

CS 240E

Personal Notes

Marcus Chan

Taught by Therese Biedl

UW CS '25



Chapter 1: Algorithm Analysis

HOW TO "SOLVE" A PROBLEM

When solving a problem, we should
① write down exactly what the problem is.

eg Sorting Problem
→ given n numbers in an array,
put them in sorted order

② Describe the idea;

eg Insertion Sort



Idea:
repeatedly move
one item into the
correct space of
the sorted part.

③ Give a detailed description; usually pseudocode.

eg Insertion Sort:

```
for i=1,..,n-1
    j=i
    while j>0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j-=1
```

④ Argue the correctness of the algorithm.
→ In particular, try to point out loop invariants & variants.

⑤ Argue the run-time of the program.

→ We want a theoretical bound.
(Using asymptotic notation).

To do this, we count the # of primitive operations.

PRIMITIVE OPERATIONS

In our computer model,
① our computer has memory cells
② all cells are equal
③ all cells are big enough to store our numbers

Then, "primitive operations" are $+, -, \times, \div$, load & following references.

We also assume each primitive operation takes the same amount of time to run.

ASYMPTOTIC NOTATION

BIG-O NOTATION: $O(f(n))$

We say that " $f(n) \in O(g(n))$ " if there exist $c > 0, n_0 > 0$ s.t.

$$|f(n)| \leq c|g(n)| \quad \forall n \geq n_0.$$

eg $f(n) = 75n + 500$ & $g(n) = 5n^2$,
 $c=1$ & $n_0=20$

Usually, " n " represents input size.

SHOW $2n^2 + 3n^2 + 11 \in O(n^2)$

To show the above, we need to find c, n_0 such that $0 \leq 2n^2 + 3n^2 \leq cn^2 \quad \forall n \geq n_0$.

Sol². Consider $n_0=1$. Then

$$\begin{aligned} 1 \leq n &\Rightarrow 1 \leq 1n^2 \\ 1 \leq n &\Rightarrow n \leq n^2 \Rightarrow 3n \leq 3n^2 \\ &\xrightarrow{(+)} 2n^2 \leq 2n^2 \\ &\Rightarrow 2n^2 + 3n^2 \leq 11n^2 + 3n^2 = 14n^2 \\ \text{Hence } c=14 \text{ & } n_0=1, \text{ so } 2n^2 + 3n^2 \in O(n^2). \end{aligned}$$

Ω -NOTATION (BIG OMEGA): $f(n) \in \Omega(g(n))$

We say " $f(n) \in \Omega(g(n))$ " if there exist $c > 0, n_0 > 0$ such that

$$c|g(n)| \leq |f(n)| \quad \forall n \geq n_0.$$

Θ -NOTATION (BIG THETA): $f(n) \in \Theta(g(n))$

We say " $f(n) \in \Theta(g(n))$ " if there exist $c_1, c_2 > 0, n_0 > 0$ such that

$$c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|.$$

Note that

$$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \text{ & } f(n) \in \Omega(g(n)).$$

\mathcal{O} -NOTATION (SMALL O): $f(n) \in \mathcal{O}(g(n))$

We say " $f(n) \in \mathcal{O}(g(n))$ " if for any $c > 0$, there exists some $n_0 > 0$ such that

$$|f(n)| < c|g(n)| \quad \forall n \geq n_0.$$

If $f(n) \in \mathcal{O}(g(n))$, we say $f(n)$ is "asymptotically strictly smaller" than $g(n)$.

ω -NOTATION (SMALL OMEGA): $f(n) \in \omega(g(n))$

We say $f(n) \in \omega(g(n))$ if for all $c > 0$, there exists some $n_0 > 0$ such that

$$0 \leq c|g(n)| < |f(n)| \quad \forall n \geq n_0.$$

FINDING RUNTIME OF A PROGRAM

- To evaluate the run-time of a program, given its pseudocode, we do the following:
- Annotate any primitive operations with just " $\Theta(1)$ ";
 - For any loops, find the worst-case bound for how many times it will execute;
 - Calculate the big-O run time of the program;
 - Argue this bound is tight (ie show program is also in $\Omega(g(n))$, so runtime $\in \Theta(g(n))$.)

e.g. insertion sort

```
for i=1, ..., n-1
    j=i  $\Theta(1)$ 
    while j>0 and A[j-1] > A[j]
        swap A[j] and A[j-1]  $\Theta(1)$ 
        j-1  $\Theta(1)$ 
```

Then, let c be a const s.t. the upper bounds all the times needed to execute one line.

So runtime $\leq n \cdot n \cdot c = c \cdot n^2 \in O(n^2)$.
Next, consider the worst pos. case of insertion sort.



For each $A[i]$, we need $i-1$ swaps.

So

$$\begin{aligned} \text{runtime} &\geq \sum_{i=2}^{n-1} (i-1) \\ &= \sum_{i=1}^{n-2} i = \frac{(n-2)(n-1)}{2} \\ &\in \Omega(n^2), \end{aligned}$$

and so
runtime $\in \Theta(n^2)$.

