Algorithms

Correctness and Running time

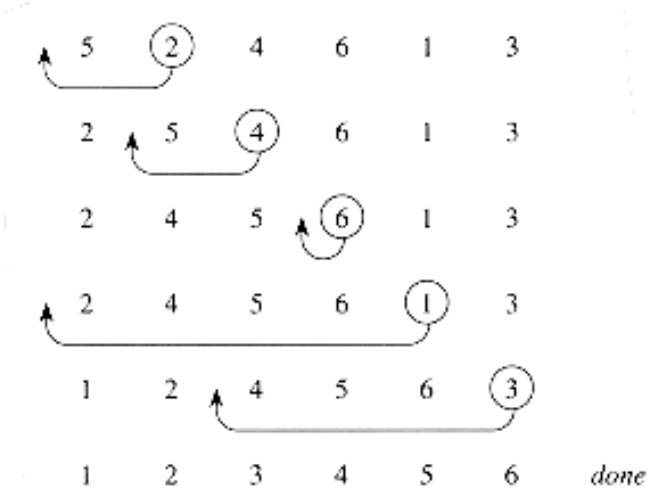
In assignments and exams, whenever you present an algorithm always argue its correctness.

If you are claiming some running time, you need to prove that too.

You don't need to do the above if you are using some algorithm described in class.

Exercise

- Provide the pseudocode of a sorting algorithm that runs in $\Theta\ (n^2)$



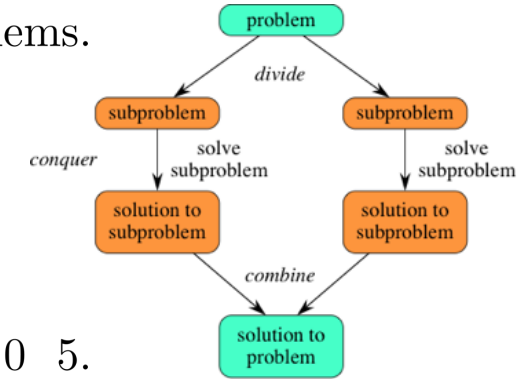
INSERTION-SORT (*A*)
1 **for** *j* <- 2 **to** *length*[*A*]
2 **do** *key* <- *A*[*j*]
Now: Insert *A*[*j*] into the sorted sequence *A*[1 . . *j* - 1].
4 *i* <- *j* - 1
5 **while** *i* > 0 and *A*[*i*] > *key*
6 ***do*** *A*[*i* + 1] <- *A*[*i*]
7 *i* <- *i* - 1
8 *A*[*i* + 1] <- *key*

Divide and Conquer

Idea. Decompose the problem into easier subproblems.

Solve the subproblems.

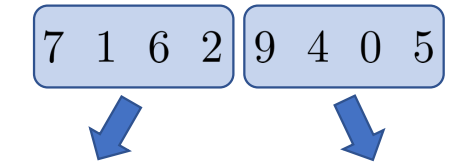
Combine the solutions to the subproblems
to obtain a solution to the original problem.



Example. Sorting an array. e.g. 7 1 6 2 9 4 0 5.

We want to split the array into two halves,
sort them recursively and combine.

Option 1.



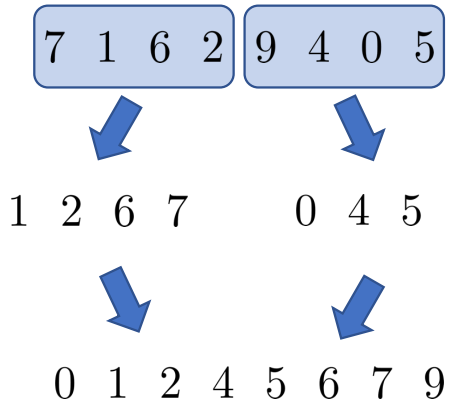
Split into two halves. Easy!

1 2 6 7 0 4 5 9 *Sort recursively.*

0 1 2 4 5 6 7 9 *Combine. Requires work!*

Mergesort

Option 1.



