Algorithm	Worst-case running time	
any algorithm	$\Omega(n \log n)$	
InsertSort	$\Theta(n^2)$	
BinaryInsertSort	$\Theta(n^2)$	🙂 #comparisons is $O(n \log n)$
SelectSort	$\Theta(n^2)$	🙂 #writes is $O(n)$
QuickSort	$\Theta(n^2)$	🙂 average case is $\Theta(n \log n)$
MergeSort	$O(n \log n)$	😈 uses $\Theta(n)$ extra memory
HeapSort	$O(n \log n)$	👍 😂 👍

2.11 Quicksort

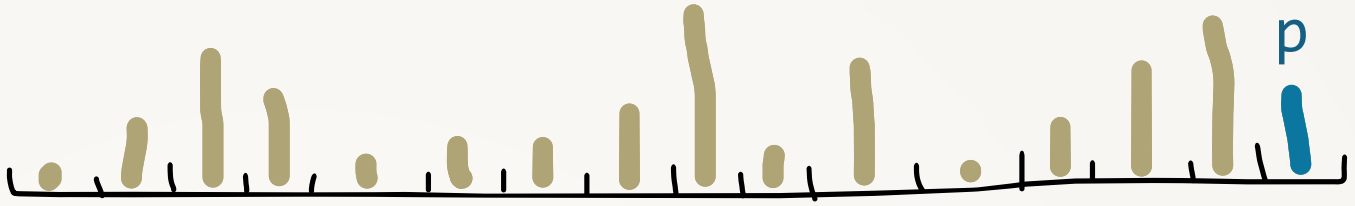
```
let rec quicksort = function
  | [] -> []
  | [x] -> [x]
  | pivot::xs ->
      let rec partition lefts rights = function
        | [] -> (quicksort lefts) @ (pivot :: quicksort rights)
        | y::ys ->
            if (y <= pivot) then
               partition (y::lefts) rights ys
            else
               partition lefts (y::rights) ys
      in
      partition [] [] xs
```

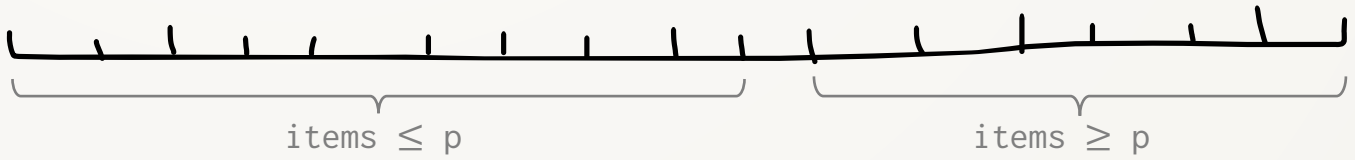
Our OCaml quicksort copies the items. It is still pretty fast, and it is much easier to understand. It is not hard to prove that quicksort does $n \log n$ comparisons, in the average case.

Let's implement it more carefully, being explicit about assignments as well as comparisons, so we can find the *total* running time.

```
def quicksort(x):
```

1. Pick the last item to be the pivot, $p = x[\text{len}(x) - 1]$.
2. Partition the array, so that it has the form $(\text{items} \leq p) :: p :: (\text{items} \geq p)$
3. The pivot p is now in its correct place. Call quicksort on the left portion, and on the right portion.

