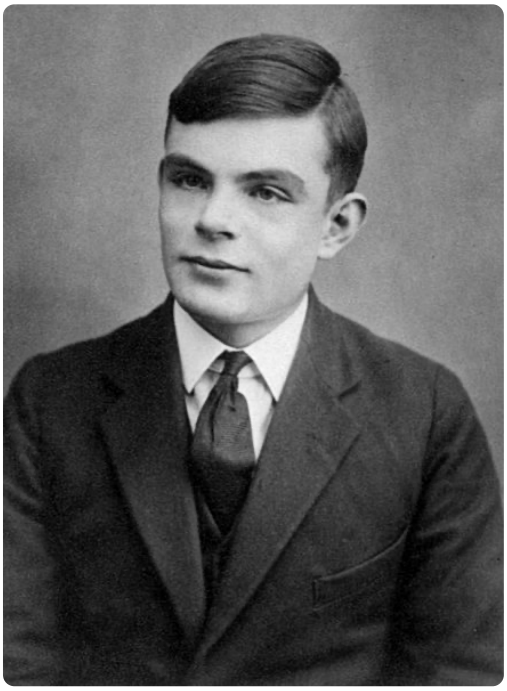


Quick admin stuff

(we will take more questions next class)

- 5 excellent TA's ← "good guys"

  Md. Hassan, Xiaoni, Nitish, Mohit, Yu

- fundamentalalgorithms.com

  one big .pdf w/ everything

  our goal is to be very organized + explicit on everything (to make up for pandemic)

  please ask staff for clarifications so we can revise syllabus

Some of the things in syllabus

weekly HW, 2 midterms, "tips for success", collab. policy, late policy, "resubmit HW policy", ~

Part I.

You already know TCS

- counting
- over/under

Counting

Counting

||||||||||||||||||||||| = 23

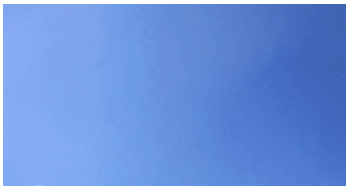
RHS much more efficient than LHS

"unary"

Decimal notation:

10^k #'s w/ k digits

Combinatorial explosion!

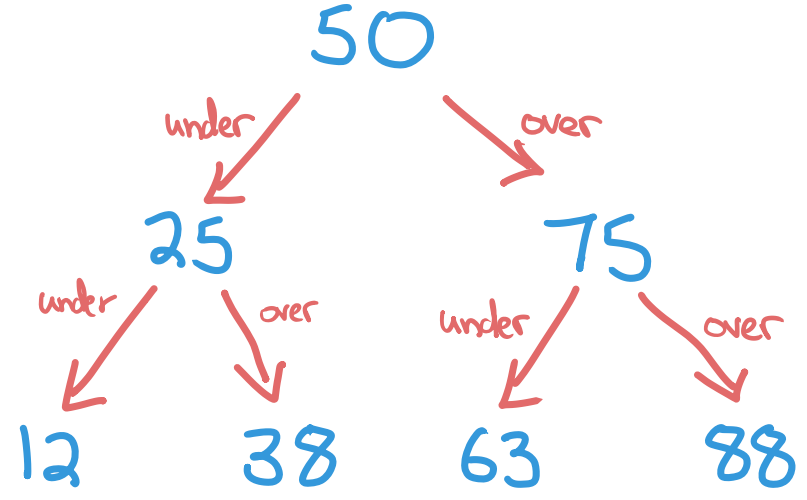


Over/under

guess # between 1 and 100

Slow algo: loop through 1, 2, ~, 100

FAST



rounds?

$\log_2 100$

≈ 6.67

$\Rightarrow$ 6 rounds

exponentials

$$2^n = \underbrace{2 \cdot 2 \cdot 2 \cdot \cdots \cdot 2}_{n \text{ times}}$$

logarithms

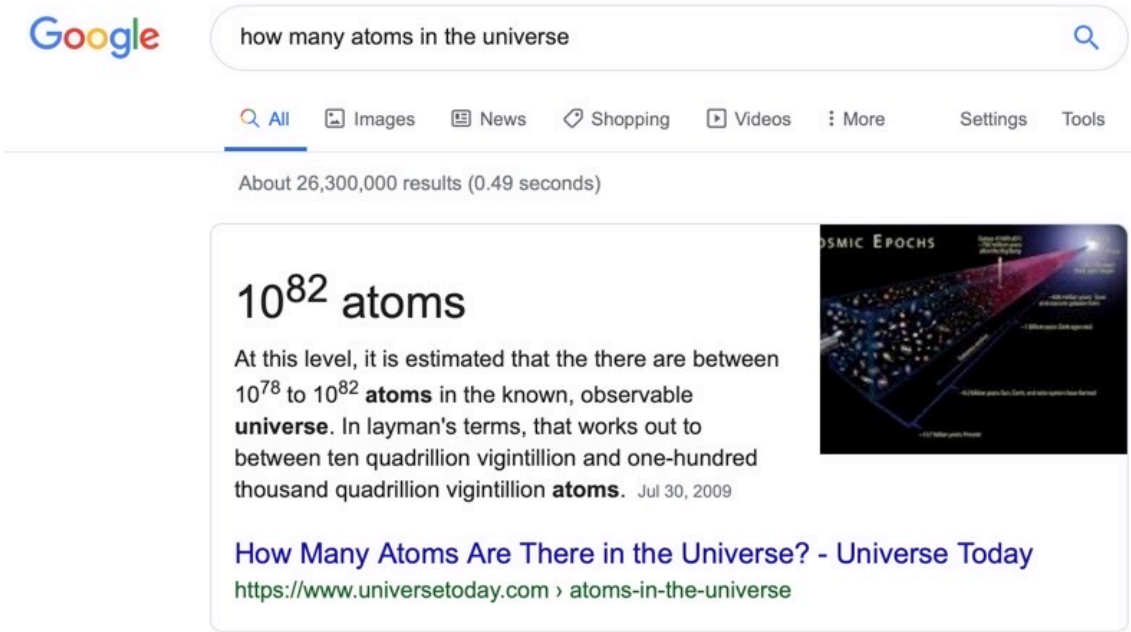
$\log_2 n$ = "logarithm (base 2) of n"

