Sorting

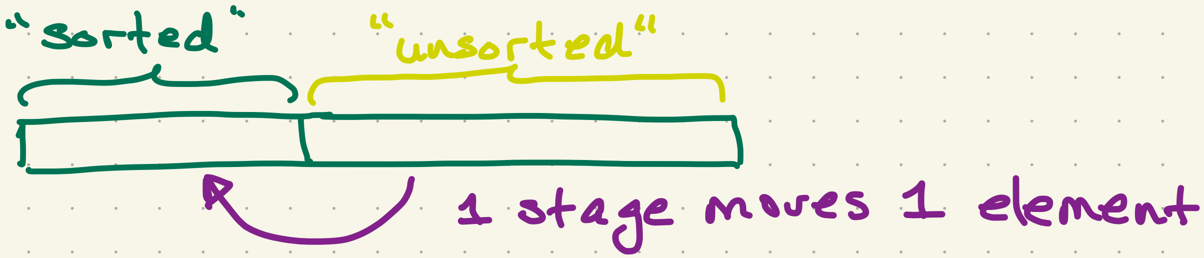
Sorting

- re-arranging elements of a sequence S s.t. $s_0 \leq s_1 \leq s_2 \leq \dots \leq s_{n-1}$

- We will look at 5 sorting algorithms:
  - 3 iterative
  - 2 recursive

The iterative algorithms:

- maintain a partition: "unsorted part" & "sorted part"
- Sort a sequence of n elements in n-1 stages
- at each stage, move 1 element from the unsorted part to the sorted part:

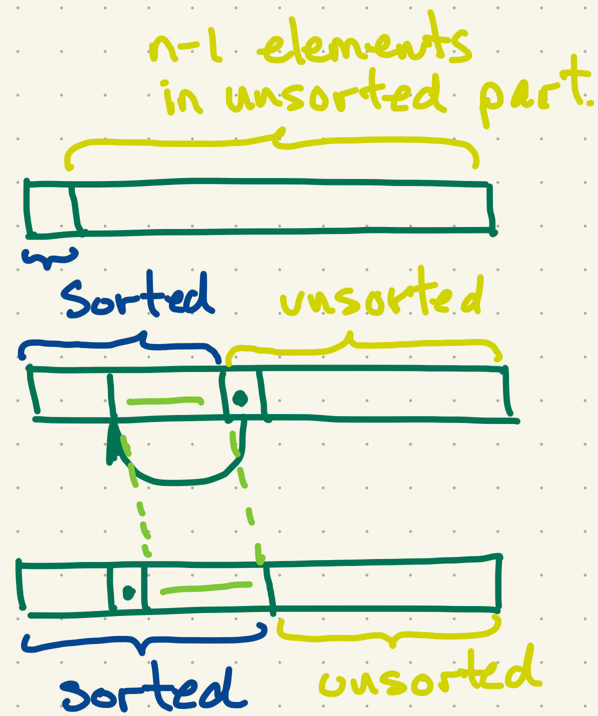



```
Sort(A){
    · initialize
    · repeat n-1 times
        move 1 element from unsorted to sorted part
}
```

- the algorithms differ in how they:
  - select an element to remove from the unsorted part
  - insert it into the sorted part

Insertion Sort

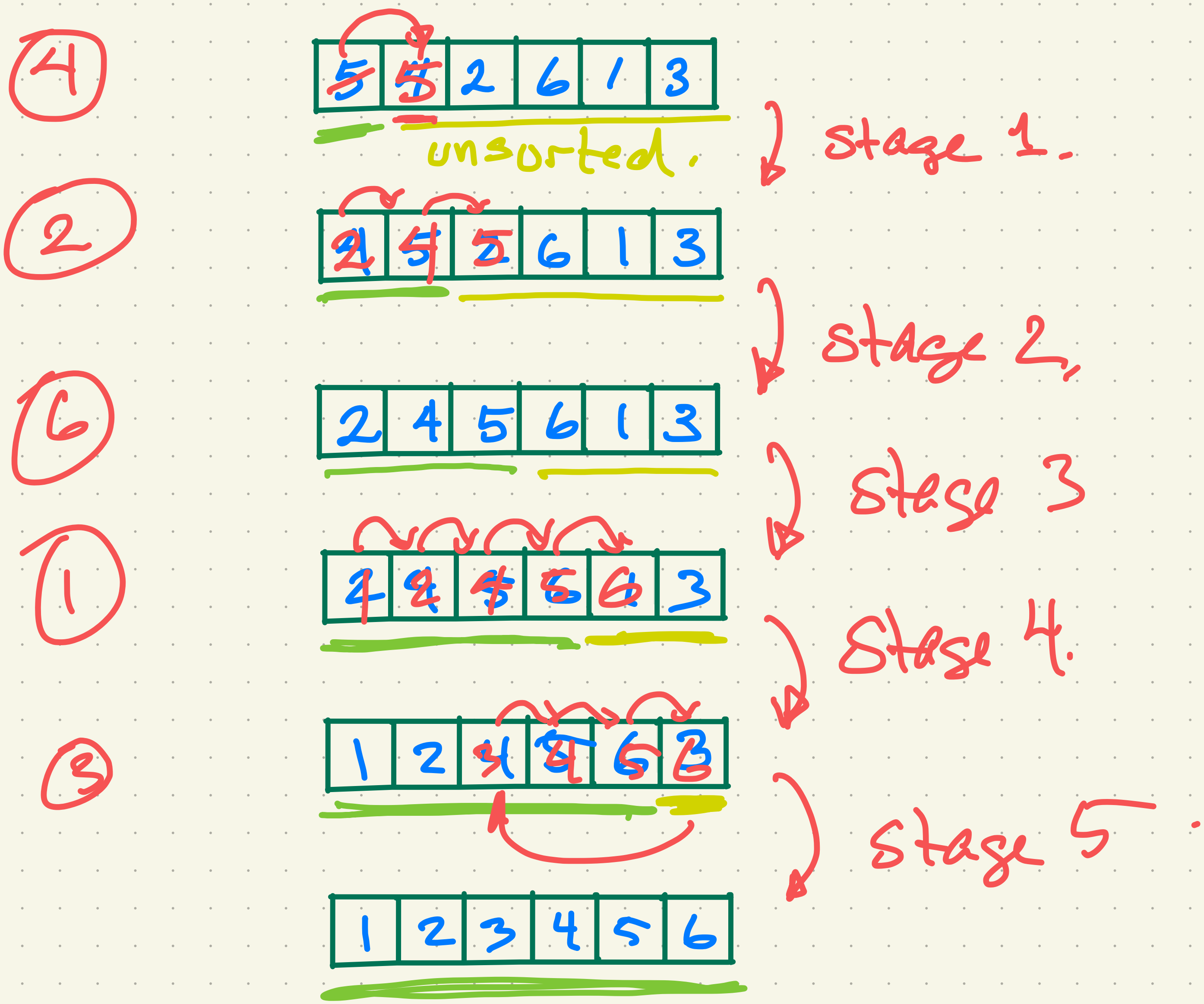
- intially: sorted part is just A[0]
- repeat n-1 times:
  - remove the first element from the unsorted part
  - insert it into the sorted part (shifting elements to the right as needed)




```
insertion_sort(A)
    for(i = 1 to n-1){
        pivot = A[i] // first element in unsorted part
        j = i - 1
        while(j≥0 AND A[j] > pivot){
            A[j+1] = A[j] // shift jth
            j = j-1
        }
        A[j+1] = pivot // move pivot into position.
    }
```

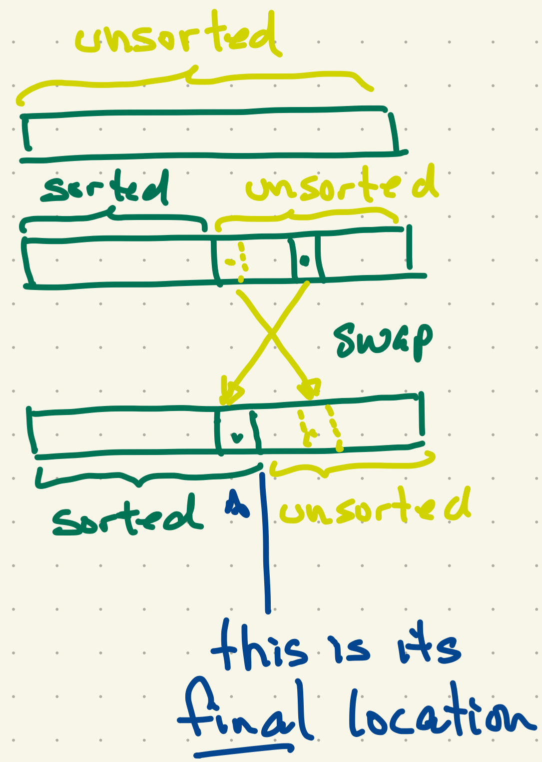
Shift all elements in sorted part that are larger than pivot 1 "to the right".