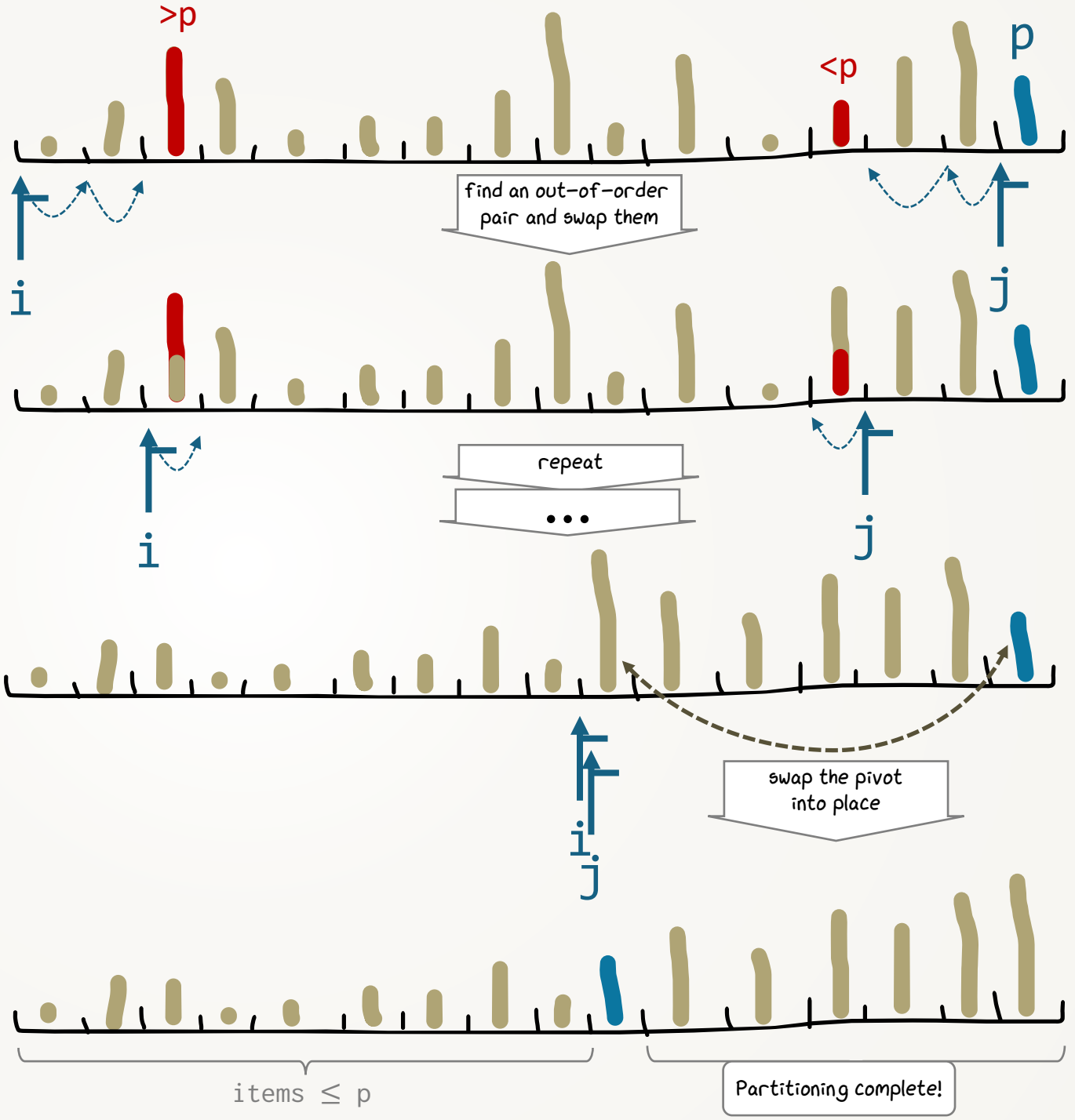




```
def quicksort(x):
```

1. Pick the last item to be the pivot, $p = x[\text{len}(x) - 1]$.
2. Partition the array, so that it has the form $(\text{items} \leq p) :: p :: (\text{items} \geq p)$
3. The pivot p is now in its correct place. Call quicksort on the left portion, and on the right portion.

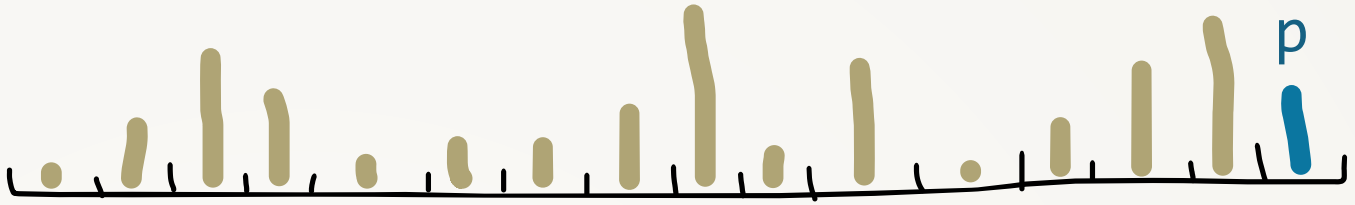
```
def partition(x, p):
  i = just before first item
  j = just before p
  while True:
    while i < j and x[i] <= p: i++
    while i < j and x[j-1] >= p: j--
    if i < j:
      swap x[i] with x[j-1]
      i++, j--
  swap p with x[j]
```

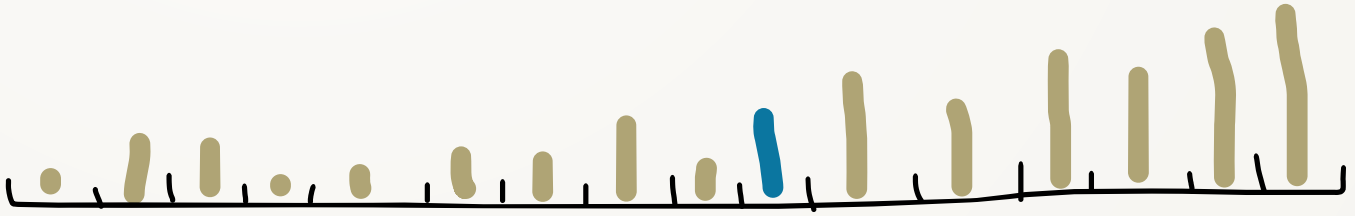


```
def quicksort(x):
```

1. Pick the last item to be the pivot, $p = x[\text{len}(x) - 1]$.
2. Partition the array, so that it has the form $(\text{items} \leq p) :: p :: (\text{items} \geq p)$
3. The pivot p is now in its correct place. Call quicksort on the left portion, and on the right portion.

```
def partition(x, p):
  i = just before first item
  j = just before p
  while True:
    while i < j and x[i] <= p: i++
    while i < j and x[j-1] >= p: j--
    if i < j:
      swap x[i] with x[j-1]
      i++, j--
  swap p with x[j]
```