Split into two haves. Easy!

Sort recursively.

Combine. Requires work!

Non-trivial task. Merging two sorted arrays into a single sorted array.

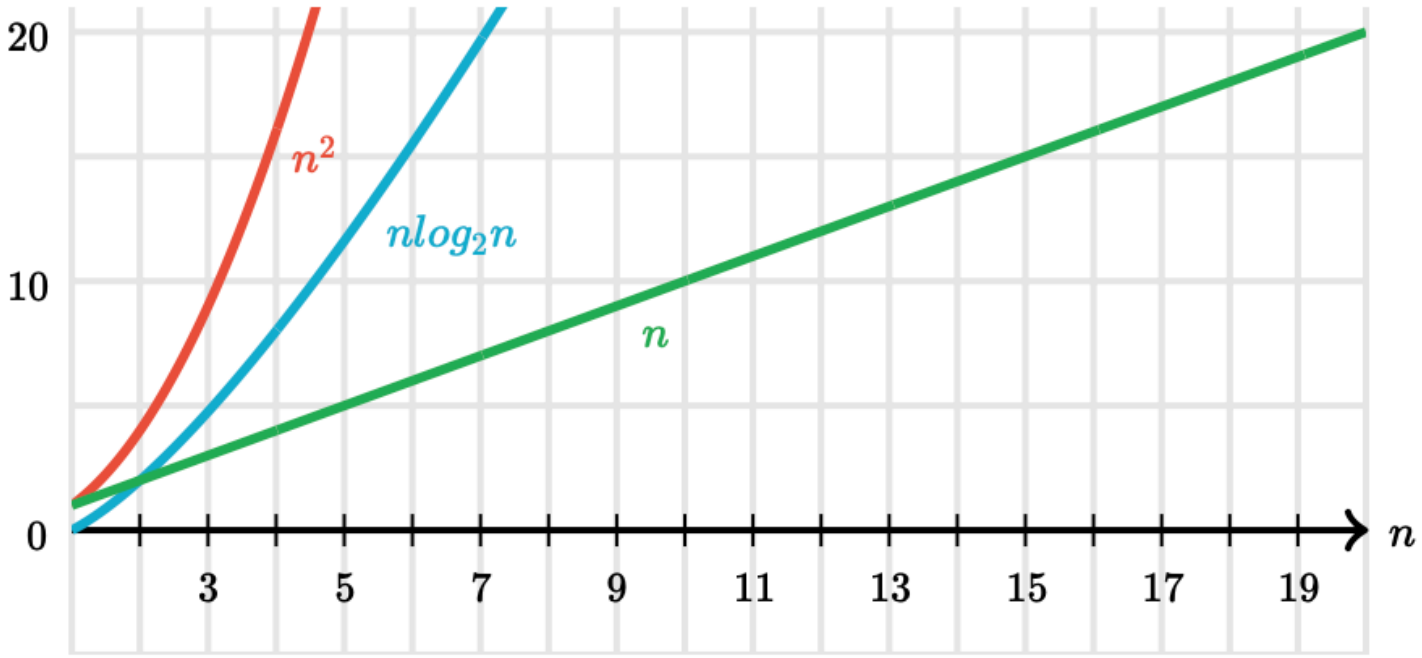
This takes time proportional to the total size of the two arrays.

Let $T(n)$ the time taken by Mergesort on an input of size n.

Recurrence relation. $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$, $T(1) \leq c$. *c : some constant*

$$\Longrightarrow T(n) = O(n \log n).$$

Visualization

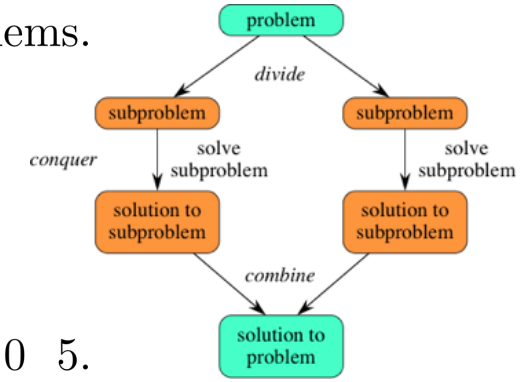


Divide and Conquer

Idea. Decompose the problem into easier subproblems.