Insertion Sort Example



Selection Sort

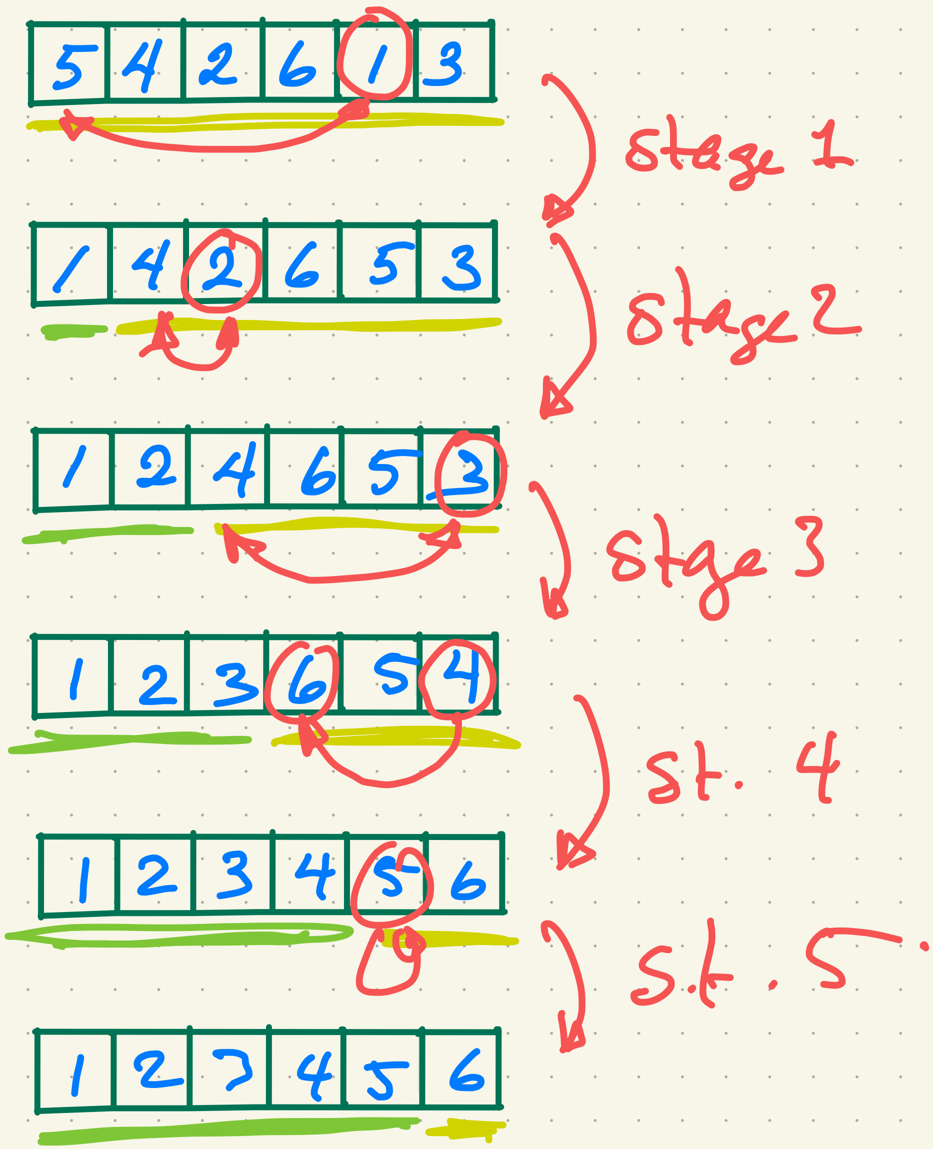
- initially, sorted part empty
- repeat n-1 times
  - find the smallest element in the unsorted part
  - move it to the first position which becomes the new last position of sorted part.




```
selection_sort(A){
  for( i=1 to n-1 ){
    j= i-1   // j is index of min found so far.
    k=i
    while( k<n){
      if( A[k] <A[j]) j=k;
      k=k+1
    }
    swap A[i-1] and A[j]
  }
}
```

find min element in unsorted

Selection Sort Example

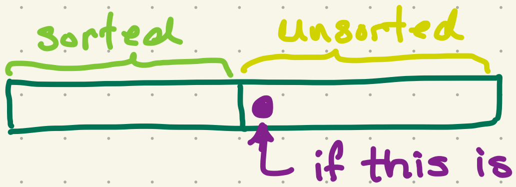


~~Selection Sort~~ Heapsort

- initially, sorted part empty
- make unsorted part into a heap ← takes $O(n)$ time
- repeat n-1 times
  - find the smallest element in the unsorted part } heap extract takes $\log(n)$ time (vs. $\Theta(n)$ for the scan in Selection Sort).
  - move it to the first position which becomes the new last position of sorted part.

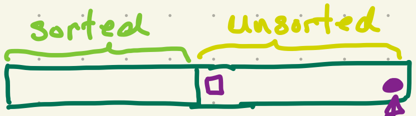
Consider the organization of array contents:

①



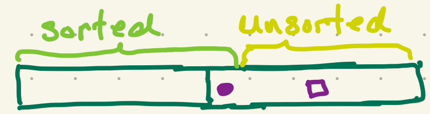
if this is the root of the heap, then it is also the smallest element in the usorted part, so is in its correct final position. To use this arrangement, the root of the heap keeps moving, so we have lots of shifting to do.

2



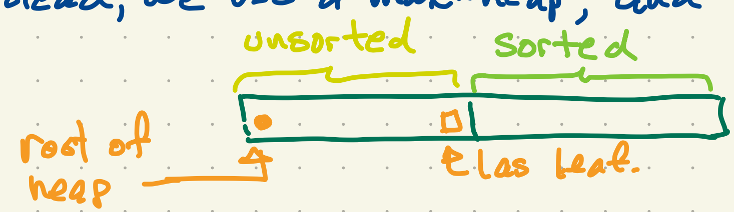
If this is the root of the heap, then everything works:

- we extract •; move the last leaf □ to the root + do a percolate-down; store • where □ <u>was</u>, which is now free, and is the correct final location for •; after which we have:



<u>But</u>: we must re-code our heap implementation s.t. the root is at A[n-1], with the result that the indexing is now less 'intuitive'.

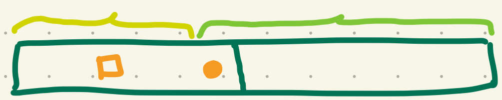
3 Instead, we use a max-heap, and this arrangement:



Now: · the heap root is at A[0]

· heap extraction removes •, moves □ to A[0], freeing up the spot where • belongs.

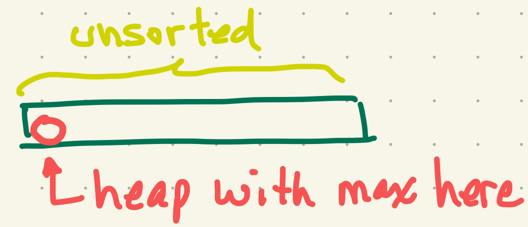
Leaving us:

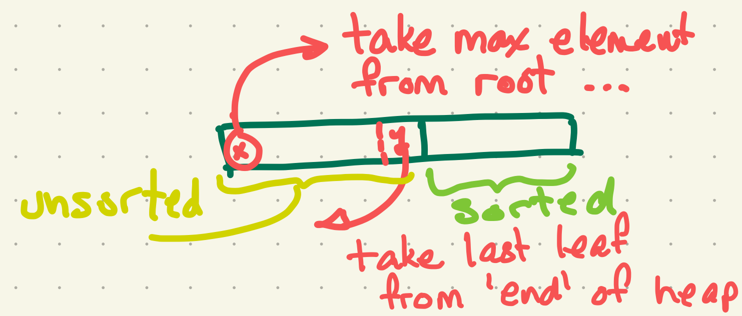


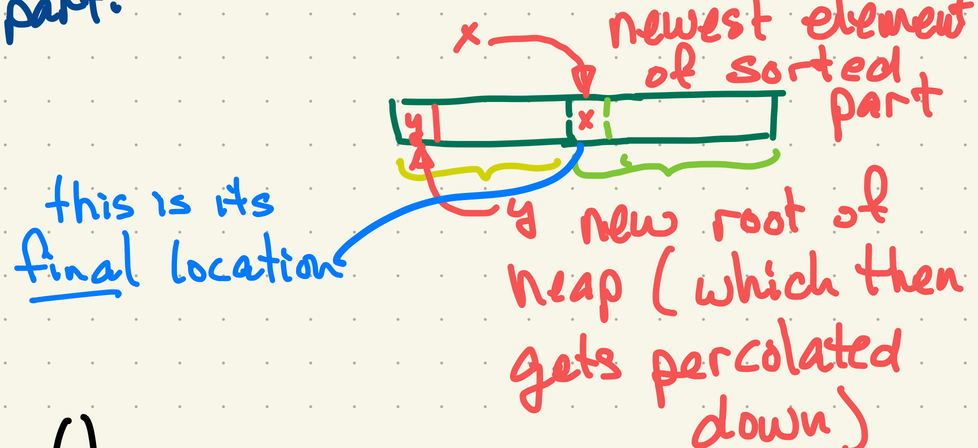
Re-coding a min Heap into a max heap is just replacing $<$ with $>$ and vice versa.

~~Selection Sort~~ Heapsort

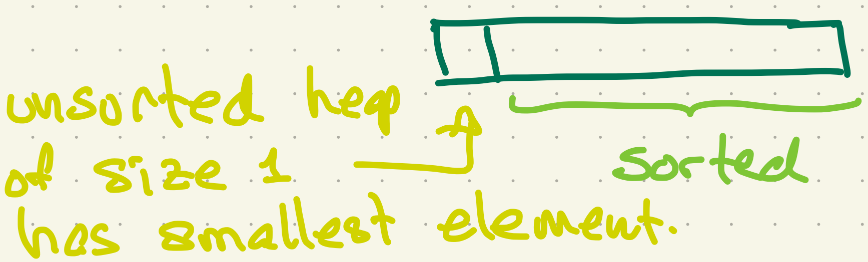
- initially, sorted part empty
- make unsorted part into a max heap
- repeat n-1 times
  - find the ~~smallest~~ largest element in the unsorted part
  - move it to the ~~first~~ last position which becomes the new ~~last~~ first position of sorted part.








```
heapsort(A){
  buildMaxheap(A)
  for( i=1 to n-1 ){
      A[n-i] = extractMax()
  }
}
```

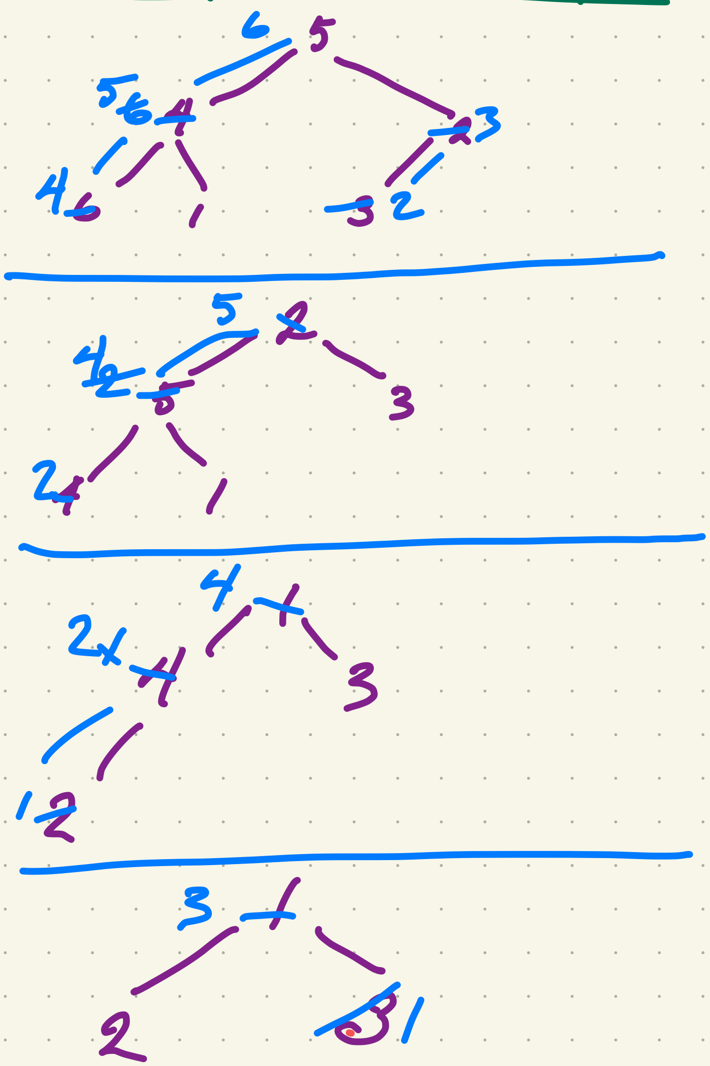


Heapsort with in-line percolate-down

```
heapsort(A){
    makeMaxHeap(A)
    for(i=1 to n-1){
        swap A[0] and A[n-i]   // move last leaf to root
                               // and old root to where
                               // last leaf was.
        size ← n-i+1  // size of heap = size of unsorted part
        j ← 0
        while(2j+1 < size){
            child ← 2j+1
            if(2j+2 < size AND A[2j+2] < A[2j+1]){
                child ← 2j+2
            }
            if(A[child] < A[j]){
                swap A[child] and A[j]
                j ← child
            } else
                j ← size / terminate the while.
        }
    }
}
```

percolate down

Heapsort Example:



5	4	2	6	1	3

make into a heap

6	5	3	4	1	2

stage 1

2	5	3	4	1	6

5	4	3	2	1	6

stage 2

1	4	3	2	5	6

4	2	3	1	5	6

stage 3

1	2	3	4	5	6

3	2	1	4	5	6

stage 4

1	2	3	4	5	6

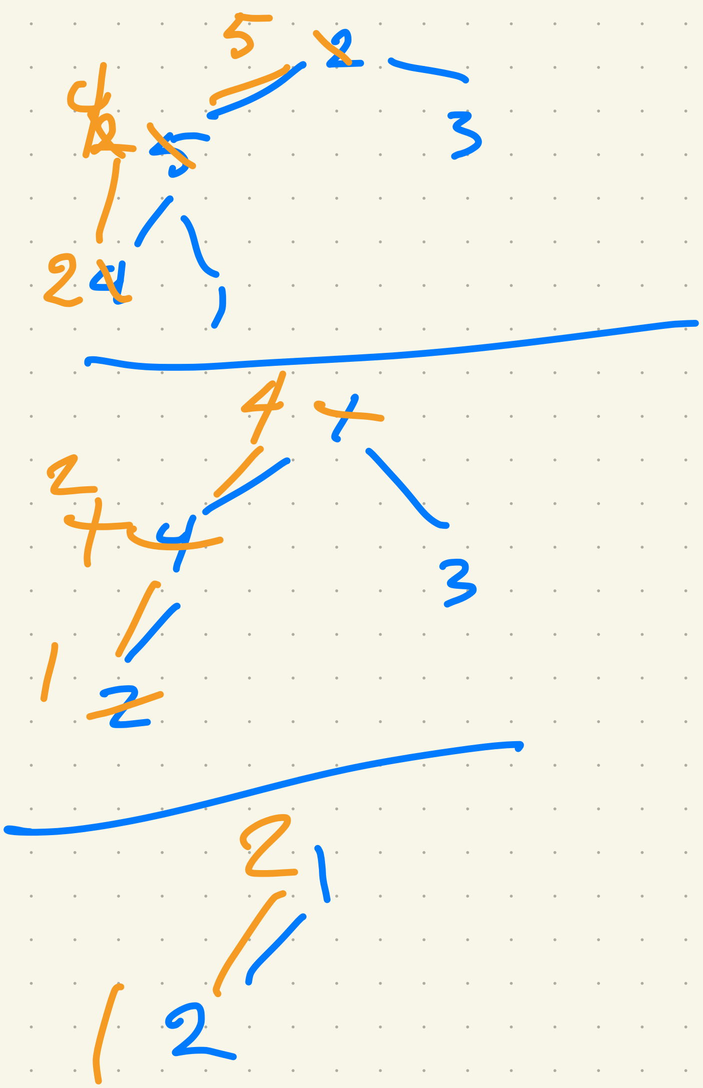
2	1	3	4	5	6

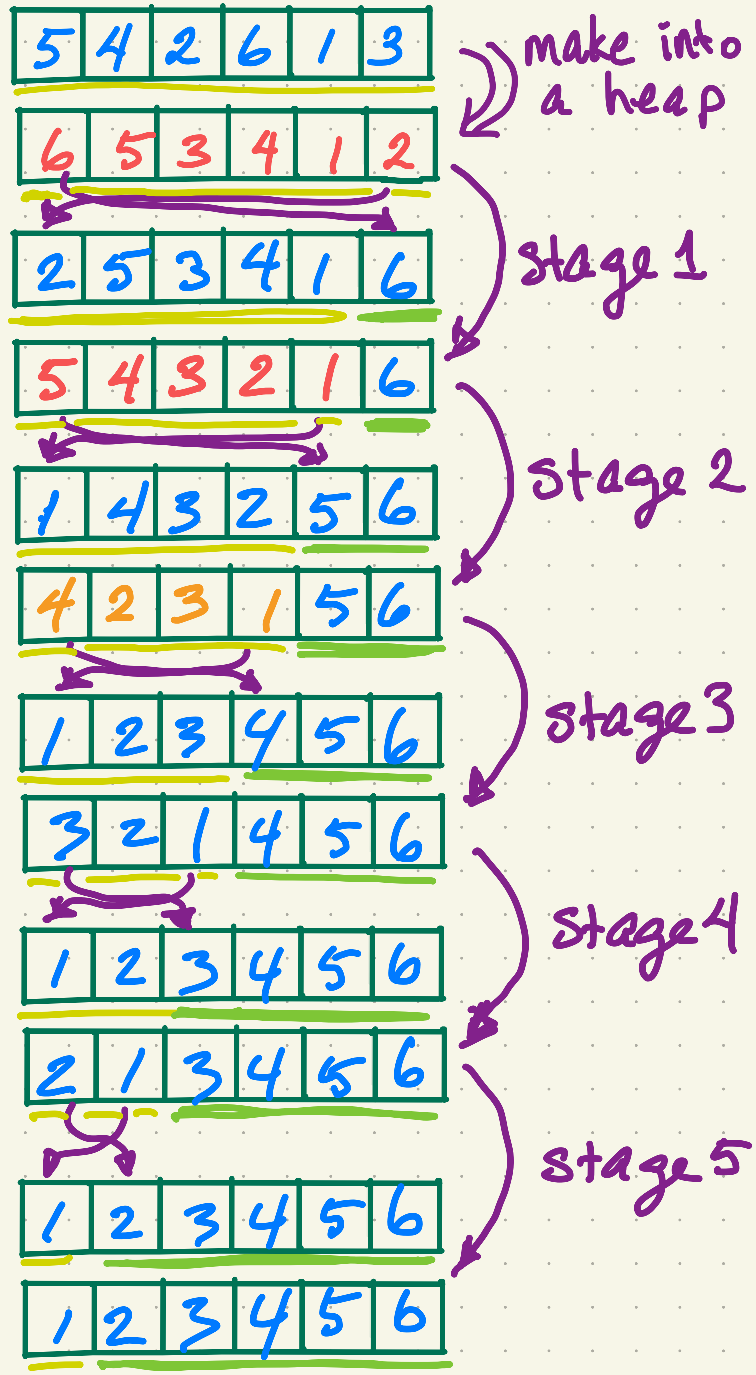
stage 5

1	2				

1	2	3	4	5	6

Heapsort Example:





Time Complexity of Iterative Sorting Algorithms

- each algorithm does exactly $n-1$ stages
- the work done at the i^{th} stage varies with the algorithm (& input).
- we take # of item comparisons as a measure of work/time. *

Selection Sort - exactly $n-i$ comparisons to find min. element in unsorted part

Insertion Sort - between 1 and i comparisons to find location for pivot

Heap Sort: - between 1 and $2\log_2(n-i+1)$ comparisons for percolate-down

* Number of comparisons

- We must verify #comparisons (or some constant times #comparisons) is an upper bound on work done by each algorithm.

- # of assignments (& swaps) also matters in actual run time.

Selection Sort

On input of size n, # of comparisons is always (regardless of input):

$$\sum_{i=1}^{n-1} (n-i) = \sum_{i=1}^{n-1} i$$

$$= S(n-1)$$

$$= \frac{(n-1)(n)}{2}$$

$$= \frac{n^2-n}{2}$$

$$= \Theta(n^2)$$

Insertion Sort – Worst Case

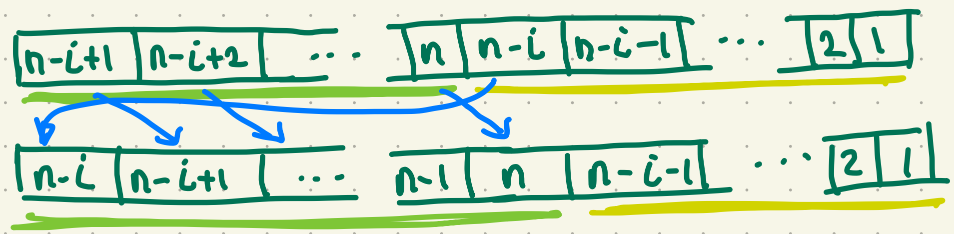
Upper Bound: # comparisons $\leq \sum_{i=1}^{n-1} i = \frac{n^2-n}{2} = O(n^2)$.

Lower Bound: Worst case: initial sequence is in reverse order.

Eg.

n	$n-1$	$n-2$	$\cdots\cdots$	1

In the i^{th} stage we have



This takes i comparisons, because the sorted part is of size i.

So, # comparisons $\geq \sum_{i=1}^{n-1} i = \Omega(n^2)$

So, Insertion Sort Worst Case is $\Theta(n^2)$

Insertion Sort Best Case

Best case: initial sequence is fully ordered.

Then: In each stage exactly 1 comparison is made.

So: # comparisons $= n-1 = \Theta(n)$.

Heapsort Worst Case

Upper Bound:

$$\# \text{comparisons} \leq \sum_{i=1}^{n-1} 2 \log_2(n-i+1)$$

$$= 2 \sum_{i=1}^{n-1} \log_2(i+1)$$

$$\leq 2 \sum_{i=1}^{n-1} \log_2 n$$

$$\leq 2n \log_2 n$$

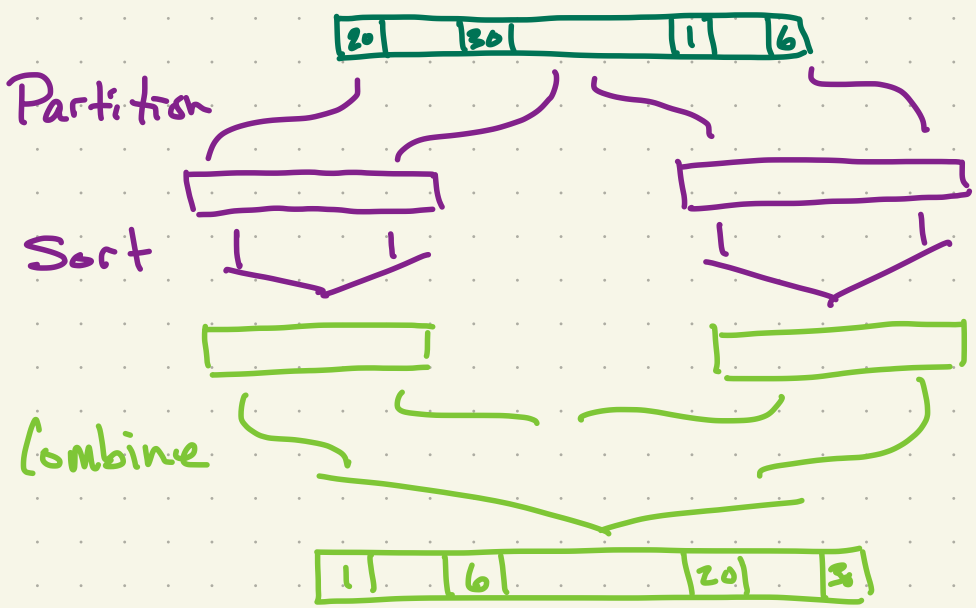
$$= O(n \log n)$$

Lower Bound?