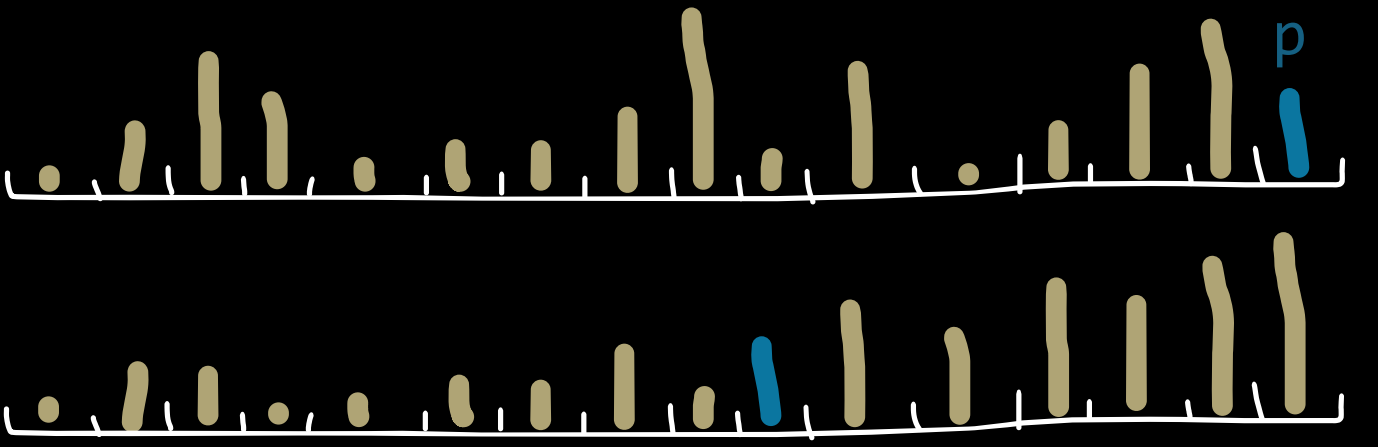




```
def quicksort(x):
```

1. Pick the last item to be the pivot, $p = x[\text{len}(x) - 1]$.
2. Partition the array, so that it has the form $(\text{items} \leq p) :: p :: (\text{items} \geq p)$
3. The pivot p is now in its correct place. Call quicksort on the left portion, and on the right portion.

```
def partition(x, p):
  i = just before first item
  j = just before p
  while True:
    while i < j and x[i] <= p: i++
    while i < j and x[j-1] >= p: j--
    if i < j:
      swap x[i] with x[j-1]
      i++, j--
  swap p with x[j]
```



QUESTION
If we manage to split the array in half each pass, what's the running time (O)?

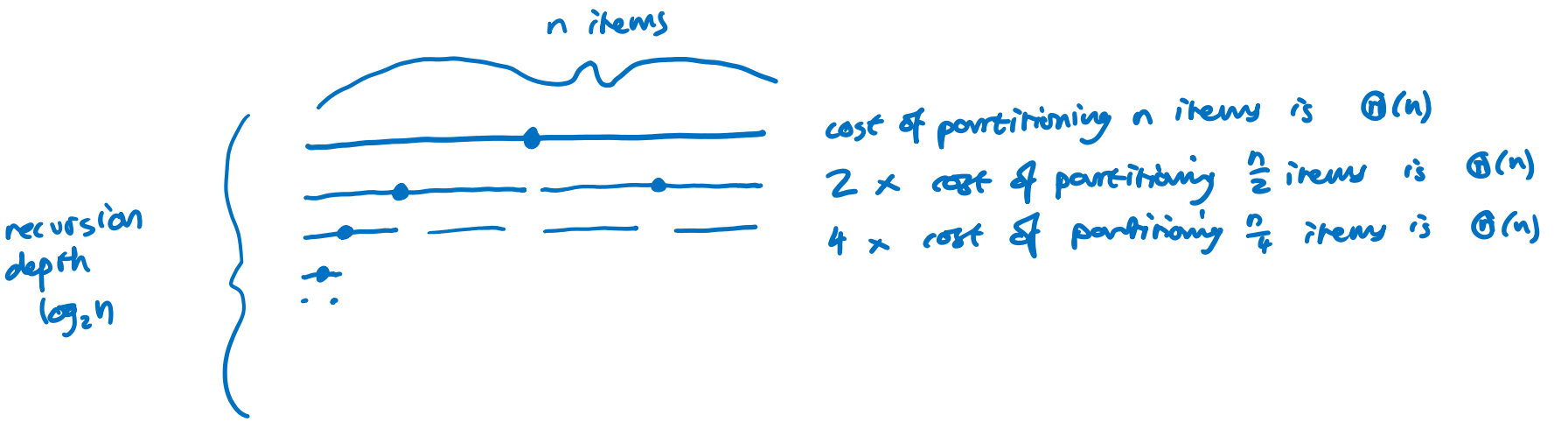
FACT. In the input is random (all permutations equally likely) then the expected running time is $\Theta(n \log n)$.

QUESTION
What's the worst-case running time (Ω)?

EXERCISE. The worst-case running time is $O(n^2)$.

FACT. In fact, QuickSort's best-case running time is $\Omega(n \log n)$.

Suppose we happen to find a pivot that partitions the array perfectly into two halves of size $\approx n$.



Total running time is $\Theta(n \log n)$

After studying Dynamic Programming next week, you'll be able to prove that the best case is $\Omega(n \log n)$.

In other words, the algorithm always takes $\geq n \log n$ steps (in the correct asymptotic sense) whatever the input.

If we happen to pick a bad pivot,
such that all other items are smaller than it,
on every single pass, then

partition n items: cost n
partition n-1 items: cost n-1
partition n-2 items: cost n-2

} total cost is $\Theta(n^2)$.

