Algorithms
Exercise Sheet
(first half of the course)

Andrej Ivašković (ai294)
Compiled on: 25th January 2021

Recommended additional work

I highly recommend making use of the *Introduction to Algorithms* (CLRS) textbook. Manber's *Introduction to Algorithms: A Creative Approach* has some interesting exercises, proofs and covers certain topics in a concise and approachable way. Sedgewick's many books on algorithms also cover some topics in an easy to digest way. If you are looking for a challenge, Knuth's *The Art of Computer Programming* books (especially Volume 3), as well as *Concrete Mathematics* contain interesting supplementary material.

If you are keen on doing algorithmic exercises, I suggest looking into ICPC-like contests and invite you to participate at UKIEPC in Michaelmas term. Websites such as HackerRank and Codeforces will help you get better at these kinds of problems and prepare for job interviews.

Some of these problems are based on existing exercise sheets provided by lecturers or other supervisors. Many thanks to Professor Frank Stajano, Dr Damon Wischik, and Petar Veličković.

Have a look at the exercises in the lecture notes and feel free to ask me any questions about any of them.

1 Sorting

1.1 Before attempting the exercises

This set of exercises covers the first topic covered in the lectures: sorting algorithms. Throughout the chapter, you also explore concepts in searching, computational complexity, and algorithm design.

The hardest part of this section of the course is probably coming to terms with reasoning formally and mathematically about algorithms. It also requires a lot of creativity on your part – you develop that with practice. I strongly encourage looking up the so-called 'master theorem' for solving recurrences.

1.2 Problems

Exercise 1.1. Find the asymptotic solutions of the following recurrences:

(*a*) $T(n) = T(n-1) + 5n^2 - 3n$

(*b*) $T(n) = T(n/2) + c$

(*c*) $T(n) = 2T(n/4) + \log n$

(*d*) $T(n) = T(\sqrt{n}) + c$

Exercise 1.2. Prove that Bubble Sort, as presented in the lectures, will never make more than n passes of the array.

Exercise 1.3. One optimisation of Insertion Sort illustrated in the lectures and the notes is using *binary search* in order to find the appropriate place where the new item will be inserted.

Write a Java method that implements binary search. It should take as arguments an array sorted in increasing order[1] with element type `T`, a comparator that represent this ordering, and a value of type `T`. It should return a position in the array whose value is equal to the value given in the argument, with `-1` returned in case no such item exists.

Exercise 1.4. Given an array a of n integers and q queries defined by pairs $\langle x_i, y_i \rangle$ representing 'how many items in a are strictly greater than x_i and strictly less than y_i', populate an array of length q with the answers of the respective queries. Assume both n and q are bounded by 10^6. Give efficient solutions and state their time complexities with the following underlying assumptions:

(*a*) all items are positive integers less than 1000;

(*b*) all items are signed integers, no greater than 10^9 in magnitude.

Exercise 1.5.

(*a*) Describe an algorithm that performs $n + \lceil \log_2 n \rceil - 2$ pairwise comparisons in order to find the second smallest of n items.

(*b*) In practice, is this algorithm ever useful or notably better than the simple one?

Exercise 1.6. Write pseudocode for the bottom-up version of Merge sort.

Exercise 1.7. Somebody claims that the best way of sorting an array is by converting an array to a list, deleting the original array, and then performing Merge sort on these list, which requires $O(1)$ additional space and $O(n \log n)$ time. Comment on this.

Exercise 1.8. The Quick sort algorithm can be slightly modified to give the least k items of an array of length n, not in a sorted order – this algorithm is

[1] I make a distinction between increasing and *strictly increasing*, where the latter implies that that there are no items that are equal. An alternative is to refer to them as *non-decreasing* and increasing, respectively – I do not like this terminology.

sometimes called *Quickselect*.

(*a*) What modifications do you have to make?

(*b*) Argue why the average case complexity of Quickselect is $O(n)$.

Exercise 1.9. Show how Heap sort sorts the array `[7 1 5 3 4 2 8 6]` in increasing order.