Best Case? (What input would lead to <u>no</u> movement during percolate-down? What if we exclude this case?)

Recursive Divide & Conquer Sorting

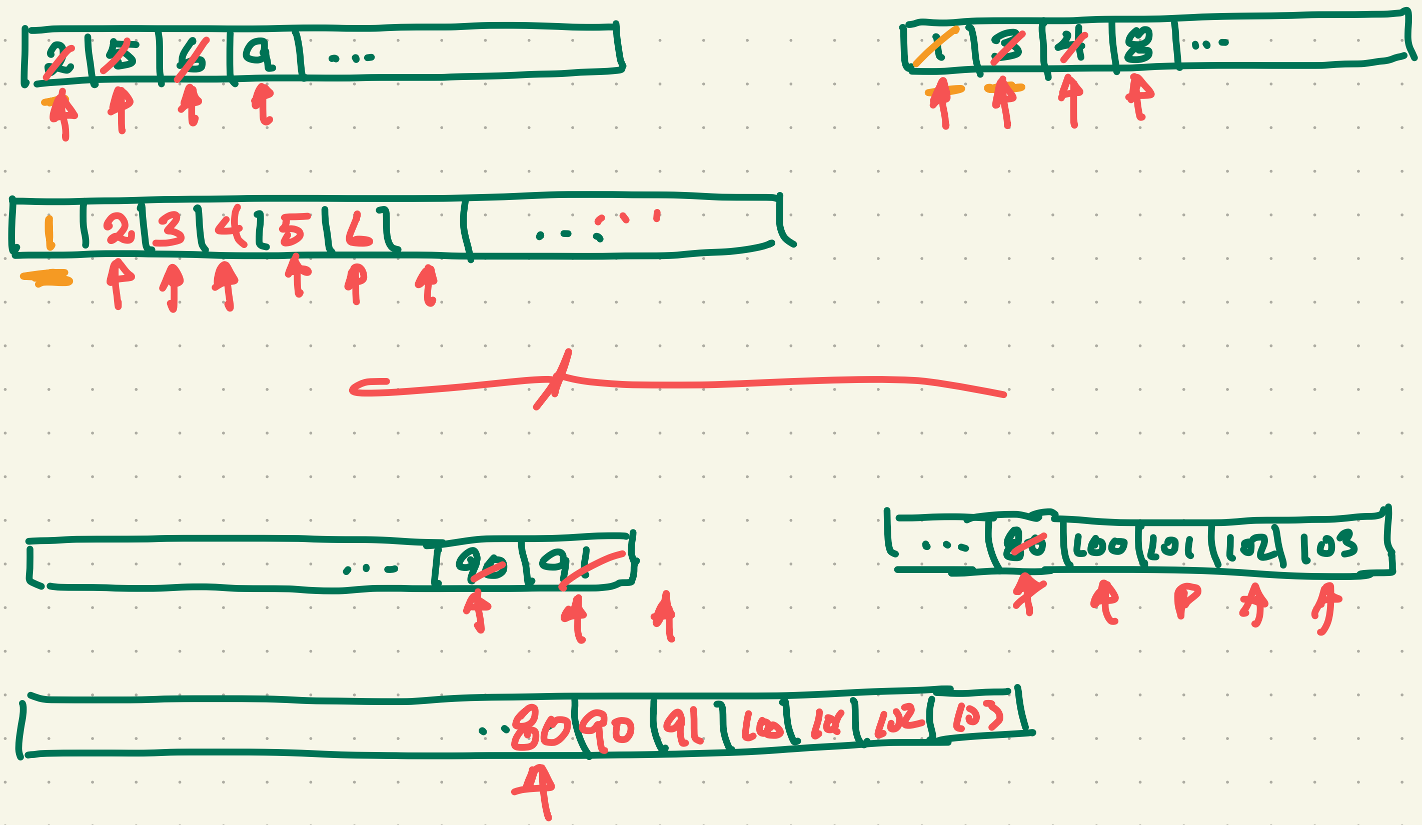
- Partition the sequence A into two parts A_1, A_2
- Recursively sort each of A_1 and A_2
- Combine the sorted versions of A_1 and A_2 to obtain a sorted version of A



- The algorithms differ in how they choose the partition, and how they combine the sorted parts

Mergesort:

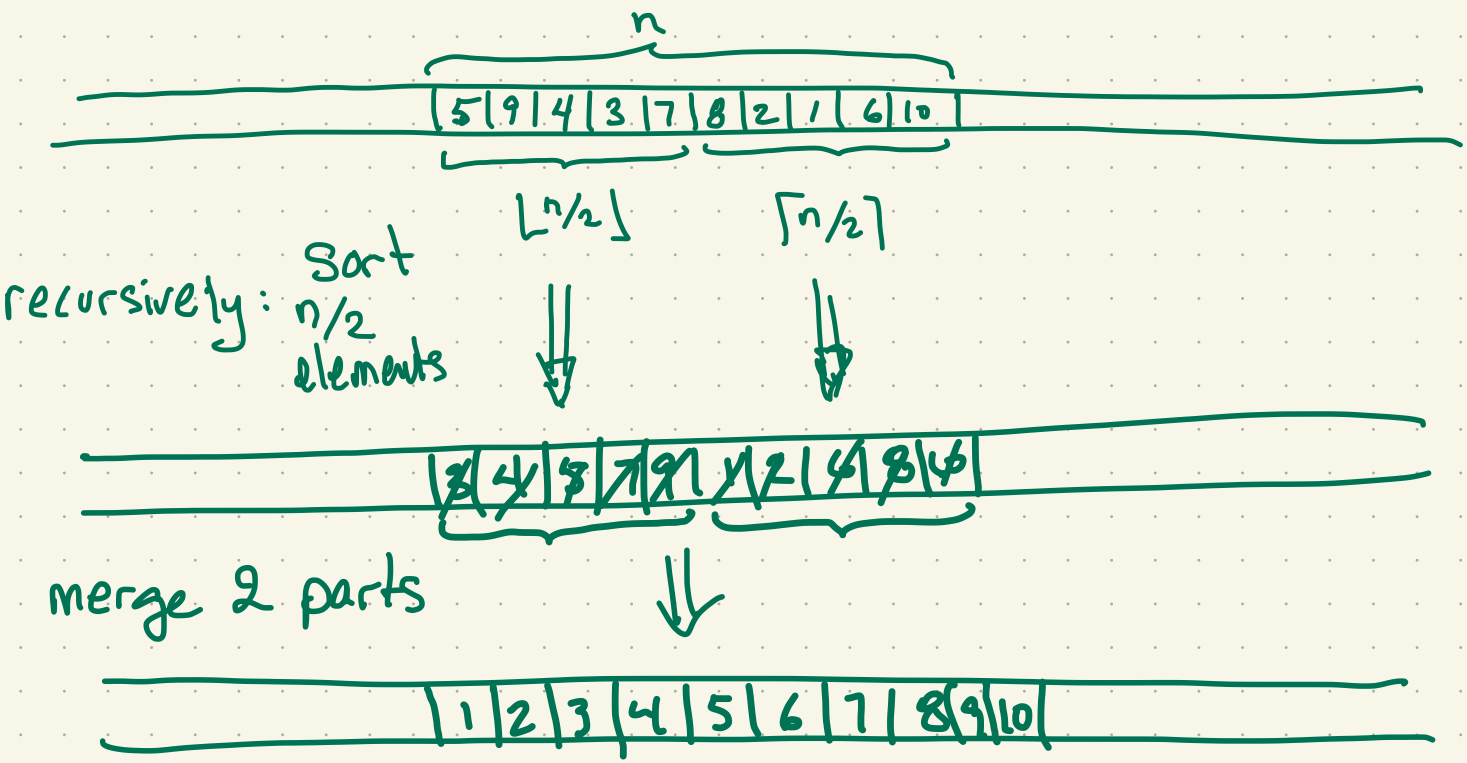
- Uses the fact that merging two sorted lists is easy



- Takes $O(n)$ time, where n is the total size

Mergesort:

- partition: first half & second half.
- combine: merge the parts



- works with linked-list or array implementations
- in array implementations, uses $\Theta(n)$ extra space