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L < \infty \Rightarrow f(n) \in O(g(n))$$

\Leftarrow (CL1.1(2))

\Leftarrow (let $f(n), g(n)$ be such that $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L < \infty$.

Then necessarily $f(n) \in O(g(n))$.

Proof. We know $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L$.
 $\Rightarrow \forall \epsilon > 0: \exists n_0$ s.t. $|\frac{f(n)}{g(n)} - L| < \epsilon \quad \forall n \geq n_0$.

We want to show $\exists C > 0, \exists n_0 \Rightarrow \forall n \geq n_0, f(n) \in O(g(n))$.

$\exists C > 0, \exists n_0 \Rightarrow \forall n \geq n_0, f(n) \in O(g(n))$.

Choose $\epsilon = 1$. Then there exists n_1 s.t.

$$\forall n \geq n_1, |\frac{f(n)}{g(n)} - L| \leq 1.$$

$$\Leftrightarrow \frac{f(n)}{g(n)} - L \leq |\frac{f(n)}{g(n)} - L| \leq 1.$$

$$\Leftrightarrow \frac{f(n)}{g(n)} \leq L + 1$$

$$\Leftrightarrow f(n) \in O(g(n)) + g(n) \quad (\text{since } g(n) > 0)$$

Choose $C = L+1$. Note $f(n), g(n) > 0$, so $L+1 > 0$, and since $L < \infty$, thus $C < \infty$.
Now, for all $n \geq n_1$, $f(n) \leq C \cdot g(n)$, and so $f(n) \in O(g(n))$. \square

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Leftrightarrow f(n) \in O(g(n))$$

\Leftarrow (CL1.1(1))

\Leftarrow (let $f(n), g(n)$. Then $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$

iff $f(n) \in o(g(n))$.

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L > 0 \Rightarrow f(n) \in \Omega(g(n))$$

\Leftarrow (CL1.1(3))

\Leftarrow (let $f(n), g(n)$ such that $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L > 0$.

Then necessarily $f(n) \in \Omega(g(n))$.

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \in \omega(g(n))$$

\Leftarrow (CL1.1(4))

\Leftarrow (let $f(n), g(n)$ such that $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$.

Then necessarily $f(n) \in \omega(g(n))$.

OTHER LIMIT RULES

The following are corollaries of the limit rules:

- ① $f(n) \in O(fcn) \quad \left\{ \begin{array}{l} \text{(Identity)} \\ \text{(Constant multiplication)} \end{array} \right.$
- ② $K \cdot f(n) \in O(fcn) \quad \forall K \in \mathbb{R}$
- ③ $f(n) \in O(g(n)), g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$
- ④ $f(n) \in \Omega(g(n)), g(n) \in \Omega(h(n)) \Rightarrow f(n) \in \Omega(h(n))$
- ⑤ $f(n) \in O(g(n)), g(n) \leq h(n) \quad \forall n \geq N \Rightarrow f(n) \in O(h(n))$
- ⑥ $f(n) \in \Omega(g(n)), g(n) \geq h(n) \quad \forall n \geq N \Rightarrow f(n) \in \Omega(h(n))$
- ⑦ $f_1(n) \in O(g_1(n)), f_2(n) \in O(g_2(n)) \Rightarrow f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$
- ⑧ $f_1(n) \in \Omega(g_1(n)), f_2(n) \in \Omega(g_2(n)) \Rightarrow f_1(n) + f_2(n) \in \Omega(g_1(n) + g_2(n))$
- ⑨ $h(n) \in O(fcn) + g(n) \Rightarrow h(n) \in O(\max(fcn, g(n)))$
- ⑩ $h(n) \in \Omega(fcn) + g(n) \Rightarrow h(n) \in \Omega(\max(fcn, g(n)))$

$f(n) \in P_d(\mathbb{R}) \Rightarrow f(n) \in \Theta(n^d)$ \Leftarrow POLYNOMIAL RULE

\Leftarrow (at $f(n) \in P_d(\mathbb{R})$, ie of the form $f(n) = c_0 + c_1 n + \dots + c_d n^d$).

Then necessarily $f(n) \in \Theta(n^d)$.

$b > 1; \log_b(n) \in \Theta(\log n)$ \Leftarrow LOG RULE I

\Leftarrow (at $b > 1$. Then necessarily $\log_b(n) \in \Theta(\log n)$).

Proof. Note

$$\lim_{n \rightarrow \infty} \frac{\log_b(n)}{\log n} = \lim_{n \rightarrow \infty} \frac{\left(\frac{\log(n)}{\log(b)}\right)}{\log(n)} = \lim_{n \rightarrow \infty} \frac{1}{\log(b)} > 0,$$

so by Limit Rules 1 & 3, $\log_b(n) \in \Theta(\log n)$. \square

*convention:
"log" = "log₂".

$c, d > 0; \log^c n \in O(n^d)$ \Leftarrow LOG RULE II

\Leftarrow (at $c, d > 0$. Then necessarily $\log^c n \in O(n^d)$),

where $\log^c n \equiv (\log n)^c$.