Solve the subproblems.

Combine the solutions to the subproblems
to obtain a solution to the original problem.



Example. Sorting an array. e.g. 7 1 6 2 9 4 0 5.

We want to split the array into two halves,
sort them recursively and combine.

Option 2.

7 1 6 2 9 4 0 5

1 2 4 0 7 6 9 5

0 1 2 4 5 6 7 9

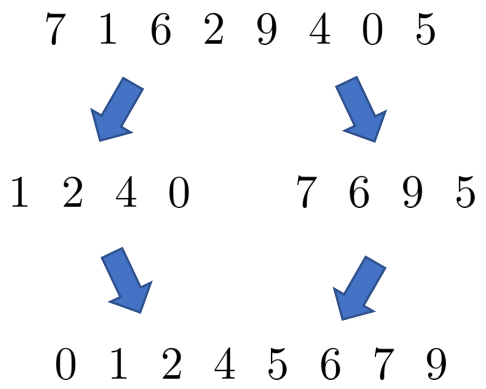
Split s.t. the no.s in the first subproblem are smaller than all the no.s in the second subproblem. Non-trivial!

Sort recursively.

Combine by concatenating. Easy!

Quicksort

Option 2.



Split s.t. the no.s in the first subproblem are smaller than all the no.s in the second subproblem. Non-trivial!

Sort recursively.

Combine by concatenating. Easy!

Idea. Suppose that the elements are distinct. *arbitrary but consistent tie breaking*

Suppose also that the median of n numbers can be found in $O(n)$ time.

Then, splitting the problem into two halves takes $O(n)$ time.

Let $T(n)$ the time taken by Quicksort on an input of size n.

Recurrence relation. $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$, $T(1) \leq c$. *c : some constant*

$$\implies T(n) = O(n \log n).$$

Quicksort

How do we find the median of an array of n elements in $O(n)$ time?

Do we really need the median?

Suppose that we find in $O(n)$ time, an element x such that at most $3n/4$ elements are $\leq x$ and at most $3n/4$ are $\geq x$.

We can call such an element an *approximate median*.

What if we use the approximate median instead of the actual median?

Then the two subproblems may not have the same size!

Does it matter? Not really!

The recurrence relation for the running time is of the form:

$$T(n) \leq T(n_1) + T(n_2) + cn,\ T(1) \leq c$$

where $n_1, n_2 \leq 3n/4$, $n_1 + n_2 = n$ and c is some constant.