s.t. $2^{\log_2 n} = n$

also: $\log_e n$ (aka $\ln$)

$\log_{10} n$

$\log_b n$ for some $b > 1$



$$\log_2(\#\text{atoms}) = \log_2(10^{82})$$

$$= 82\log_2(10)$$

$$\log_2(10) \approx 3.32$$

$$\approx 272.398$$

$$\log_2(\log_2(\#\text{atoms})) =$$

$$\approx \log_2(272.4) \approx 8.\text{something}$$

$$(2^8 = 256)$$

TCS is about distinguishing

2^n, combinatorial explosion

vs

$\log_2 n$, binary search

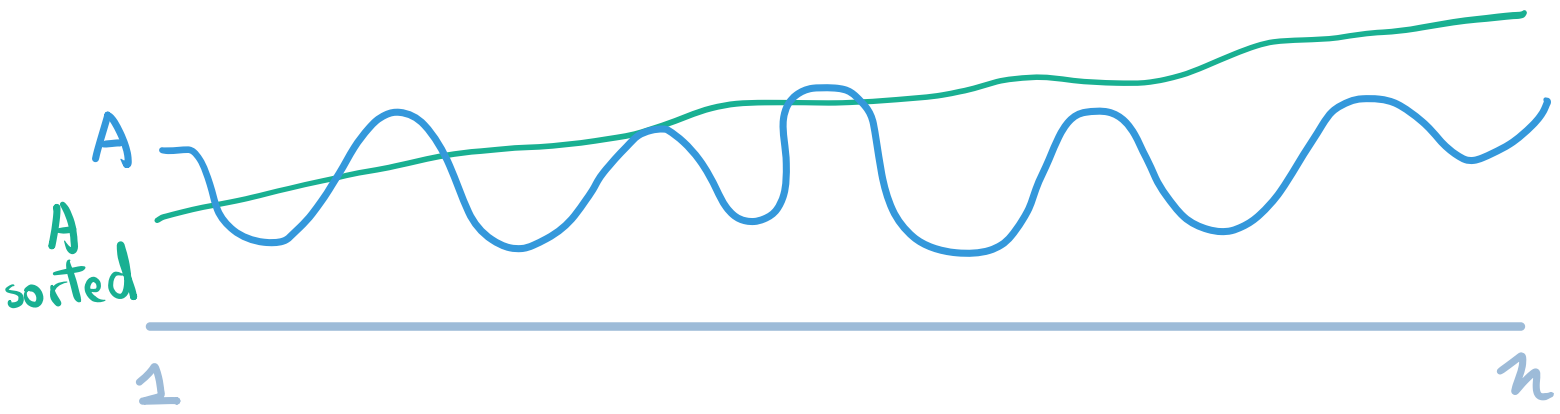
Part II: Sorting

Input: n numbers

5, 1, 8, 9, 2, 6, 7, 4, 11, 7, 9

Goal: sorted list

1, 2, 4, 5, 7, 7, 9, 9, 11



Human sort

5, 1, 8, 9, 2, 6, 7, 4, 11, 7, 9

- what comes first?
- what comes second?

1 2

builds sorted list
from beginning to end

each time, we scan the list of
n inputs

n iterations

x n items scanned per iteration

---

n^2 time

Insertion sort (array A[1..n])

① allocate B[1~n]

② for i=1,...,n

  ⓐ scan A[1~n] and let A[j] be be the smallest unmarked number

  ⓑ set B[i]=A[j] and mark A[j]

③ return B

Big O

$O(n^2)$ means: there exists constants N, c s.t. for all $n > N$, the algorithm runs in at most cn^2 "steps".

WTF "step" ?!?!

varies by computer, language, ...

only effect the constant

$O(n)$ hides the constant

so we can develop a general

THEORY OF ALGORITHMS

Human sort (aka insertion sort)

$O(n^2)$ time

Better?

n^2 ← each # is compared to every other #

can we sort all #'s without comparing every pair?

do we really need all comparisons?