Mergesort

```
mergesort(A, lo, hi){
    if(lo < hi){ // there are ≥ 2 items, so work to do
        mid ← ⌊(lo + hi)/2⌋
        mergesort(A, lo, mid)
        mergesort(A, mid+1, hi)
        merge(A, low, mid, hi)
    }
}
```

Merge for Merge Sort

```
merge(A, lo, mid, hi){
    l ← lo
    r ← mid+1
    n ← lo
    while( l < mid  AND r < hi){
        if( A[l] < A[r]){
            B[n] ← A[l]
            l++
        }else{
            B[n] ← A[r]
            r++
        }
        n++
    }
    while(l < mid){
        B[n] ← A[l]
        l++ ; n++
    }
    while(r < hi){
        B[n] ← A[r]
        r++ ; n++
    }
}*
```

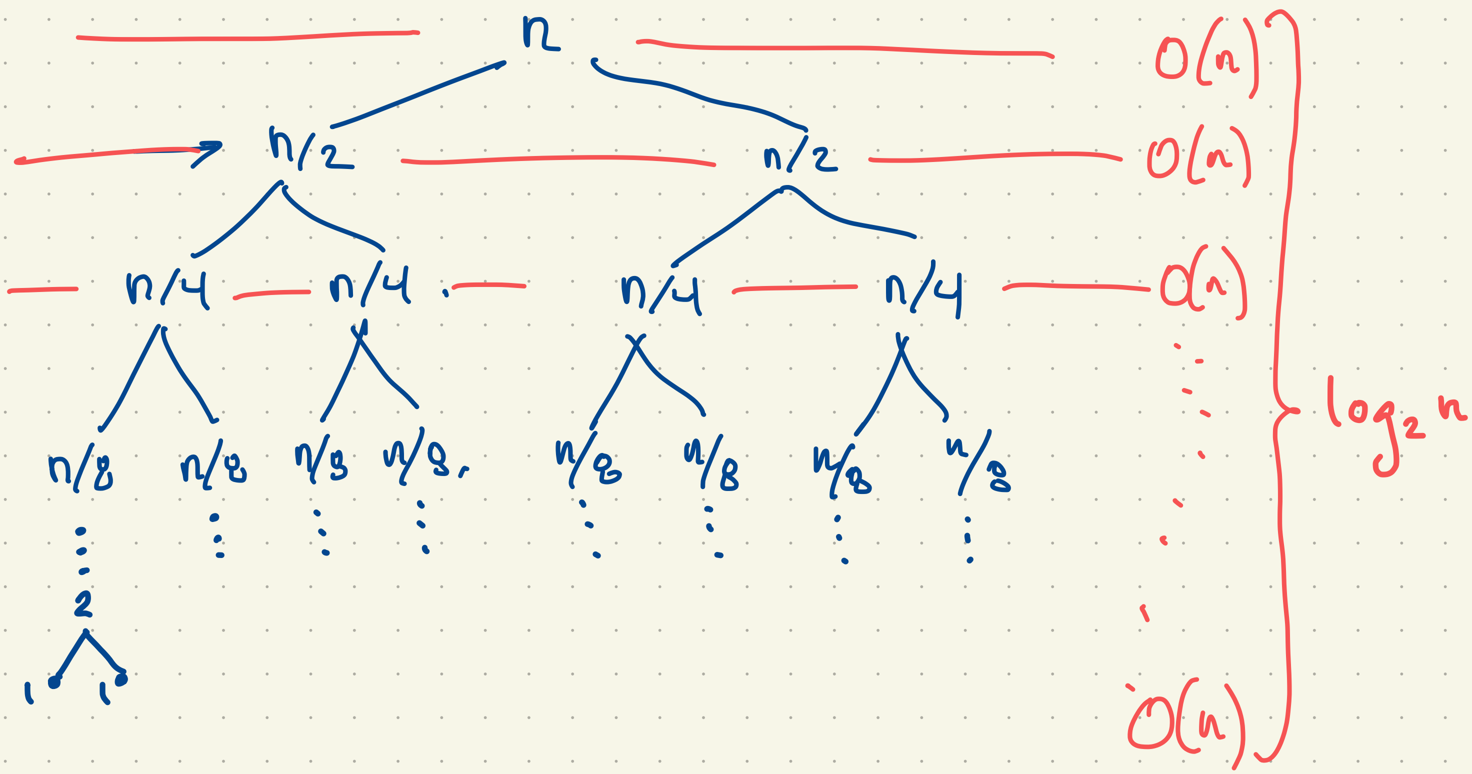
After *, the sorted sequence is in B[lo] ... B[hi].

'lazy' solution:
copy B[lo]..B[hi] to A.

'fast' solution:
swap A, B:

temp ← A
A ← B
B ← temp

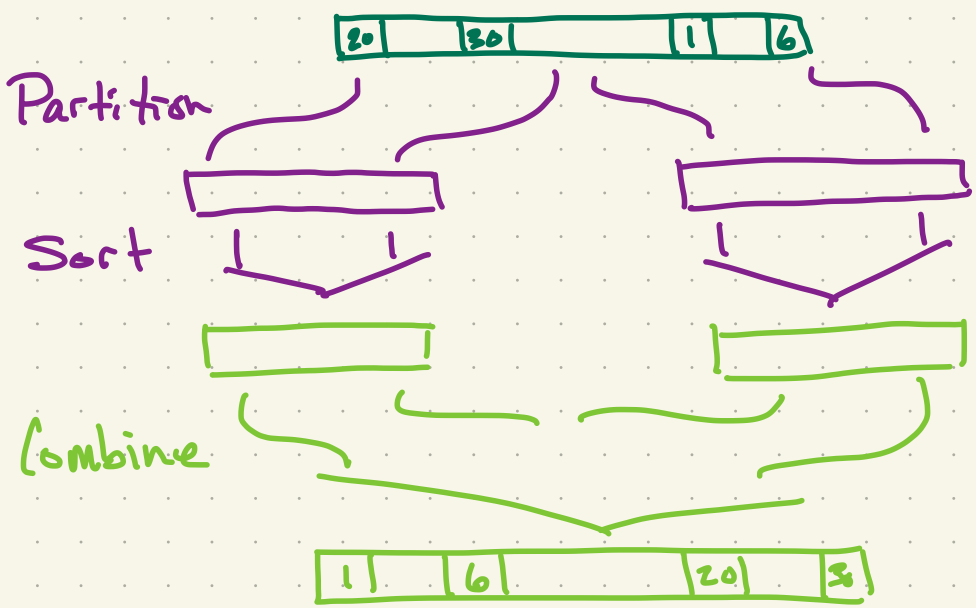
Time Complexity of Merge Sort: via tree of recursive calls



- If n is a power of 2, the tree of recursive calls is a perfect binary tree with n leaves, and height $\log_2 n$.
- At depth i there are 2^i calls to merge, each to merge two lists of size $n/2^{i+1}$ into one of size $n/2^i$.
- Total work at depth i is $2^i \cdot O(n/2^i) = O(n\, 2^i/2^i) = O(n)$.
- Total work is #depths $\cdot\, O(n) = \log n \cdot O(n) = O(n \log n)$.