Exercise 1.10. Consider the Quicksort, Merge sort and Heap sort algorithms. All of them have certain advantages and disadvantages. State them and discuss when you would use one over the other.

Exercise 1.11. What is the least and the greatest number of elements that a binary heap of height k may have?

Exercise 1.12. Suppose you are given an array of $\langle k_i, v_i \rangle$ pairs, where k_i are integers in the range between -1000 and 1000. Explain how Counting, Bucket and Radix sort would sort such data, with only k_i being considered for comparison.

Exercise 1.13. Why does Radix sort require the underlying sorting algorithm to be stable? Is it possible to modify any comparison-based sorting algorithm to make it stable?

2 Dynamic programming and greedy algorithms

2.1 Before attempting the problems

This set of exercises covers dynamic programming and greedy algorithms. Most of these require you to devise algorithms using some of the design ideas covered in the course. These problems do not require you to know almost any bookwork, just use experience and try out possible approaches. As an additional bonus, this will give you some more experience tackling coding questions in job interviews.

To solve a dynamic programming problem, think about what it means to solve a particular problem instance, and how the solution differs from the one you get if you, say, remove a subset of the data. Usually it involves you creating a $helper[i]$ array, where $helper[n]$ consider the entirety of the data. If you introduce such a *helper*, you should consider two things: the *base case* and the *recurrence*. If at first you fail, try a variation of this approach: you can try recurrences that choose these subsets in a different way; you might want to consider a multidimensional array; you can try solving a slightly different problem; you might want to sort the data according to something first. If you are asked to compute a set of data that optimise something, you can usually 'work backwards' to perform reconstruction. In some cases, you can greatly reduce the space complexity (especially if dealing with a multidimensional array).

As before, whenever you are asked to devise an algorithm, either pseudocode or a sufficiently detailed explanation will do. If you write pseudocode, make it clear and easy to read.

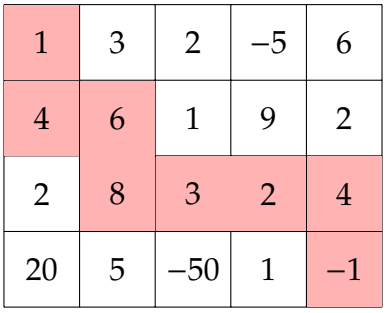

Figure 1: Maximum sum: 27

2.2 Problems

Exercise 2.1. For an integer array a with length n, define its *nonconsecutive subsequence* as a subsequence[2] of a that does not contain any two items that have adjacent positions in a. Its *maximum nonconsecutive subsequence* is the nonconsecutive subsequence of a whose sum is greatest out of all nonconsecutive subsequences. For example, for the array `[3, -2, -1, 4, 5, 7]`, its maximum nonconsecutive subsequence is `[3, 4, 7]`.

Devise a dynamic programming algorithm that computes the maximum achievable sum. How can you then reconstruct the sequence?

Exercise 2.2.

(*a*) A robot walks on an $n \times m$ map represented by a matrix of integers. Whenever it steps on a tile, it will gain the number of points equal to the value on this tile. Suppose the robot starts moving from the top left corner of the array and aims to reach the lower right corner by only moving one tile to the right or one tile down in every turn. Compute the maximum number of points achievable with such movement. See Figure1 for an example.

(*b*) [★] What if you can also move up, and you cannot step on a tile more than once?

Exercise 2.3. Recall the *knapsack* problem explored when you covered greedy algorithms in lectures. It is an NP-complete problem.[3] Devise a dynamic programming algorithm for the case when all item masses are positive integers and the knapsack capacity is also a positive integer. Assume that this capacity is v and that there are n items, and that $v \leq 10^4$ and $n \leq 5000$.

