* = \max(\frac{T_1}{P}, T_\infty)$. From the previous theorem,

$$T_P \quad \leq \quad \frac{T_1}{P} + T_\infty \quad \leq \quad 2\max(\frac{T_1}{P}, T_\infty) \quad = \quad 2T_P^*$$

QED.

Theorem

If you use a greedy scheduler and have sufficiently high parallel slackness, you get almost perfect linear speedup.

Proof: High parallel slackness means

$$\frac{T_1}{P \cdot T_\infty} \gg 1 \quad \Rightarrow \quad \frac{T_1}{P} \gg T_\infty \quad \Rightarrow \quad \frac{T_1}{P} + T_\infty \approx \frac{T_1}{P}.$$

Plugging this approximation into the first theorem above, and combining with the work law,

$$T_P \leq \frac{T_1}{P} + T_\infty \approx \frac{T_1}{P} \leq T_P \quad \Rightarrow \quad \frac{T_1}{P} \approx T_P \quad \Rightarrow \quad \frac{T_1}{T_P} \approx P$$

QED.

3.2.3 Race conditions

A parallel algorithm is **deterministic** if its results on a given input do not depend on the order in which instructions are scheduled; it is **nondeterministic** if the results on the same input might change between executions (by virtue of the scheduler selecting different combinations of ready threads in different runs).

For everyone's sanity, especially yours, you are strongly advised not to write non-deterministic programs. People do, but usually not on purpose. The main cause of nondeterminism is a *race condition*—a fertile source of spectacular bugs that are very hard to reproduce and hence debug.

A **determinacy race** occurs when two or more threads access the same memory location in parallel and at least one performs a write.

Imagine two parallel threads accessing variable x, initialized to 0. Each thread consists of the code `x = x + 1`. What is going to be the value in x after executing and syncing the two threads? It might be reasonable to expect that the answer will be 2 regardless of the order in which the threads execute.

However, by our definition above, this situation contains a determinacy race. Although the problem is not obvious when looking at the high level instructions, when each assignment is expanded into its machine code components, and when we consider that the machine code instructions for the high level assignment may not be executed atomically, we see where the race might occur. Each `x = x + 1` assignment expands into a machine code sequence similar to

```
LOAD R from x
INC R
STORE R into x
```

where R is a processor register and x is a memory address. If these low-level instructions for the two threads are interleaved, we might have a situation where both threads read x while it is still 0, both increment it to 1 and both write it back as 1. The final result in x is 1, different from what would have happened if all the instructions of one thread had been executed before those of the other.

Exercise 25
Construct a minimal case of determinacy race with two threads accessing variable x but only one of them writing to it. Show at least two ways of sequencing the machine code instructions that will cause different results.

There are ways of handling races using synchronization primitives such as the mutex. The strategy we use here is simply to ensure that strands that may execute in parallel

are independent. Any code with a determinacy race will be deemed illegal (for this programming model, it is possible to determine statically whether a code has a determinacy race or not).

Exercise 26
If you can solve this one without help, you are pretty good.
As written, several listings in chapter 27 of CLRS3 (third printing) contain an unintended determinacy race. Which listings? Where is the race? How could such a bug occur?
(When I noticed this, I wrote to Professor Leiserson, who wrote the chapter, and he confirmed I had found a severe bug, which would be fixed in the fourth printing.)

3.3 Case study: chess program

The CLRS3 textbook reports an interesting tale from the development of the award-winning chess program *Socrates.

The development machine had $P = 32$ processors. The target machine, not available to the developers, was to have 512. The running time on 32 processors was 65 seconds. One of the programmers found a way to bring this down to 40 seconds! Major improvement, right? Yet this change to the code was not incorporated in the final version, because an analysis based on work and span showed that it would have actually slowed down the execution on the target machine. How is this possible? And how could they figure it out before being able to try the code on the $P = 512$ machine?

The first theorem of the greedy scheduler tells us that $T_P \leq \frac{T_1}{P} + T_\infty$: treating the inequality as an equation to obtain an approximation for the running time instead of a bound, we get $T_P = \frac{T_1}{P} + T_\infty$; solving it for the span we get $T_\infty = T_P - \frac{T_1}{P}$.