Recursive Divide & Conquer Sorting

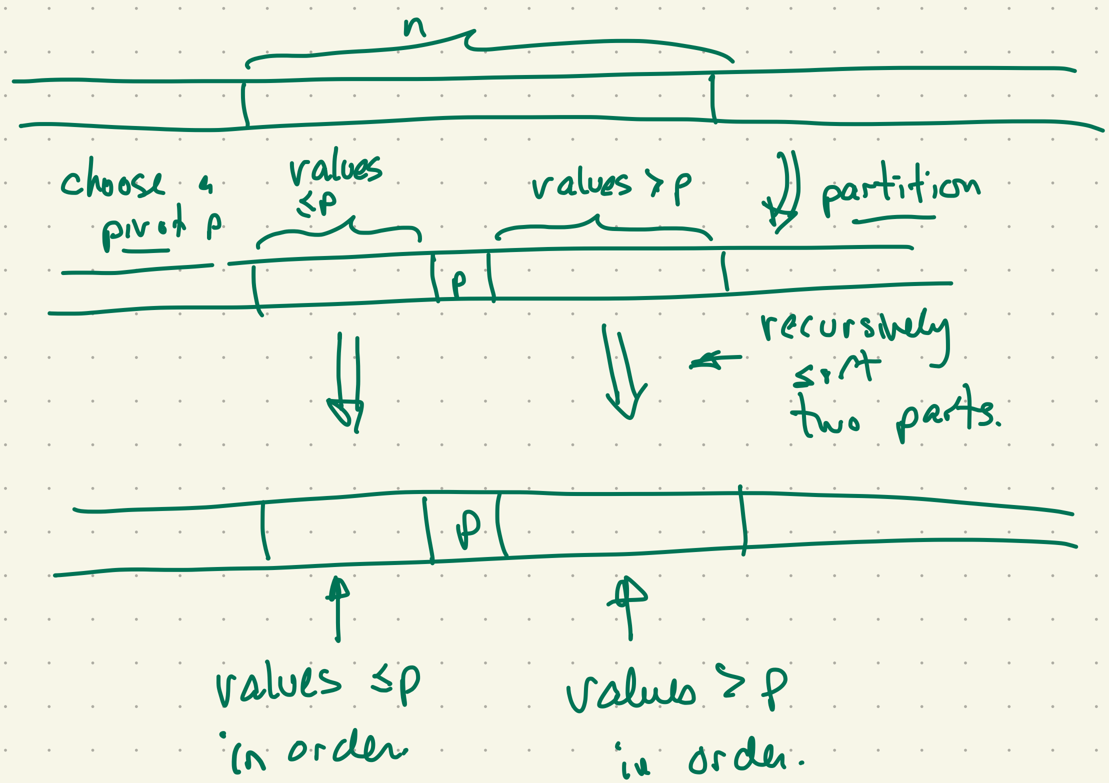
- Partition the sequence A into two parts A_1, A_2
- Recursively sort each of A_1 and A_2
- Combine the sorted versions of A_1 and A_2 to obtain a sorted version of A



- The algorithms differ in how they choose the partition, and how they combine the sorted parts

Quicksort

- Uses a pivot p to partition sequence into "small" and "large" elements: small elements $< p <$ large elements
- combining sorted versions is trivial



- choosing pivots is key to performance.

Quicksort

```
quicksort(A, lo, hi){
    if(lo < hi){ // there are ≥2 items
        pivotposition ← partition(A, lo, hi) // partition
        quicksort(A, lo, pivotposition - 1)
        quicksort(A, pivotposition + 1, hi)
    }
}
```

Quicksort is correct as long as every call to partition() returns and leaves the variables satisfying the following:

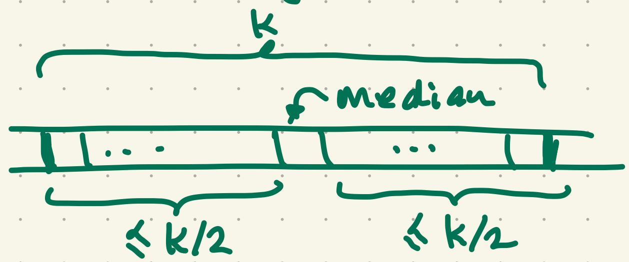
1. $lo \le pivotposition \le hi$
2. for every i, j with $lo \le i \le pivotposition \le j \le hi$

$$A[i] \le A[pivotposition] \le A[hi]$$

However efficiency relies critically on choice of pivot.

Ex: Perfect Pivots

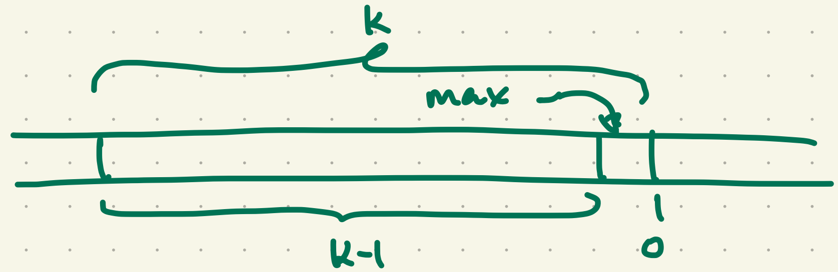
- Suppose all elements are distinct, and the pivot is chosen to be the median element in A[lo]... A[hi].
- Then, every call to Quicksort on sequence of size $k \geq 2$ makes two recursive calls on sequences of size $\leq k/2$:



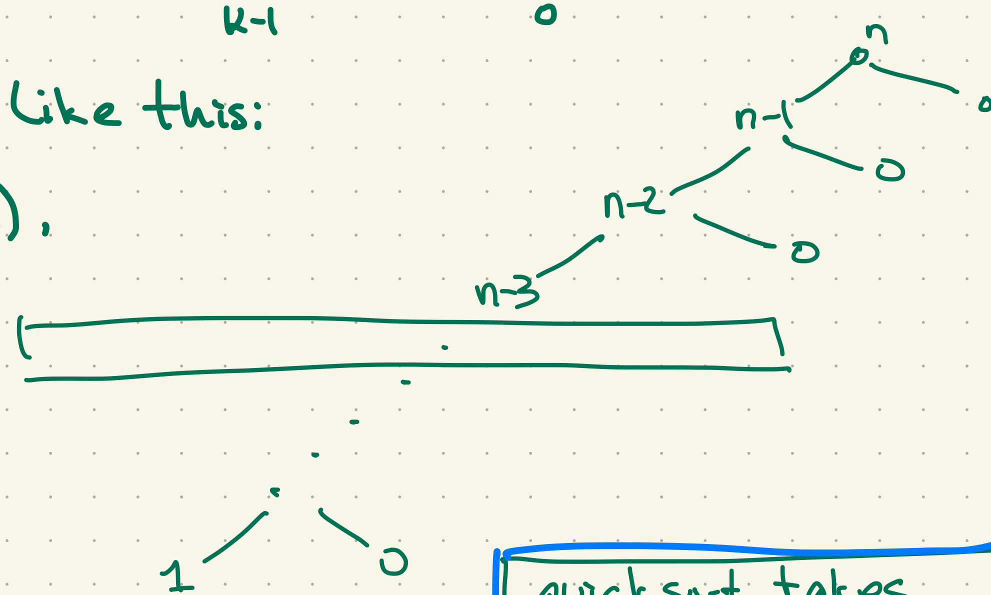
- By essentially the same argument as used for MergeSort, this gives us running time of $\log(n) \cdot f(n)$, where $f(n)$ is the time to run partition on a sequence of size n.
- Assuming $O(n)$ time for partition, this would give us $O(n \log n)$ time for Quicksort.
- But: finding medians is too slow in practice.
- Optional exercise: Can the median be found in $O(n)$ time?

Ex: Worst Case Pivots.

- Suppose all elements are distinct, and the max is always chosen as pivot.
- Then every call to Quicksort on a sequence of $k \geq 2$ elements makes one recursive call on a sequence of size $k-1$, and one on a sequence of size 0:



- The recursion tree looks like this:
- This tree is of height $\Theta(n)$, giving us a running time of $\Theta(n) \cdot f(n)$, or $\Theta(n^2)$ assuming $\Theta(n)$ for partition.

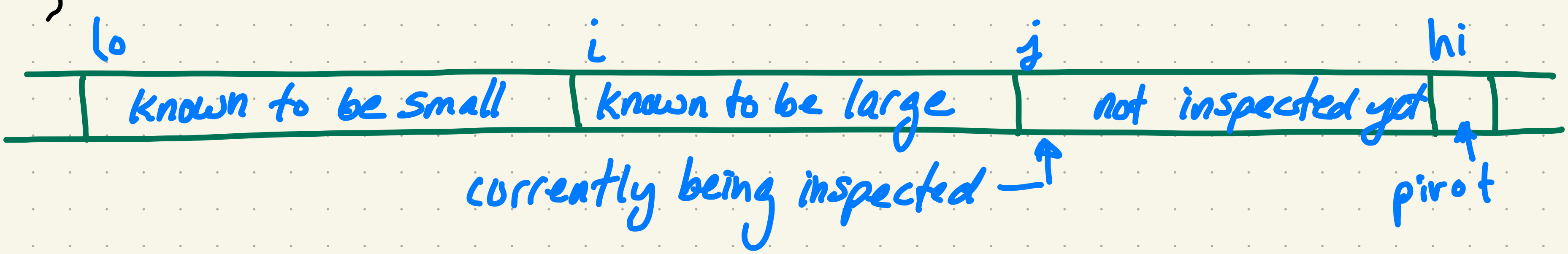


quicksort takes time $\Theta(n^2)$ in the worst case.