Quicksort

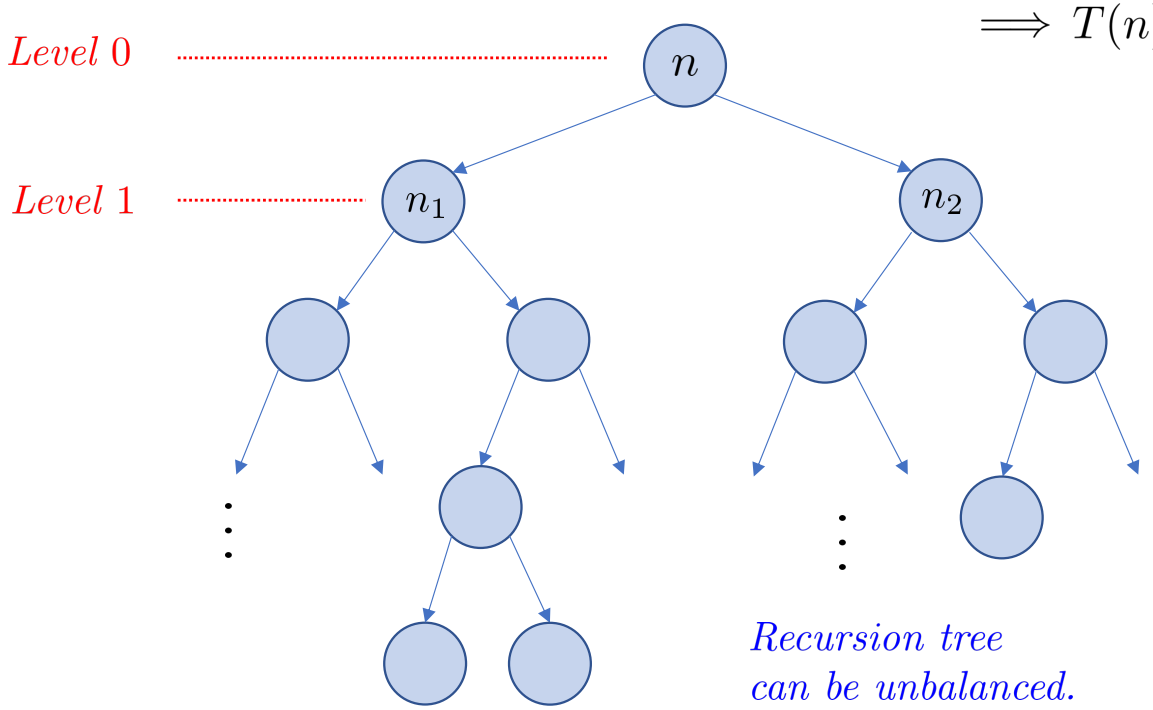
Recurrence relation. $T(n) \leq T(n_1) + T(n_2) + cn,\ T(1) \leq c$

where $n_1, n_2 \leq 3n/4$, $n_1 + n_2 = n$ and c is some constant.

All subproblems at level i have size at most $(3/4)^i\, n$ $\Longrightarrow \leq 1 + \log_{4/3} n$ *levels.*

The total size of all subproblems at level i is at most n $\Longrightarrow$ *$O(n)$ time per level.*

$\Longrightarrow T(n) = O(n \log n)$.



IDEA

- What happens if we choose the rightmost element in the array?

- Pros: Simple to code, fast to calculate
  Cons: If the data is sorted or nearly sorted, quick sort will degrade to $O(n^2)$
- If it is unlikely that the data will be sorted, and you are willing to accept $O(n^2)$ in the rare cases when the array is sorted then use the leftmost or rightmost element

Exercise

Median Finding

Ok, so it suffices to find an approximate median. How do we do that?

Idea. Pick a random element from the array.

Half of the elements in the array are approximate medians!

50% chance of being lucky!

Say we pick some element x at random from the array.

How do we know if it is actually an approximate median?

Simply check how many elements are $\leq x$ and how many $\geq x$. Takes $O(n)$ time.

What if we are unlucky again an again?

That's a possibility but on average two attempts suffice.

This is a *randomized algorithm* and the typical implementation of Quicksort.

The ***average*** running time of randomized Quicksort ***on any input*** is $O(n \log n)$.

Median Finding

We are dissatisfied! We want a deterministic algorithm!

Ok, we'll solve a more general problem.

Selection Problem.

Given an array A and a positive integer k, find the k^{th} smallest element in A.

The algorithm we will discuss is called **quickselect**.

Exercise: divide and conquer algorithm

- Given an array $A[]$ of integers, write an algorithm to find the maximum difference between any two elements such that the larger element appears after the smaller element. In other words, we need to find $\max(A[j] - A[i])$, where $A[j] > A[i]$ and $j >$ i.

- Example 1
  - Input: $A[] = [1, 4, 9, 5, 0, 3, 7]$,
  - Output: 8
  - Explanation: Maximum difference is between 9 and 1

- Complexity?
- Solution: https://www.enjoyalgorithms.com/blog/maximum-difference-between-two-elements