With the original code, the work was $T_1 = 2048$, with $T_{32} = 65$. The approximation for the span thus gave $T_\infty = 65 - 2048/32 = 65 - 64 = 1$. Extrapolating these results to $P = 512$ gave $T_{512} = 2048/512 + 1 = 4 + 1 = 5$.

After the change in the code, instead, the work was $T'_1 = 1024$, with $T'_{32} = 40$. The approximation for the span gave $T'_\infty = T'_P - \frac{T'_1}{P} = 40 - 1024/32 = 40 - 32 = 8$. Extrapolating to 512 processors gave $T'_{512} = 1024/512 + 8 = 2 + 8 = 10$. In other words, the "optimization" would have made the code twice as slow on the target machine!

We note that the original code had pretty high parallelism of $T_1/T_\infty = 2048/1 = 2048$, which gave a parallel slackness of at least 4 even on the highly parallel target machine. The rearranged code, instead, exhibited a much lower parallelism of $1024/8 = 128$ giving a pathetic parallel slackness factor of just 1/4 on the target machine. This meant that the hardware parallelism of the target machine could not be fully exploited by the changed code and that's why the original code performed better.

The moral of this story is to trust the work and span metrics more than raw execution times when evaluating and predicting the performance of parallel algorithms.

Chapter 4

Geometric algorithms

Chapter contents

Intersection of segments. Convex hull: Graham's scan, Jarvis's march.
Expected coverage: about 1 lecture.

Textbook

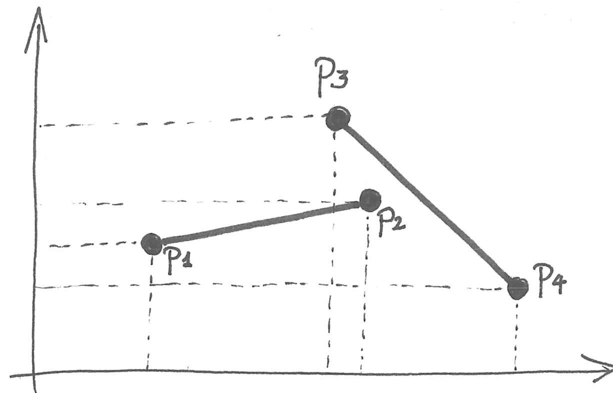
Study chapter 33 in CLRS3.

The topics included here show off some of the techniques that can be applied and provide some basic tools to build upon. Many more geometric algorithms are presented in the computer graphics courses. Large scale computer aided design and the realistic rendering of elaborate scenes will call for plenty of carefully designed data structures and algorithms.

Note that in this overview chapter we work exclusively in two dimensions: upgrading the algorithms presented here to three dimensions will *not*, in general, be a trivial extension.

4.1 Intersection of line segments

Our first problem is, given two line segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$ (4 endpoints p_1, p_2, p_3, p_4 and 8 coordinates $x_1, y_1, x_2, \ldots, y_4$), to determine whether they intersect—that is, whether they have any points in common. Since the input is of fixed size (8 real numbers), it should come as no surprise that this can be solved in $O(1)$. To make matters more interesting, then, we shall show how to compute this result efficiently, without using any trigonometry or even division. To that effect, let's first take a little detour.



4.1.1 Cross product

Our friend here, and for the rest of the chapter, is the cross product. A point $p = (x, y)$ in the Cartesian plane defines a point-vector $\vec{p}$ from the origin $(0, 0)$ to point p. The cross product $\vec{p_1} \times \vec{p_2}$ is defined as a third vector $\vec{p_3}$, orthogonal to the plane of $\vec{p_1}$ and $\vec{p_2}$, of magnitude given by the area of the parallelogram of sides $\vec{p_1}$ and $\vec{p_2}$, and such that $\vec{p_1}, \vec{p_2}$ and $\vec{p_3}$ follow the “right hand rule”.