Slightly different perspective

suppose given array A[1...n] that is supposedly sorted

how long does it take to verify A is sorted?

$a_1 \overset{\leq ?}{\wedge} a_2 \overset{\leq ?}{\wedge} a_3 \quad a_4 \quad a_5 \quad a_6 \quad \cdots$

$O(n)$

sorted? (A[1...n])

① for i=1,...,n-1

  ⓐ if A[i] > A[i+1]

    ① return False

② return True

O(n) time

Human sort (aka insertion sort)

$O(n^2)$ time

We can verify a sorted array in $O(n)$ time

Better?

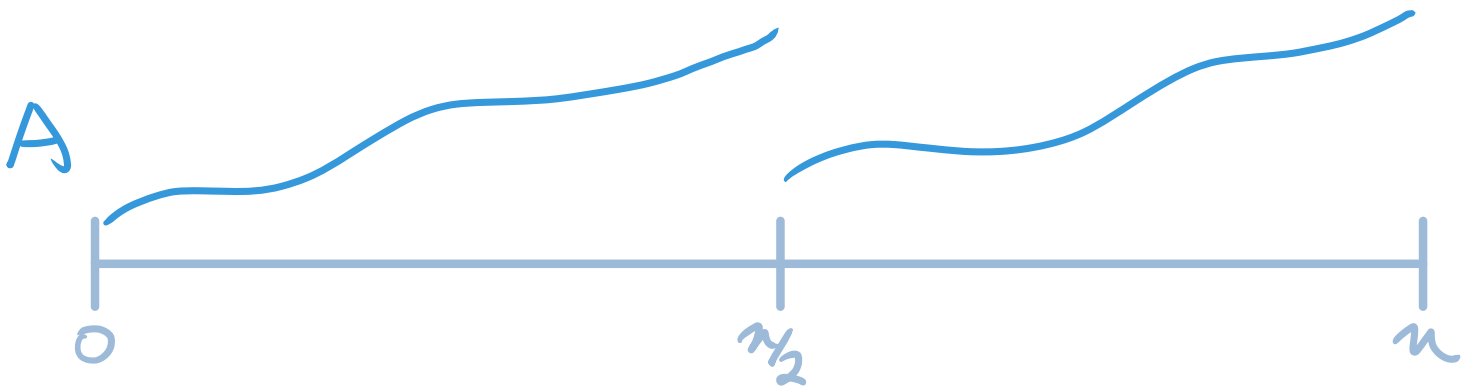
n^2 ← each # is compared to every other element

can we sort all #'s without comparing every pair?

do we really need all comparisons?

Simple case

Suppose first half and second half are already sorted



Merge (B[1,...,k], C[1,...,ℓ])

① allocate A[1,...,k+ℓ]

i←1, j←1

② while i≤k and j≤ℓ

ⓐ if B[i]≤C[j]

① A[i+j-1]= B[i], i←i+1

ⓑ else

② A[i+j-1]= C[j], j←j+1

③ while i≤k

ⓐ A[i+j-1]= B[i], i←i+1

④ while j≤ℓ

ⓐ A[i+j-1]= C[j], j←j+1

O(k+ℓ) time

Divide and Conquer

sort 1st half of A (recursively)

sort 2nd half of A (recursively)

merge 1st half and second half

```
Merge sort (A[1, m, n])
  if n≤1 return A
  k ← ⌈n/2⌉
  B ← merge-sort(A[1, m, k])
  C ← merge-sort(A[k+1, m, n])
  return merge(B, C)
```

$T(n)$ = time for input size n

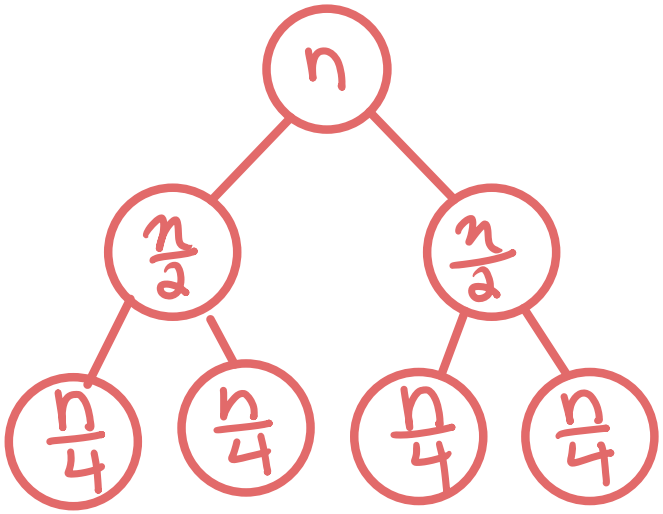
$T(1) = T(0) = 1$

$n > 1$:

$$T(n) = T(\lceil \tfrac{n}{2} \rceil) + T(n - \lceil \tfrac{n}{2} \rceil) + O(n)$$

$$\overset{*}{\sim} 2T(\tfrac{n}{2}) + O(n)$$

* see notes for justification



height = log n

Work

n

$2 \cdot (\frac{n}{2}) = n$

$4 \cdot (\frac{n}{4}) = n$

$2^k \cdot \frac{n}{2^k} = n$

1 2 1 1 ~ 1 $+ n \cdot 1 = n$

$n \times \log(n)$

k s.t. $2^k = n$

$k = \log_2(2^k) = \log_2(n)$

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

III

Lower Bounds for Sorting

merge-sort: $O(n \log n)$. better?

optimal?