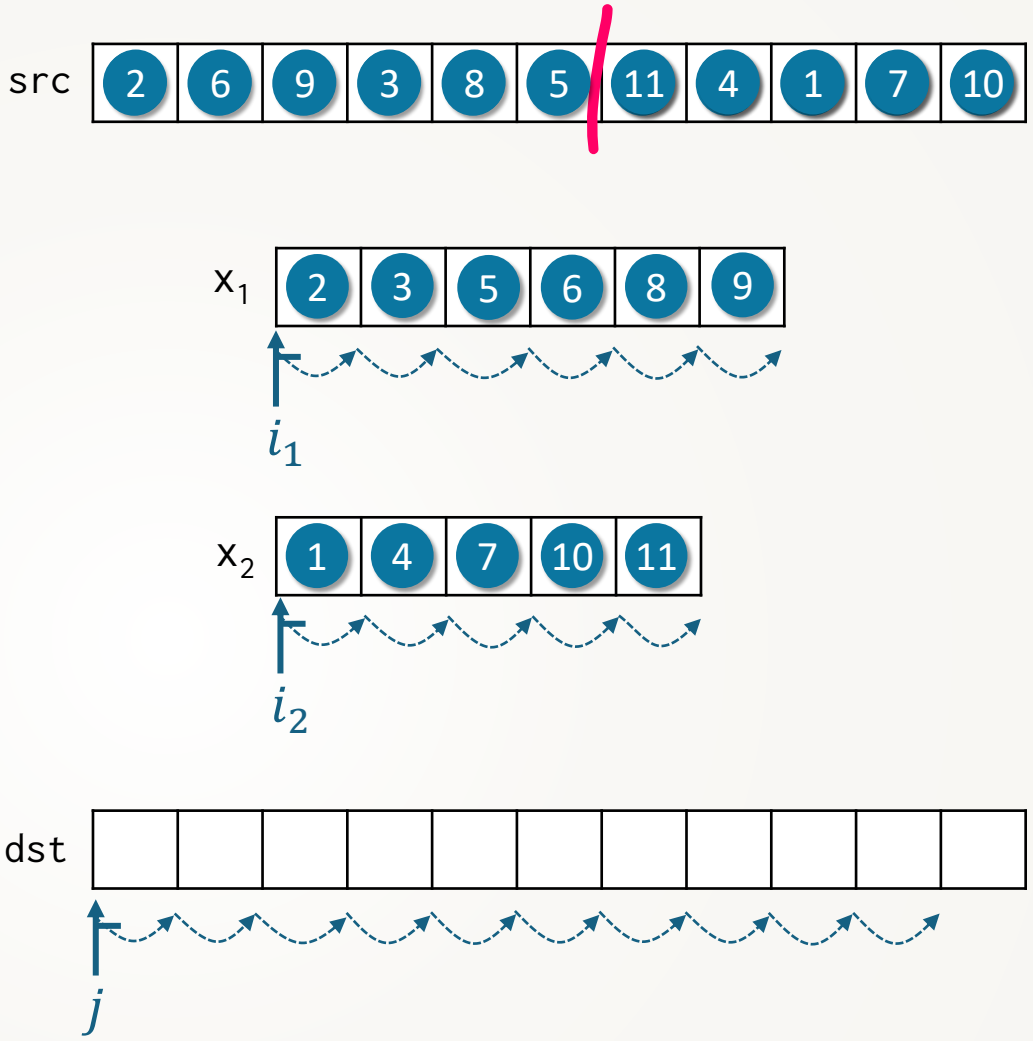
Thus the worst-case for Quicksort is $\Omega(n^2)$.

We can also show (using a similar argument) that the worst case is $O(n^2)$,
hence the worst case is $\Theta(n^2)$.

2.9 MergeSort

```
def mergesort(src, dst):
  n = len(src)
  m = int(n/2)

  x1 = new array of length m
  mergesort(src=src[0:m], dst=x1)
  x2 = new array of length n-m
  mergesort(src=src[m:n], dst=x2)

  merge x1 and x2 into dst
  free x1 and x2

  (unless n==1, in which case
  just copy src[0] into dst[0])
```

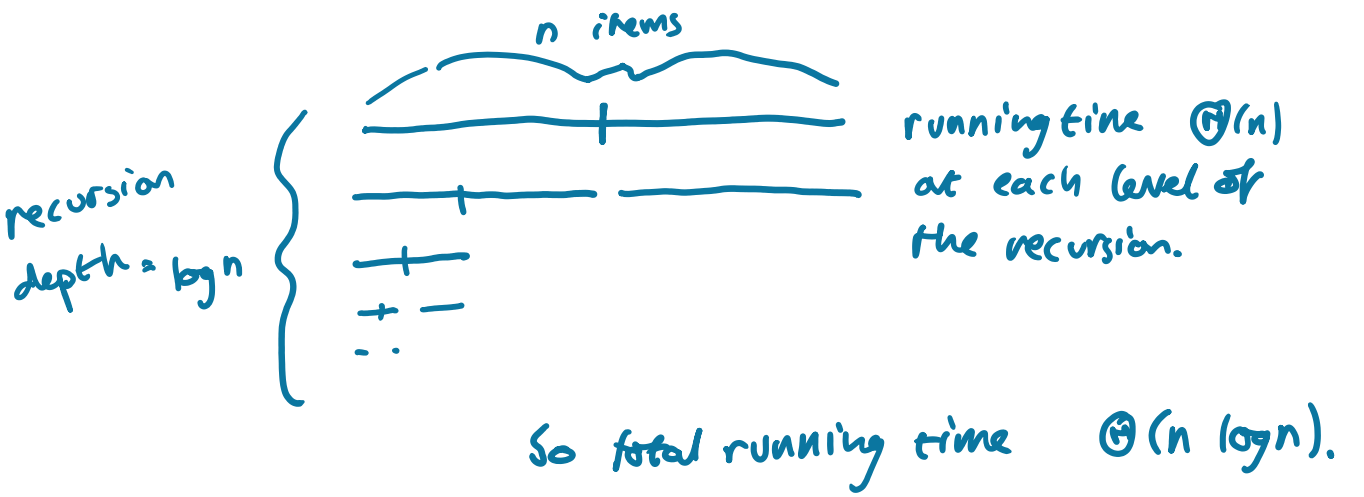
```
def merge(x1, x2, dst):
  # assert len(dst) == len(x1)+len(x2)
  i1,i2 = 0,0
  for j in 0..(len(dst)-1):
    dst[j] = min(x1[i1], x2[i2])
    advance i1 or i2 appropriately
```



Assume that the time to create an array of size m, and to free it, is $\Theta(m)$.

```
def mergesort(src, dst):
  n = len(src)
  m = int(n/2)

  x1 = new array of length m
  mergesort(src=src[0:m], dst=x1)
  x2 = new array of length n-m
  mergesort(src=src[m:n], dst=x2)

  merge x1 and x2 into dst
  free x1 and x2

  (unless n==1, in which case
  just copy src[0] into dst[0])
```

```
def merge(x1, x2, dst):
  # assert len(dst) == len(x1)+len(x2)
  i1,i2 = 0,0
  for j in 0..(len(dst)-1):
    dst[j] = min(x1[i1], x2[i2])
    advance i1 or i2 appropriately
```