Exercise 27

- Draw a 3D picture of $\vec{p_1}, \vec{p_2}$ and $\vec{p_3}$.
- Draw a 2D picture of $\vec{p_1}, \vec{p_2}$ and the parallelogram.
- Prove that the absolute value of the determinant of the matrix $\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}$, equal to $x_1 y_2 - x_2 y_1$, gives the magnitude of $\vec{p_3}$ and that its sign says whether $\vec{p_3}$ “comes out” of the plane ($\odot$) or “goes into” it ($\otimes$).

Having completed the exercise, you will readily understand that the sign of the cross product $\vec{p_1} \times \vec{p_2}$ tells us whether the (shortest) rotation that brings $\vec{p_1}$ onto $\vec{p_2}$ is positive (= anticlockwise) or negative. In other words, by looking at the sign of $\vec{p_1} \times \vec{p_2}$, we can answer the question “on which side of $\vec{p_1}$ does $\vec{p_2}$ lie?”. This can be used to provide efficient solutions to several problems. Let’s get back to one of them.

4.1.2 Intersection algorithm

Do the two line segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ intersect? (Note that we are no longer dealing with point-vectors.) We examine a tree of possibilities. Take the two endpoints p_1 and p_2 of the first segment and consider their position with respect to the second segment $\overline{p_3 p_4}$. Do p_1 and p_2 lie on the same side or on opposite sides of $\overline{p_3 p_4}$? If they lie on the same

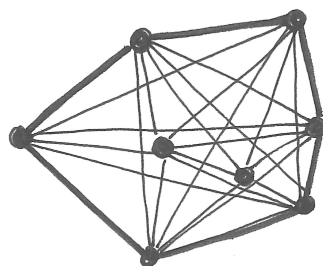
side, it's clear that the two segments **don't intersect**. If they lie on opposite sides that's a good start but[1], if we are unlucky, they only intersect the *continuation* of $\overline{p_3p_4}$, not the segment proper. So we should also turn the tables and verify the position of p_3 and p_4 with respect to $\overline{p_1p_2}$. If these are also on opposite sides, then the two segments **intersect**; if they are on the same side, the segments **don't intersect**.

In the above we have nonchalantly ignored the boundary case in which one of the endpoints being examined lies precisely on the line containing the other segment. When this happens, the relevant cross product is $\vec{0}$. We must detect all such cases and then check whether the point, as well as being collinear with the endpoints of the segment, also lies *between* them; if it does, in any of the possible configurations, the verdict will be that the segments **intersect**; if this never happens, instead, the segments **don't intersect**.

It is instructive to note how a substantial portion of the programming effort of producing a correct algorithm must go into the handling of this boundary case. This is not an isolated occurrence in computer graphics.

4.2 Convex hull of a set of points

The convex hull of a collection Q of points in the plane is the smallest convex[2] polygon such that all the points in Q are either inside the polygon or on its boundary.



> **Exercise 28**
> Sketch an algorithm (not the best possible: just any vaguely reasonable algorithm) to compute the convex hull; then do a rough complexity analysis. Stop reading until you've done that. *(Requires thought.)*

We present two algorithms[3] for building the convex hull of a set Q of n points. The first, Graham's scan, runs in $O(n \lg n)$ time. The second, Jarvis's march, takes $O(nh)$ where h is the number of vertices of the hull. Therefore the choice of one or the other

[1] Micro-exercise in mid-sentence: sketch in the margin a case in which they do and yet there is no intersection.

[2] A convex polygon is one in which, given any two points inside the polygon (boundary included), no point of the segment that joins them is outside the polygon. For examples of non-convex polygons think of shapes with indentations such as a star or a cross.

[3] Several others have been invented—see the textbook.

depends on whether you expect most of the points to be on the convex hull or strictly inside it.

Both methods use a technique known as "rotational sweep": they start at the lowest point and examine the others in order of increasing polar angle[4] from a chosen reference point, adding vertices to the hull in a counterclockwise fashion. This leads to an interesting sub-problem: how can you efficiently sort a group of point-vectors by the angles that they form with the positive x axis?