Goal: prove lower bound for all sorting algorithms

e.g. prove any correct algorithm has to take at least $\Omega(f(n))$ time

"big-Omega"

(like $O(\cdot)$ but for lower bounds)

Comparison model

- Input: $x_1, \ldots, x_n$ comparable
- only info is via comparison queries

"is $x_i < x_j$?"

Goal lower bound # comparison Q's

Suppose algo makes k queries and outputs list $(x_{i_1}, x_{i_2}, \ldots, x_{i_n})$

1. $x_{i_1} < x_{j_1}$?
2. $x_{i_2} < x_{j_2}$?
3. $x_{i_3} < x_{j_3}$?
4. $x_{i_4} < x_{j_4}$?

{

k. $x_{i_k} < x_{j_k}$?

$\Rightarrow (x_{i_1}, x_{i_2}, x_{i_3}, \ldots, x_{i_n})$

- output is a function of k queries
- each query has 2 outcomes (YES/NO)

$\Rightarrow$ at most 2^k possible outputs

meanwhile...

$n!$ possible lists

$\Rightarrow$ $2^k \geq n!$

just to be able to output all lists

$2^k \geq n!$

$k \geq \log(n!)$

$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$

$\geq \left(\frac{n}{2}\right)^{n/2}$

$k \geq \log\left(\left(\frac{n}{2}\right)^{n/2}\right) = \frac{n}{2}\log\left(\frac{n}{2}\right)$

$= \Omega(n \log n)$

Theorem In the comparison (only) model, any (correct, deterministic) sorting algorithm makes

$\Omega(n \log n)$ comparisons

in the worst case.

Sorting (in comparison-only model)

Upper bound	Lower bound
$O(n \log n)$ (mergesort)	$\Omega(n \log n)$

in general

(any lower bound) $\leq$ (any upper bound)

here we also have

(an upper bound) $\leq$ (some constant) (a lower bound)

[$O(n \log n)$] [$\Omega(n \log n)$]

bounds mutually certify each other to be optimal up to constants.

"bounds are tight"

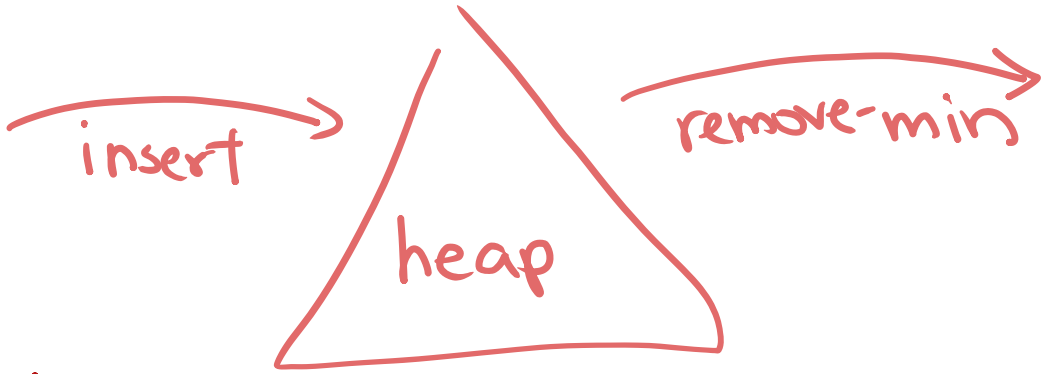
Conclusion: $n \log n$ is "right answer" for sorting in comparison model

Other sorting algorithms (for fun)

- **Heap sort** — uses heap data structure
- **Quicksort** — randomized divide & conquer

Heaps

collection of comparable values (e.g. numbers)



Operations:

- .insert(x): insert element x into heap
- .remove-min(): remove + return min element in heap

O(log n) time

Heap-sort:

1. insert all n elements
2. Remove-min all n elements

O(n log n)