Tick 1, deadline 5 Feb at noon

Algorithms tick mergesort

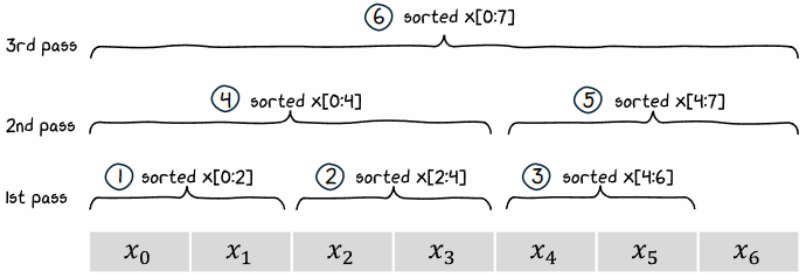
cl.cam.ac.uk/teaching/2324/Algorithm1/ticks/mergesort.html

Algorithms tick: mergesort

Bottom-up memory-constrained mergesort

In this tick you will implement a memory-efficient version of mergesort, mentioned at the end of section 2.9 in lecture notes.

This mergesort is bottom-up, as opposed to the standard top-down (recursive) version. In the first pass, it groups elements into pairs and sorts each pair by merging two chunks of size 1. In the second pass, it groups elements into quadruples and sorts each quadruple by merging two chunks of size 2. In the next pass, it merges these quadruples into sorted octuples. And so on, until the entire array is sorted. Note that most chunk sizes are a power of two, but the final chunk in each pass may be smaller. Here is the sequence of merges involved in sorting an array of length 7:



To merge two chunks, the strategy is to copy the right-hand chunk into scratch space, then merge the two chunks back into the original array. The merge should be performed right-to-left in order to avoid overwriting values from the left-hand chunk. You are given a scratch space of size `floor(n/2)`. (It's this tight limit on scratch space that forces us to merge in this way. In the example above with $n = 7$, the final step merges a chunk of size 4 and a chunk of size 3, and the scratch space is too small to fit the size-4 chunk.)

I asked ChatGPT for code, and its two attempts are both incorrect. Nonetheless, it has the general idea right.

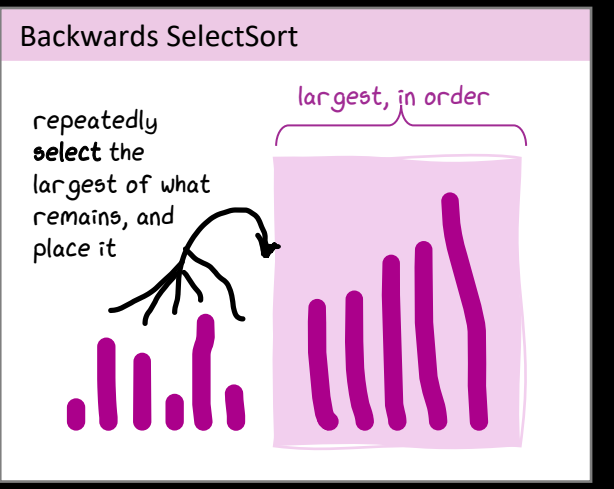
Task 1. Demonstrate that ChatGPT's code doesn't work. You should find an array which causes ChatGPT's `mergesort1` to fail with an exception, and another array for which `mergesort2` doesn't trigger an exception but does result in an incorrect answer. Both arrays should have size > 4.

With cunning, we can implement mergesort using only $\lfloor n/2 \rfloor$ extra space.

Can we sort in $O(n \log n)$ without using extra memory?

OUR STARTING POINT: SELECTSORT

SelectSort is in-place, and it's efficient in terms of swaps. But it uses very many comparisons, because on each pass it re-scans all remaining items to find the maximum.

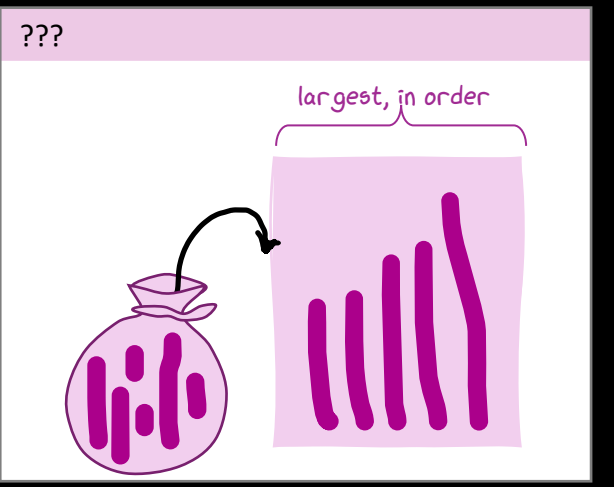


QUESTION

In each scan we do lots of comparisons, and learn a lot about the values. How might we save this information, to reuse in the next scan?

WHAT WE WANT

A data structure that is efficient for repeatedly extracting the maximum item.