The simple-minded answer is to compute that polar angle (call it $\theta(\vec{p})$) for each point-vector $\vec{p}$ using some inverse trigonometric function of $\vec{p}$'s coordinates and then to use it as the key for a standard sorting algorithm. There is, however, a better way. Here, too, the cross product is our friend. Actually…

Exercise 29
How can you sort a bunch of points by their polar coordinates using efficient computations that don't include divisions or trigonometry? (Hint: use cross products.) Don't read beyond this box until you've found a way.

Hey! I said stop reading! I know, the eye is quick, the flesh is weak… but just do the exercise first, OK? You'll be much better off at the exam if you work out this kind of stuff on your own *as you go along.*

Anyway, the trick is that there is no need to *compute* the actual angles in order to sort the point-vectors by angle—all that is needed is a plug-in replacement for the core comparison activity of "is this point-vector's polar angle greater than that of that other point-vector?". This, regardless of the actual values of the angles, is equivalent to asking whether a point-vector is to the left or the right of another; therefore the question is answered by the sign of their cross product, as follows:

$$\theta(\vec{p}) \lesseqgtr \theta(\vec{q}) \iff \vec{p} \times \vec{q} \gtreqless 0.$$

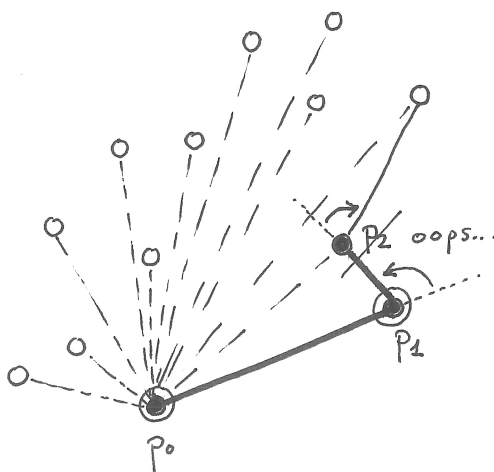
4.2.1 Graham's scan

Although the details are a bit tricky, the basic idea of the algorithm is quite straightforward: starting from the bottom point, with the origin on the bottom point, we go through all the others in order of their polar angle; each point is visited only once and is retained in a stack if it looks (so far) as being on the hull; it may be removed later by backtracking if we discover it wasn't after all.

In greater detail, the algorithm is as follows. We start from the point, call it p_0, with the smallest y coordinate (the leftmost such point in case of ties). We sort all other points based on their polar angle from p_0. We then examine these points in that order to build the boundary of the hull polygon. At any instant we have a "most recently discovered

[4]The polar angle of a point p_1 with respect to a reference point p_0 is the angle between the positive x axis and the point-vector $\vec{p_1} - \vec{p_0}$. In other words, place the butt end of an East-pointing arrow on p_0, then rotate that arrow around p_0 until it points towards p_1. The angle travelled is the polar angle of p_1 with respect to p_0.

vertex" (initially p_0) and, by drawing a segment from that to the point being examined, we obtain a new candidate side for the hull. If this new segment "turns left" with respect to the previous one, then good, because it means that the hull built so far is still convex. If instead it "turns right", then it's bad, because this introduces a concavity. In this case we must eliminate the "most recently discovered vertex" from the boundary (we now know it wasn't really a vertex of the hull after all—it must be an internal point if it's caught at the bottom of a concavity) and backtrack, reattaching the segment to the *previous* vertex in the hull. Then we must again check which way the end segment turns with respect to its predecessor, and we must keep backtracking if we discover that it still turns right. We continue backtracking until the end segment does not introduce a concavity. This is basically it: we proceed in this way until we get back to the starting point p_0.



The boundary case is that in which the new segment doesn't turn either left or right but instead "goes straight": this indicates a situation with more than two collinear "vertices" along the edge of the hull and of course we must eliminate the inner ones from the list of vertices since they're actually just ordinary points in the middle of an edge.

The backtracking procedure may appear complex at first but it is actually easy to implement with the right data structure—namely a stack. We just keep feeding newly found vertices into the stack and pop them out if they are later found to contribute to concavities in the boundary.

A formal proof of correctness of the algorithm is in the textbook.