Exercise 2.4. Suppose you are playing a game where you control a character at the bottom of the screen and some objects fall down as time passes, and you collect points for collecting these. Suppose there are n such objects, where the

[2] a sequence x is a subsequence of sequence y if x can be derived from y by removing some of its items

[3] Without going into a proper complexity theory definitions, there are no known polynomial time algorithms for NP-complete problems, and the existence of a polynomial time algorithm for one of them would imply the existence of a polynomial time algorithm for all other NP-complete problems.

symbol	A	B	C	D	E	F
frequency	263	451	73	83	59	71

Figure 2: Character frequencies in the document

i-th one will be at the bottom of the screen at time t_i, it will add s_i to your score if you manage to catch it, and it will be at position x_i. All of these numbers are positive integers. At time 0 you are at position p and at every time step you can move one space to the left or one step to the right. How can you compute the top achievable score? There are several possible solutions depending on what the constraints are. Consider:

- $\max_i x_i \leq 5000$, $\max_i t_i \leq 5000$ and $n \leq 10^6$, all of the other values are 32-bit values;
- $n \leq 5000$, all of the other values are 32-bit integer values.

Choose one of these sets of constraints and solve the problem in that case. Briefly outline the approach that you would make in order to solve the problem in the other case.

Exercise 2.5. The lectures explored a greedy approach to solving the knapsack problem. Provide a counterexample that demonstrates that it will not work in all cases.

Exercise 2.6. *(based on CLRS3, problem 16-1)* Consider the problem of making change for n pennies using the fewest number of coins. Assume the denominations are $a_1, \ldots, a_k$ which are all integers, and one of which is 1p.

(*a*) Describe a greedy $O(k \log k)$-time algorithm to make change. What is it time complexity? Why is the constraint of there being 1p coins important?

(*b*) Suppose the available coins are in the denominations that are powers of c: the denominations are $c^0, c^1, \ldots, c^{k-1}$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm you described always yields an optimal solution.

(*c*) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. You should still assume the existence of a 1p coin.

Exercise 2.7. Consider a document in which only symbols `A`, `B`, `C`, `D`, `E` and `F` occur, with frequencies given in Figure 2. Compute its Huffman code, the total length of the encoded document in bits, as well as the average codeword length.

3 Abstract data types and machine implementations

3.1 Before attempting the problems

This small set of exercises is aimed at helping you get used to using ADT terminology.

I suggest you look at the standard libraries of all the programming languages you use and write a few simple programs that make use of data structures that implement some of these ADTs. Sometimes you will find exactly specified running times for their operations.

3.2 Problems

Exercise 3.1. Write a Java implementation of the `Deque` ADT using a doubly-linked list underlying implementation. Your implementation should not use any of the Java `Collection`s.

Exercise 3.2. A colleague tells you that 'anything you want to do with sets, you can do with dictionaries' and advises you to never use sets. Comment on this.

Exercise 3.3. Suppose you are given a function `f` that takes as argument a pair of integers and returns an integer. You know that it is a pure function (it has no side effects and cannot raise any exceptions), but have no access to its implementation. How can you then write a `memoized_f` function that has the same signature as `f` and implement memoization? Feel free to refer either to ADTs or their concrete implementations.

4 Data structures

4.1 Before attempting the problems

These exercises are concerned with the data structures explored in Chapter 4 of the notes. Think of ADTs as interfaces and the other data structures as their 'concrete' implementations.

Before understanding how the operations of a particular data structure are implemented, you should first ask yourself what task it is solving and what the time complexities of these operations are. This allows you to understand better why we are even considering such a structure and what it is used for.

The tree-like data structure you will see throughout this course will usually involve some logarithm in the time complexities of most operations, compared to the 'naive' (usually array or list based) approach in which some operations take constant time, and others take much longer.

Note that 2-3-4 trees are just a special case of B-trees.

You are *strongly encouraged* to implement some of these data structures in Java.

4.2 Problems

Exercise 4.1.

(*a*) Given a binary search tree t and its node n, how can you find its successor? Prove that your approach is correct. What is the time complexity of this operation?

(*b*) Prove that, in a binary search tree, if node n has two children, then its successor has no left child.