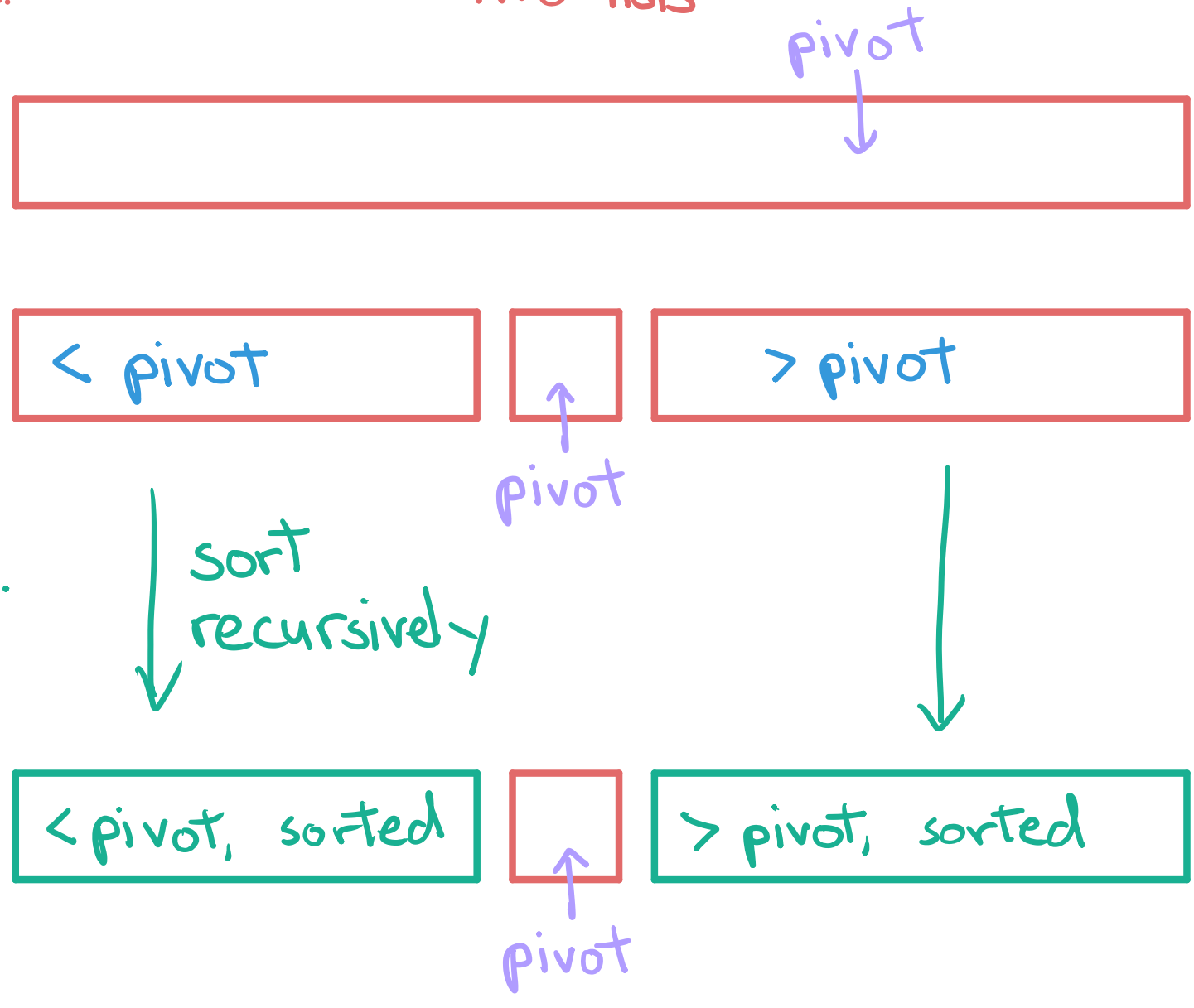
Quicksort

randomized divide & conquer

1. pick random "pivot"

   (i.e., select 1 element in A unif. at random)

2. divide A into two lists



Quicksort is a randomized algorithm

we are interested in average run time
(over random choices)

later we will prove:

Theorem Quicksort takes $O(n \log n)$ randomized time in expectation

fun fact: very practical
(easy to implement,
works well in practice)

D.1 Recitation 1

Exercise 1.1. Use recursion trees to solve the following recurrences.

1. $T(n) = 3T(n/3) + 6n$ and $T(1) = 2$.
2. $T(n) = 2T(n/3) + 4n$ and $T(1) = 7$.
3. $T(n) = 4T(n/3) + n$ and $T(1) = 11$.

Exercise 1.2. Let $A[1..n] \in \mathbb{R}^n$ be an array of n numbers. An **inversion** is a pair of numbers out of increasing order; more precisely, a pair of indices $i, j \in [n]$ such that $i < j$ and $A[i] > A[j]$. Design and analyze an algorithm (as fast as possible) for counting the number of inversions in A.

Exercise 1.3. Prove a $\Omega(\log n)$-query lower bound for binary search in the comparison-only model.

Exercise 1.6. Consider coordinates in the plane; i.e., pairs (a, b) where $a, b \in \mathbb{R}$. Recall that two coordinate pairs (a, b) and (c, d) are **comparable** if either (a) $a \leq c$ and $b \leq d$, or (b) $c \leq a$ and $d \leq c$.

Given a set of n coordinate pairs $(a_1, b_1), \ldots, (a_n, b_n)$, consider the problem of computing the number of pairs of coordinate pairs that are comparable. Design and analyze an algorithm (as fast as possible) for counting the number of comparable pairs.

Exercise 2.2. Suppose one had access to a polynomial time algorithm that solves the decision version of Boolean satisfiability (outputting `true` or `false` depending on whether there *exists* a satisfying assignment.). Show how to use this algorithm to obtain a polynomial time algorithm for the constructive version of Boolean satisfiability, outputting a satisfying assignment if one exists.