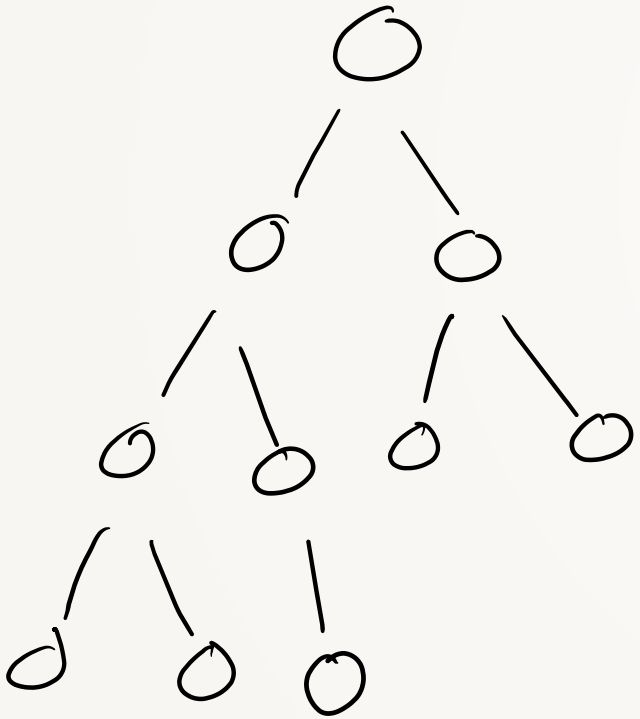


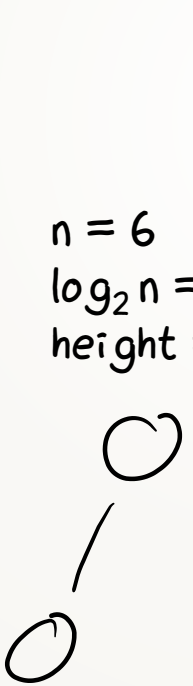
2.10 HeapSort

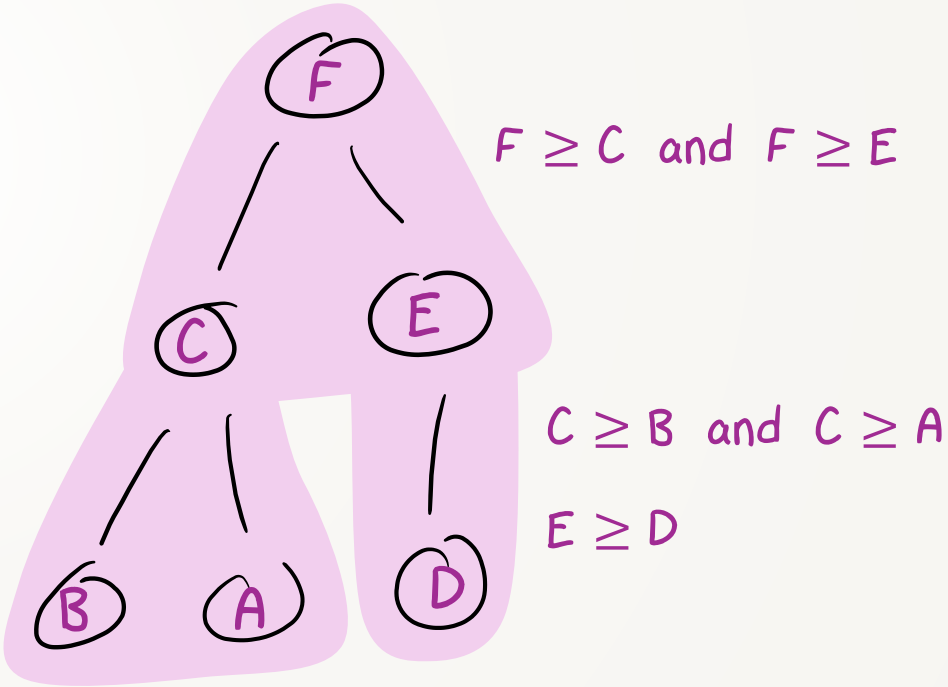
A *heap* is an almost-full binary tree that satisfies the heap property.

every level except possibly the bottom is full, and any spaces in the bottom level come at the right-hand end

everywhere in the tree, parent value ≥ child values





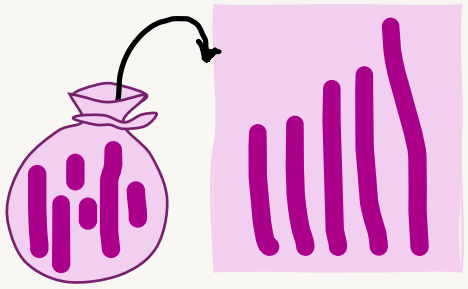


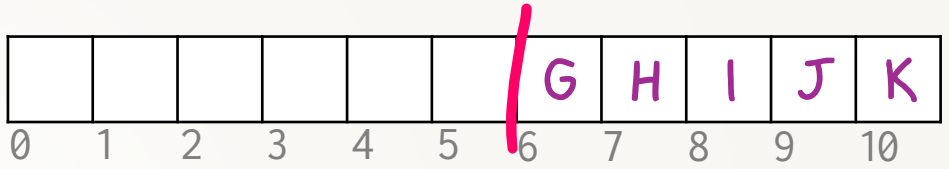
A heap with n items has height $\lceil \log_2 n \rceil$.

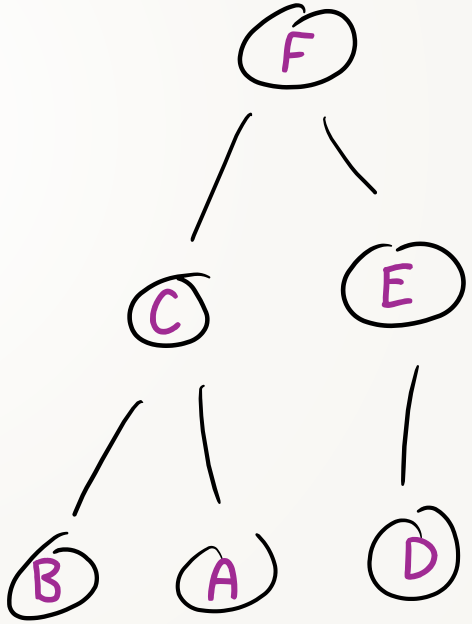
We'll use a heap to store the items that have yet to be placed into the "sorted" part of the array.