As for complexity analysis, the initial step of sorting all the points by their polar angle can be performed in $O(n \lg n)$ using a good sorting algorithm and the cross product trick mentioned earlier[5]. It can be shown that no other section of the algorithm takes more than $O(n)$; therefore the overall complexity of Graham's scan is $O(n \lg n)$.

4.2.2 Jarvis's march

Here too we start from the bottommost leftmost point of the set and work our way anticlockwise, but this time we don't pre-sort all the points. Instead, we choose the point with the *minimum* polar angle with respect to the current point and make it a vertex of

[5] Actually, the use of arcsin instead of the cross product, though much less efficient, would not, of course, change the asymptotic complexity.

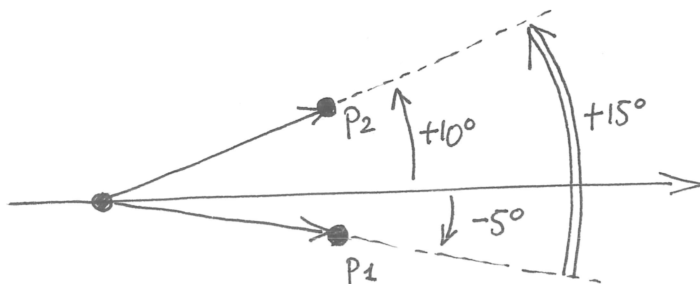
the hull: it must definitely be on the boundary if it's the first one we encounter while sweeping up from the "outside" region. We then make this new-found vertex our new reference point and we again compute the minimum polar angle of all the remaining points. By proceeding in this way we work our way up along the right side of the hull, eventually reaching the topmost point. We then repeat the procedure symmetrically on the other side, restarting from the bottommost point. Finally, if necessary, we deal with the boring but trivial boundary cases of "flat top" or "flat bottom".

Complexity analysis is easy: finding the minimum is an $O(n)$ operation given that each individual comparison is $O(1)$; this operation is repeated once for every vertex of the hull and therefore the overall complexity of Jarvis's march is $O(nh)$.

Exercise 30
Imagine a complex CAD situation in which the set contains about a million points. Which of the two algorithms we have seen would you use to compute the convex hull if you expected it to have about 1,000 vertices? And what is the rough boundary value (for the expected number of vertices of the hull) above which you would use one algorithm instead of the other? (Is this a trick question? If so, why?)

The motivation for performing distinct left and right passes is to be able to use the cross-product trick—but there are subtleties. Let's see.

Distinct left and right passes are needed if one considers the polar angles with respect to the shifted (positive or negative) x axis *and* wants to use the cross-product trick. This is because you can't get accurate results from the cross-product trick if the angle spanned by the two vectors being multiplied is greater than π.



Why? Assume p_1 is at 5 degrees below the x axis and p_2 is at 10 degrees above: then the shortest rotation from p_1 to p_2 is positive (anticlockwise), specifically +15 degrees, and indicates that p_1 has a smaller polar angle than p_2. This is indeed correct if we consider p_1 to have polar angle -5 degrees and p_2 to have +10 degrees, but it's incorrect if we say that we normalize all angles to within the 0 to 2π range, because in that case p_1 has polar angle of +355 degrees which is *not* smaller than p_2's! Normalizing angles to any other range spanning 2π (eg $-\pi$ to $+\pi$) might fix the problem for this particular counterexample but not in general: any two points on either side of the discontinuity (in

this case at angle π; in the previous case at angle 0) and separated by less than π would exhibit the same problem, regardless of where the discontinuity is placed.

It is possible to use a single anticlockwise pass if one *computes* the polar angles with respect to the shifted positive x axis (using trigonometry), thus without using the cross-product trick (but that costs more, by a constant factor).

A smarter programmer will however use both a single anticlockwise pass *and* the cross-product trick by considering polar angles not with respect to the x axis but with respect to the last segment of the hull found so far.

Finally, have your say

Before the end of term, please visit the online feedback pages and leave an anonymous comment, explicitly mentioning both the good and the bad aspects of this course. If you don't, your own opinion of my work will necessarily be ignored. Instead, I'd be very grateful to have a chance to hear about it.

End of lecture course

Thank you, and best wishes for the rest of your Tripos.