Exercise 2.3. Show that any Boolean formula with n variables and total size m can be converted to a Boolean formula with n variables and total size $O(m)$ such that every clause contains at most two sub-clauses or variables. That is, in the tree representation of the formula, each $\wedge$ and $\vee$ would have two children. For example, the formula for XOR, $f(x_1, x_2) = (\bar{x}_1 \wedge x_2) \vee (x_1 \wedge \bar{x}_2)$, is in this form.

Exercise 2.4. Show that any boolean formula with n variables and total size m can be converted to a boolean formula in CNF with $O(m + n)$ variables, $O(m)$ clauses, and total size $O(m)$.[1][2]

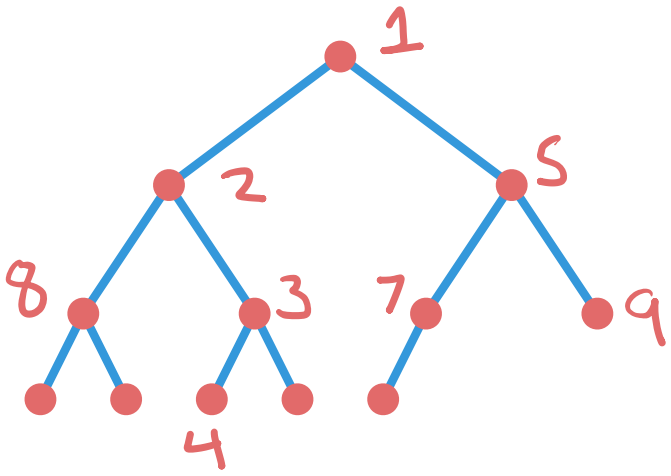
[1] It might simplify things to first apply the preceding exercise.

[2] As a second (appropriately vague) hint, how do you express "$a = b \vee c$" and "$a = b \wedge c$" in CNF?

(Extra Background on Heaps)

How to make a heap

full binary tree

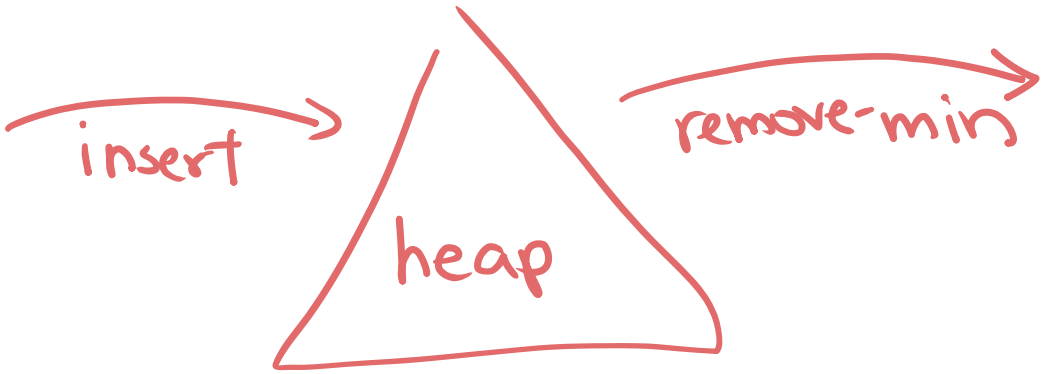


invariant:

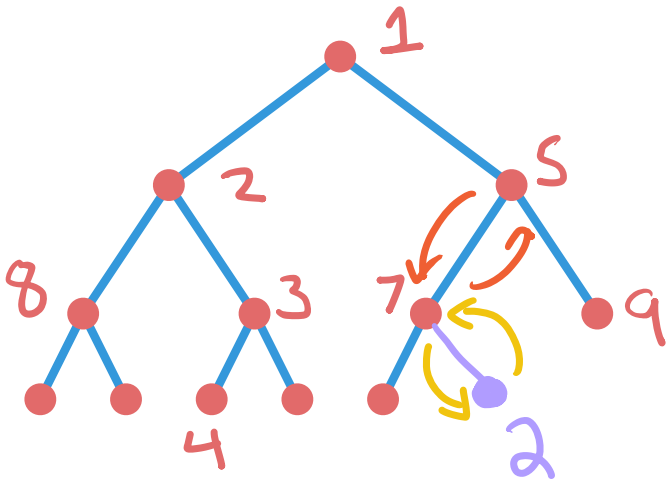
each node's # is ≤ any of its children

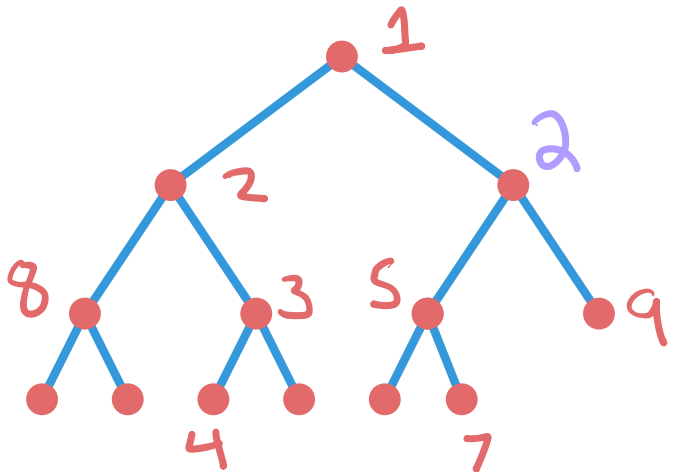
Operations:

- .insert(x): insert element x into heap
- ✓ .min(): O(1) returns minimum element in heap
- .remove-min(): remove & return min element in heap



.insert(x): insert element x into heap





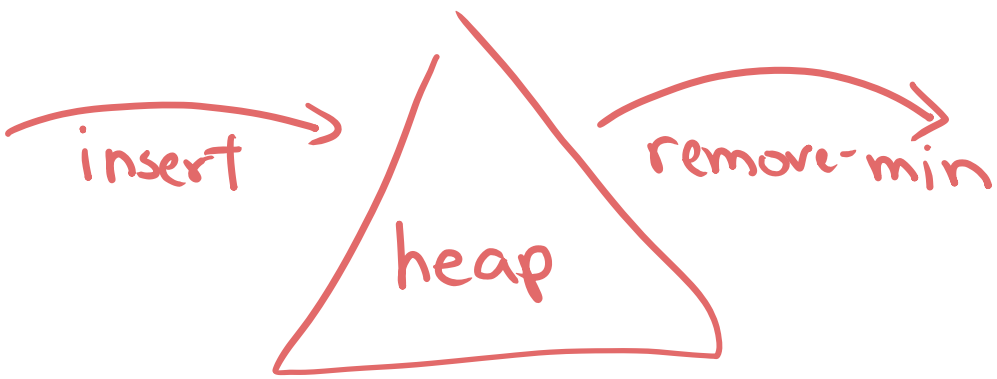
① insert x into next empty slot

② while x is not at the root and
x is smaller than its parent

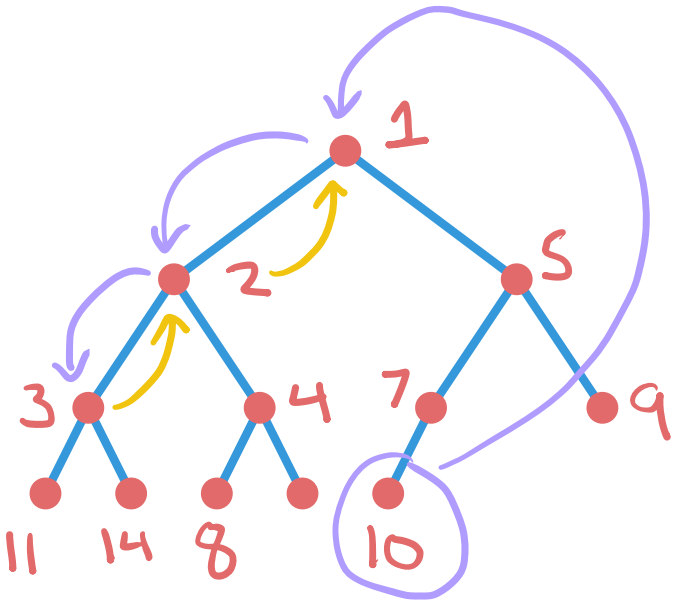
③ swap x w/ its parent

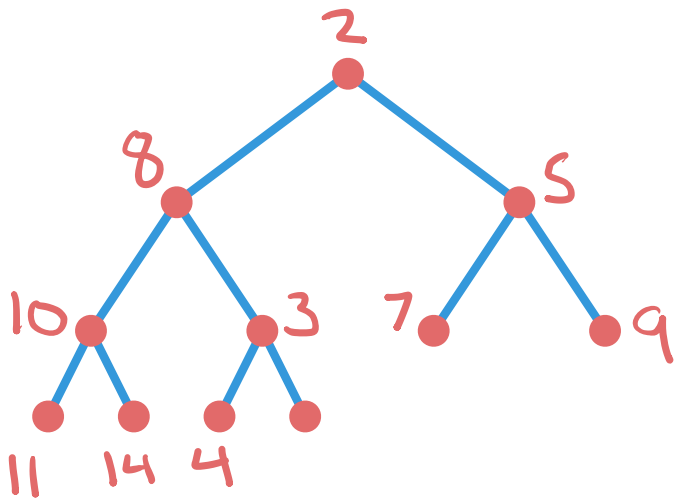
Operations:

- .insert(x): insert element x into heap — $O(\log n)$
- .min(): returns minimum element in heap — $O(1)$
- .remove-min(): remove & return min element in heap