Conveniently, we can use the array *itself* to store the heap.

Let the children of $x[i]$ be $x[2i+1]$ and $x[2i+2]$.

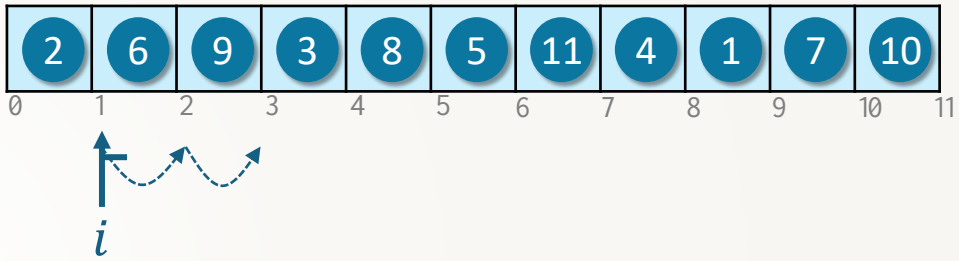


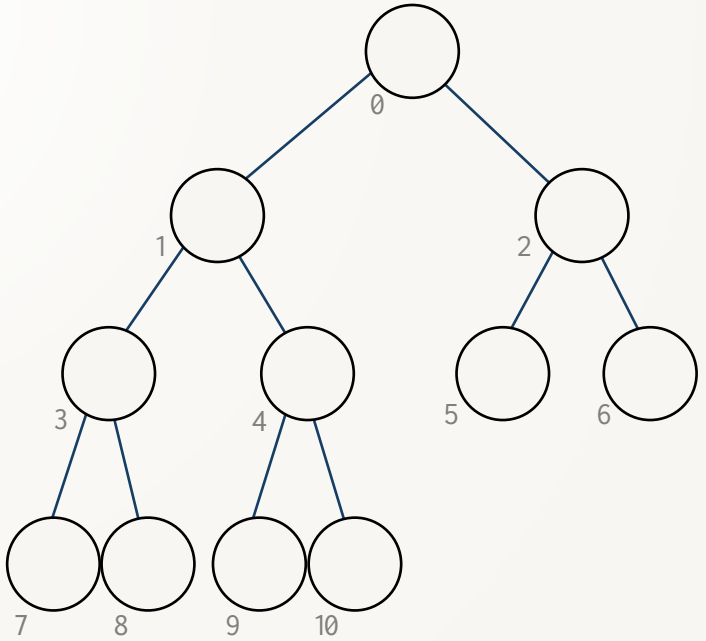




```
def heapsort(x):
  n = len(x)

  for i in 1..n-1:
    # assert x[0:i] is a heap
    add x[i] to heap and re-heapify

  # assert x[0:n] is a heap

  for i in n..1:
    # assert x[i:n] has largest n-i
    # assert x[0:i] is a heap
    swap x[0] with x[i-1]
    re-heapify x[0:i-1]
```

It's handy to visualize the data both as an array and as a tree, simultaneously. But internally there is just a single array of size n.

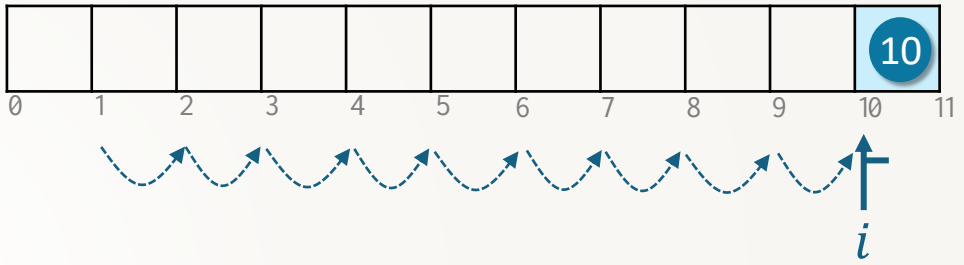


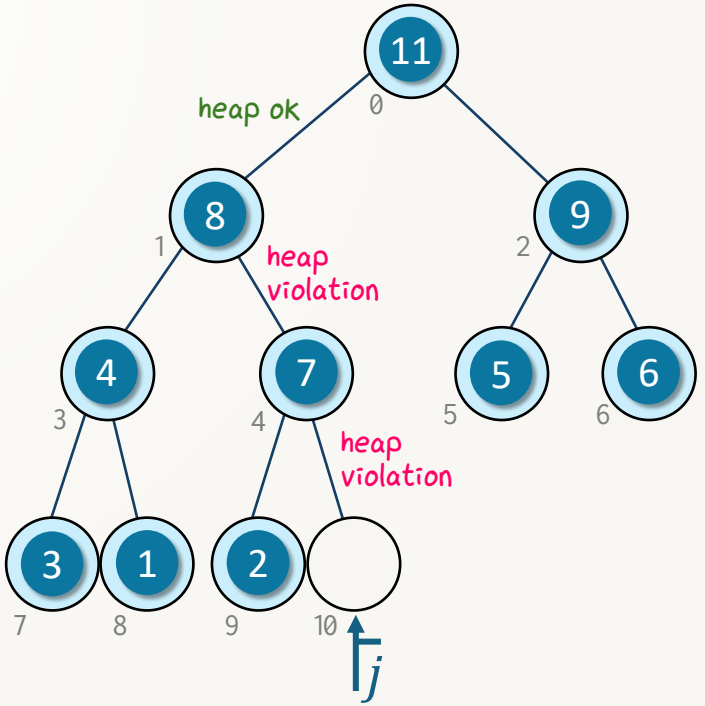


```
def heapsort(x):
  n = len(x)

  for i in 1..n-1:
    # assert x[0:i] is a heap
    add x[i] to heap and re-heapify

  # assert x[0:n] is a heap

  for i in n..1:
    # assert x[i:n] has largest n-i
    # assert x[0:i] is a heap
    swap x[0] with x[i-1]
    re-heapify x[0:i-1]
```

```
# Re-heapify by bubbling up from i

j = i

while j > 0 and x[j] > x[parent(j)]:
  swap x[j] with x[parent(j)]
  j = parent(j)
```

It's handy to visualize the data both as an array and as a tree, simultaneously. But internally there is just a single array of size n.