(*c*) *(CLRS3, 12.2-4)* Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.

Exercise 4.2. Show how 2-3-4 trees are equivalent to red-black trees by providing a way to 'encode' an arbitrary red-black tree as a 2-3-4 tree and a way to 'encode' 2-3-4 trees as red-black trees.

Exercise 4.3.

(*a*) Show that a red-black tree with b non-leaf black nodes has between b and $3b$ non-leaf nodes.

(*b*) Show that a red-black tree with r red nodes has at least $r + \lceil r/2 \rceil$ non-leaf nodes. What is the greatest number of non-leaf nodes if there are r red nodes?

(*c*) What are the least and greatest possible number of nodes of a red-black tree of height h, where the height is the length in edges of the longest path from root to leaf?

Exercise 4.4. How do you handle deletions in red-black trees? You do not need to provide a detailed answer.

Exercise 4.5. Using a soft pencil, a large piece of paper and an eraser, draw a B-tree with $t = 2$, initially empty, and insert into it the following values in order:

$$63, 16, 51, 77, 61, 43, 57, 12, 44, 72, 45, 34, 20, 7, 93, 29$$

How many times did you insert into a node that still had room? How many node splits did you perform? What is the depth of the final tree? What is the ratio of free space to total space in the final tree? What does the equivalent red-black tree look like?[4]

Exercise 4.6.

(*a*) Compare *chaining* and *open addressing* methods of hashing. When is it useful to use one over the other?

(*b*) How do you handle deletions in these two cases? Consider the different probing techniques that might be used.

Exercise 4.7. Data structures based on hashing have worst case linear time complexity for their operations, but typically operate in constant time. Devise

[4]Sorry for the tedious exercise. I do, however, encourage you to attempt it and do it by hand – if only to practice doing this under time pressure.

a data structure based on hashing that has worst case logarithmic time for its operations. Discuss the implications of your approach.

Exercise 4.8. Explain how priority queues can be used to implement Huffman coding. What is the running time of Huffman coding if a binary heap is used as the underlying implementation of the priority queue?

Exercise 4.9. Show that steps taken (in form of 'snapshots') during the lifetime of a binomial heap h_1 that is initially empty, and then the following operations are performed: `insert` 5, `insert` 8, `insert` 2, `insert` 9, `insert` 11, `extractMin`, `insert` 15, `insert` 3, `extractMin`, `extractMin`, `insert` 4, `insert` 7, `extractMin`.

Exercise 4.10.

(*a*) Prove that the sequence of trees in a binomial heap exactly matches the bits of the binary representation of the number of elements in the heap.

(*b*) Suggest a way to reduce the complexity of `first()` to $O(1)$.

Exercise 4.11. [★] What might be the signature of a *priority deque* ADT? Describe its efficient concrete implementation. You should either provide pseudocode or a sufficiently detailed explanation.

Exercise 4.12. You are tasked with implementing a component of a server that will manage a key-value store, where keys are strings and values are positive integers. Clients will send `get(key)` and `set(key, value, timeout)` queries to the server: the former asks the server to retrieve the value associated with `key`, and expects `0` if there is none; the latter sets the value associated with `key` to `value`, but this expires after `timeout` milliseconds. This queries get converted into `get(time, key)` and `set(time, key, value, timeout)` queries by the server socket, which contain the time of receipt. Your job is to implement these two calls (assuming that you are received the queries in the strictly increasing order of `time`). Explain how you would do this in the following cases:

(*a*) after the most recent value of a particular key expires, there is no associated value;

(*b*) after the most recent value of a particular key expires, the associated value is the most recently assigned one to that key that would not have expired so far (if none, `0`);

(*c*) after the most recent value of a particular key expires, the associated value is the highest one that would not have expired so far (if none, `0`).

[★] How would you modify your solutions if the `set` call had arguments `f` and `arg` instead of `value`, where `f` is a (possibly computationally expensive) pure function that can take `arg` as its argument and the value associated with `key` should be set to `f(arg)`?