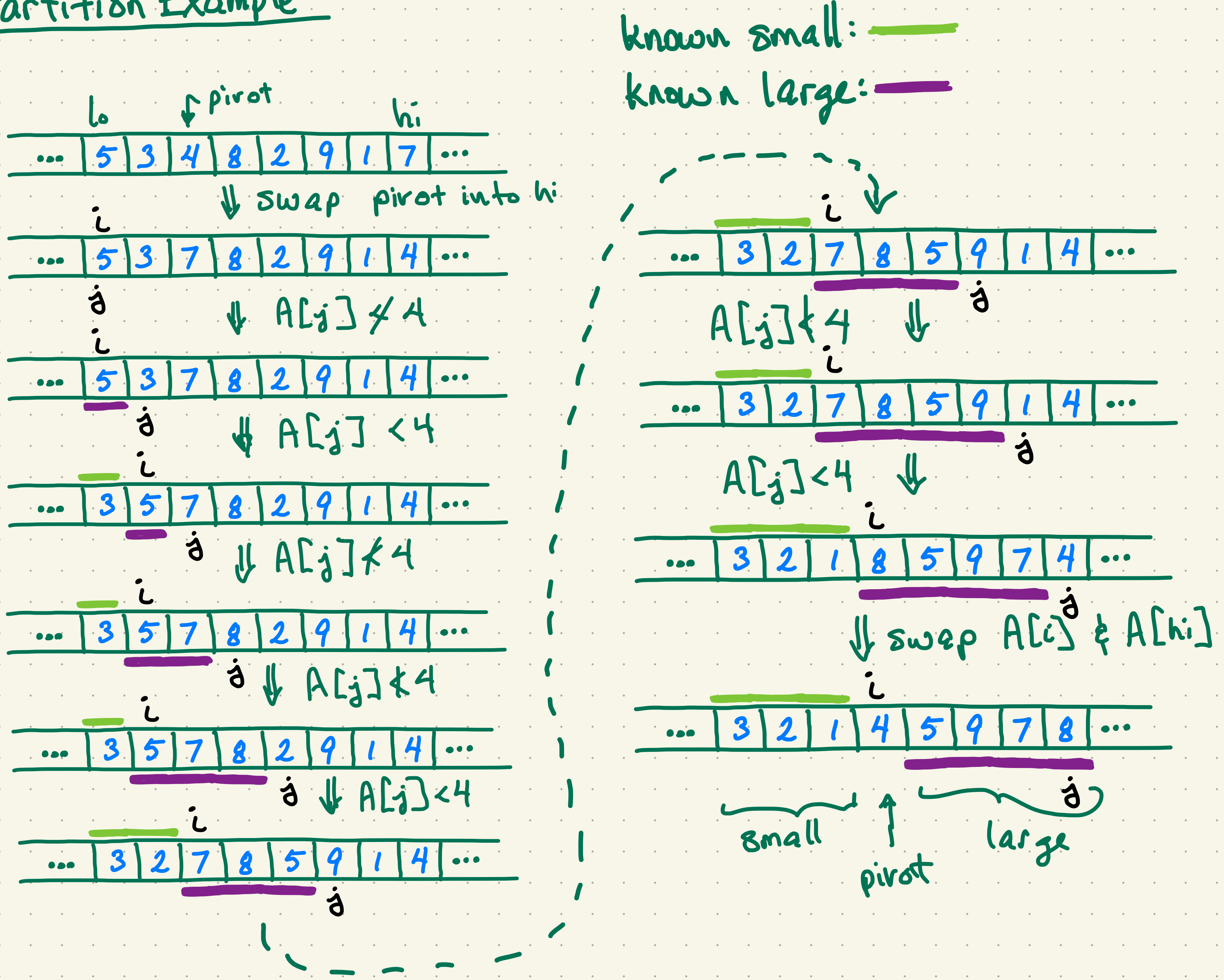
Partition

- Partition must choose a pivot p. and efficiently re-arrange elements

```
partition(A, lo, hi){
   pivotindex ← choosePivot(A, lo, hi) // choose pivot
   swap A[pivotindex] and A[hi] // move pivot out of the way.
   p ← A[hi] // p is the pivot
   i ← lo // known "small" values will be at indices < i
   for (j = lo ; j < hi ; j++){ // "already inspected" values will
                                // be at indices < j
       if (A[j] ≤ p){            // if we are inspecting a "small"
           swap A[i] and A[j]    // swap it with first "non small"
              i ← i+1            // increase size of "smalls" part.
       }
   }
   swap A[i] and A[hi] // move pivot where it belongs.
   return i // this is pivot position
}
```



Partition Example



- <u>Partition</u>

  Time complexity of partition is $\Theta(n) + g(n)$, where $g(n)$ is time taken by choosePivot.

- Q: How can we choose "good" pivots "fast"?

- Quicksort is the most-used sorting algorithm in practise, so there must be a way.

- But: - what qualifies as "fast"?

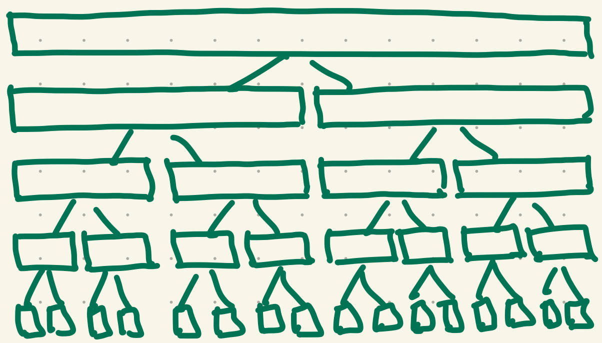
  probably a very small constant

  - what qualifies as "good"?

  (given that it must be fast)

Consider:

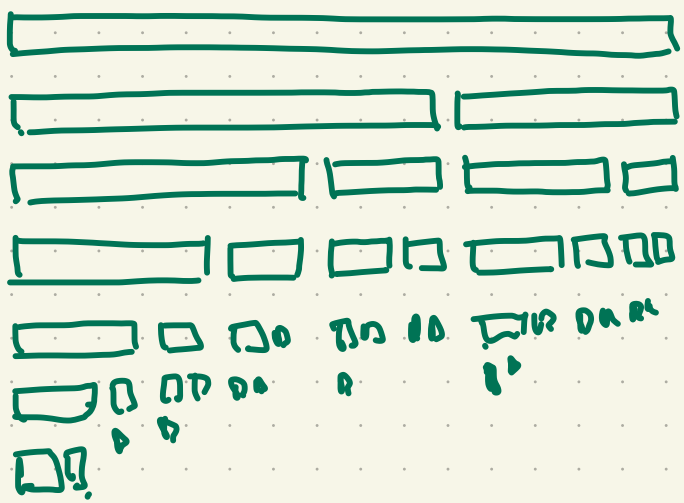
- A small number of bad pivots:



vs

makes a small difference in height.

- Perfect pivots are not needed for $O(n \log n)$ time.



Eg. if pivots are all better than $\frac{1}{3} : \frac{2}{3}$, then depth is $\log_{3/2} n$

So we still get $O(n \log n)$.

Some "simple" choosePivot options

- A[hi] – fast, but performs badly on many inputs.
- median – perfect pivots, but too slow to compute
- random: · If pivots are chosen uniformly at random, then Quicksort runs in time $O(n \log n)$ with probability $1 - \frac{1}{2^n}$ – ie almost always.
  - But: good random numbers are not fast to make.
- median{ A[lo], A[hi], A[(hi+lo)/2] }
  - fast
  - not very easy to come up with a "very bad" input.

Complexity of Quicksort

- Depends critically on how pivots are chosen
- Choosing "perfect" pivots is too slow for a practical sorting algorithm
- Fortunately, choosing pivots that are "good enough" for <u>most inputs</u> can be done fast.
- Quicksort – with practical pivot choice strategies – is $\Theta(n^2)$, but is often described as "like $\Theta(n \log n)$ in practice".

In practice

- There are settings where Merge sort & Insertion Sort are preferred
- In most settings, the preferred algorithm is Quicksort
- For small sets, Selection Sort is faster
- Often, this variant (or similar) is faster:

```
quicksort(A, lo, hi){
    if(lo < hi){ // there are ≥2 items
        if(lo + 15 > hi){ // less than 15 items
            selectionSort(A, lo, hi)
        }
        else{
            pivotposition ← partition(A, lo, hi) // partition
            quicksort(A, lo, pivotposition - 1)
            quicksort(A, pivotposition + 1, hi)
        }
    }
}
```

End