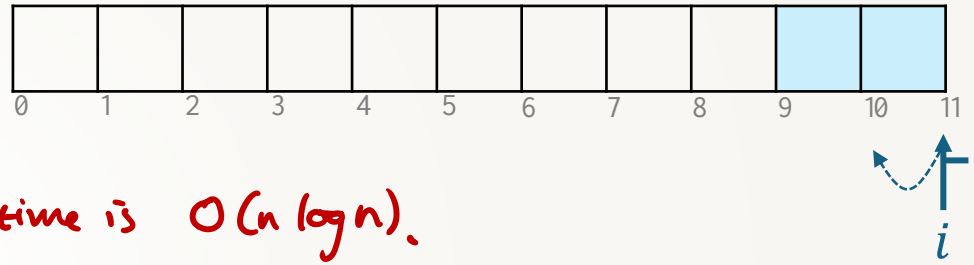




```
def heapsort(x):
  n = len(x)

  for i in 1..n-1:
    # assert x[0:i] is a heap
    add x[i] to heap and re-heapify

  # assert x[0:n] is a heap

  for i in n..1:
    # assert x[i:n] has largest n-i
    # assert x[0:i] is a heap
    swap x[0] with x[i-1]
    re-heapify x[0:i-1]
```

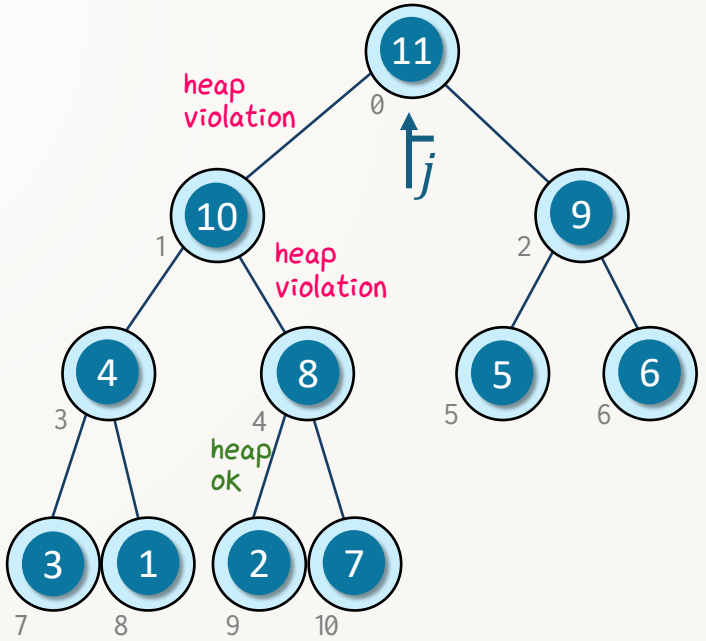
Total running time:
Each phase of the algorithm iterates over all n items in the array;
the re-heapify operations are $O(\text{heap height})$ i.e. $O(\log n)$.
Thus total running time is $O(n \log n)$.

It's handy to visualize the data both as an array and as a tree, simultaneously. But internally there is just a single array of size n.




```
# Re-heapify by bubbling up from i

j = i

while j > 0 and x[j] > x[parent(j)]:
  swap x[j] with x[parent(j)]
  j = parent(j)
```

```
# Re-heapify by bubbling down from 0

j = 0

while x[j] < max(x[child1(j)], x[child2(j)]):
  swap x[j] with larger child
  j = larger child
```

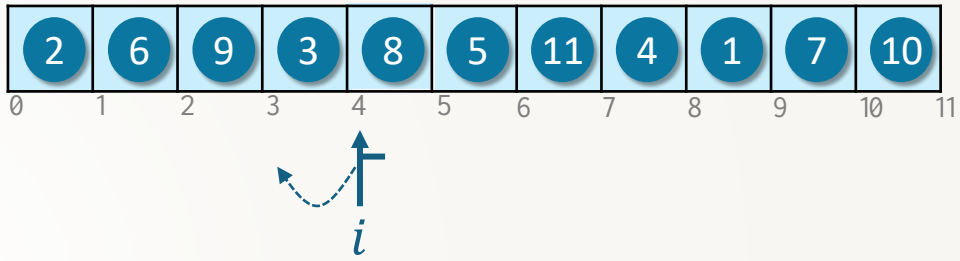


In fact, we can create the initial heap in $O(n)$.

```
def heapsort(x):
  n = len(x)
  for i in 1..n-1:
    # assert x[0:i] is a heap
    add x[i] to heap and re-
  # assert x[0:n] is a heap
  for i in n..1:
    # assert x[i:n] has largest n-i
    # assert x[0:i] is a heap
    swap x[0] with x[i-1]
    re-heapify x[0:i-1]
```

```
# Faster heap creation
for i in (⌊n/2⌋-1)..0:
  # assert trees rooted at (i+1)..n are heaps
  re-heapify the tree rooted at x[i]
  by bubbling down
```

Big idea: at the depths with lots of items (d ≈ h so 2^d big) the work needed is small (d ≈ h so h-d small).



This procedure does a bubble-down from every non-leaf position in the tree.

A bubble-down from depth d takes time h-d

Total cost = $\sum_{d=0}^{h} 2^d (h-d)$

- 2^d: # of nodes at level d
- (h-d): work for each of them

$\leq 2 \times 2^h$ [see printed notes for the maths]

$= 2n$