.remove-min(): remove & return min element in heap





① save the root element, which is returned at the end

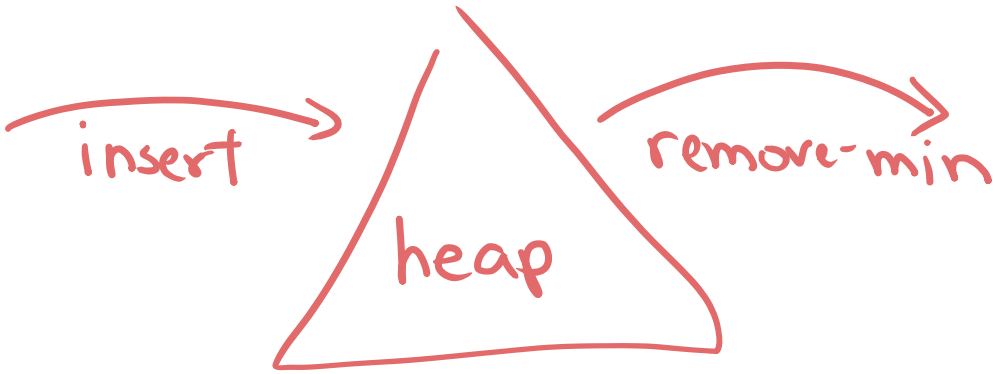
② take the last element x and put it at the root

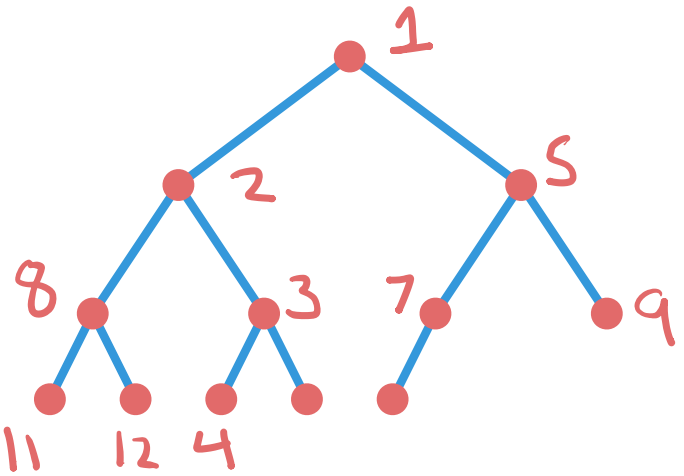
③ while x is > than at least one child

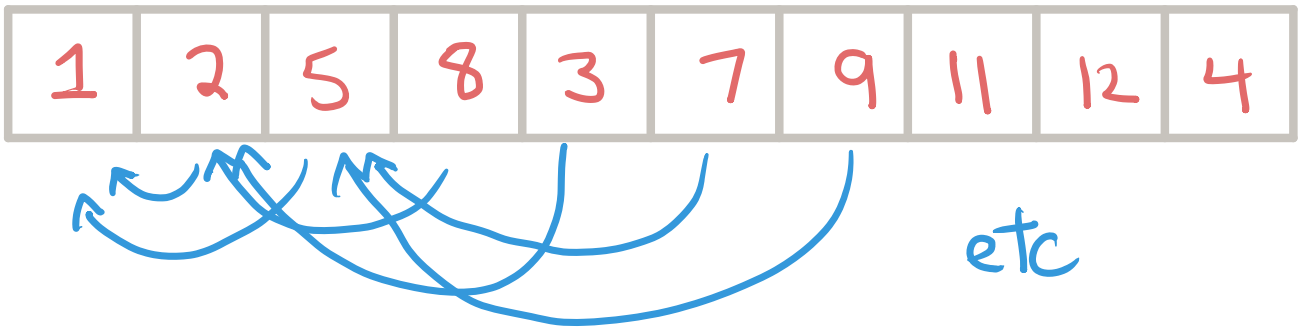
ⓐ swap it w/ the smaller child

Operations:

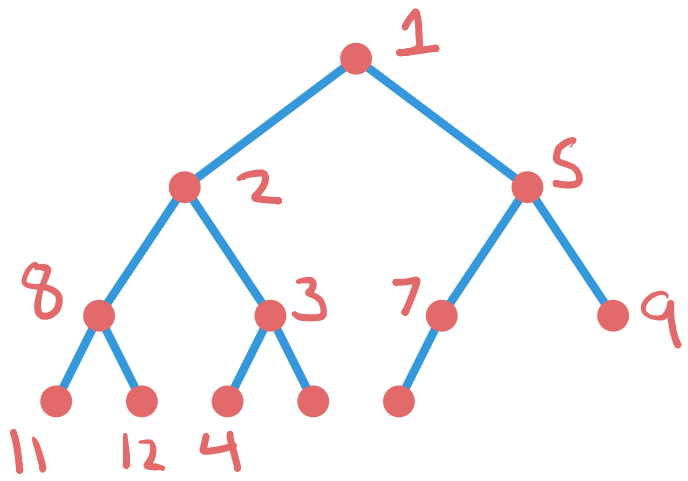
- .insert(x): insert element x into heap — O(log n)
- .min(): returns minimum element in heap — O(1)
- .remove-min(): remove & return min element in heap — O(log n)







$i \rightarrow \lfloor i/2 \rfloor$



1	2	5	8	3	7	9	11	12	4						

preallocated array of size n

what if we don't know n?

Operations:

- insert(x): $O(\log n)$ ← becomes $O(n)$ worst case — insert element x into heap
- min(): $O(1)$ returns minimum element in heap
- remove-min(): $O(\log n)$ remove & return min element in heap