Proof. See that

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{\ln^k n}{n} &= \lim_{n \rightarrow \infty} \frac{k \ln^{k-1} n \cdot \frac{1}{n}}{1} \\ &= \dots \\ &= \lim_{n \rightarrow \infty} \frac{k!}{n} = 0, \end{aligned}$$

so $\ln^k n \in o(n)$.

Fix $c, d > 0$. Then

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{\ln^c n}{n^d} &= \left(\lim_{n \rightarrow \infty} \frac{\ln^c n}{\ln^d n} \right)^d \\ &\leq \left(\lim_{n \rightarrow \infty} \frac{\ln^c n}{n} \right)^d \\ &= 0^d = 0. \end{aligned}$$

As $\log^c n = (\frac{1}{\ln 2})^c \ln^c n$, thus $\lim_{n \rightarrow \infty} \frac{\log^c n}{n^d} = (\frac{1}{\ln 2})^c \cdot 0 = 0$.

Proof follows from the limit rule. \square

$f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$

\Leftarrow Suppose $f(n) \in o(g(n))$. Then $f(n) \in O(g(n))$.

$f(n) \in O(g(n)) \Rightarrow f(n) \in \Omega(g(n))$

\Leftarrow Suppose $f(n) \in O(g(n))$. Then $f(n) \in \Omega(g(n))$.

Proof. Prove by contrapositive:

$$f(n) \in \Omega(g(n)) \Rightarrow f(n) \notin o(g(n)).$$

Consider cases for $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$.

Case 1 It DNE.

$$\Rightarrow f(n) \notin o(g(n)).$$

Case 2 Limit exists.

Then by $f(n) \in \Omega(g(n))$, thus

$$f(n) \geq c \cdot g(n) \quad \text{for some } c > 0 \text{ & } n \geq n_0.$$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \geq c > 0$$

$$\Rightarrow \text{limit} \neq 0$$

$$\Rightarrow f(n) \notin o(g(n)). \quad \square$$

$f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$

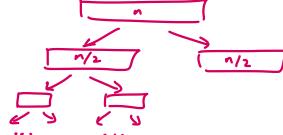
Suppose $f(n) \in \omega(g(n))$. Then $f(n) \in \Omega(g(n))$.

WORST-CASE RUNTIME: $T_A^{\text{worst}}(n)$

The "worst-case runtime" for an algorithm, denoted $T_A^{\text{worst}}(n)$, is the max run-time among all instances of size n .

ANALYZING RECURSIVE ALGORITHMS

Consider merge sort:



Analysis of MergeSort:

```
MergeSort(A, n, l=0, r=n-1, S=NULL)
A: array of size n, 0 ≤ l ≤ r ≤ n-1
if S is NIL init it as array S[0...n-1]
if (r < l) then
    return
else
    m = (l+r)/2
    MergeSort(A, n, l, m, S)
    MergeSort(A, n, m+1, r, S)
    Merge(A, l, m, r, S)
```

Merge(A, l, m, r, S)

$A[0, \dots, n-1]$ is an array. $A[l, \dots, m]$ is sorted.
 $A[m+1, \dots, r]$ is sorted. $S[0, \dots, n-1]$ is an array.

```
copy A[l...r] into S[l...r]
int i_L=l; int i_R=m+1
for (k=L; k < R; k++) do
    if (i_L > m) A[k] ← S[i_R++]
    else if (i_R > r) A[k] ← S[i_L++]
    else if (S[i_L] ≤ S[i_R]) A[k] ← S[i_L++]
    else A[k] ← S[i_R++]
```

Arguing run-time:

Let $T(n)$ = run-time of merge sort with n items.

$$\therefore T(n) \leq \begin{cases} c, & n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n, & \text{otherwise} \end{cases}$$

$\in \Theta(n \log n)$ (see below)

SOME RECURRENCE RELATIONS

Note:

Recursion...	... resolves to ...	Example
$T(n) = T(n/2) + \Theta(1)$	$T(n) \in \Theta(\log n)$	Binary search
$T(n) = 2T(n/2) + \Theta(n)$	$T(n) \in \Theta(n \log n)$	Mergesort
$T(n) = 2T(n/2) + \Theta(\log n)$	$T(n) \in \Theta(n)$	Heapify
$T(n) = T(cn) + \Theta(n), 0 < c < 1$	$T(n) \in \Theta(n)$	Selection
$T(n) = 2T(n/4) + \Theta(1)$	$T(n) \in \Theta(N^{\sqrt{2}})$	Range Search
$T(n) = T(N^{\sqrt{n}}) + \Theta(1)$	$T(n) \in \Theta(\log \log n)$	Interpolation Search

SORTING PERMUTATION [OF AN ARRAY]

A "sorting permutation" of an array A is the permutation $\pi: \{0, \dots, n-1\} \rightarrow \{0, \dots, n-1\}$ such that

$$A[\pi(0)] \leq A[\pi(1)] \leq \dots \leq A[\pi(n-1)].$$

* A off would be "sorted".

eg if $A = [14, 3, 2, 6, 1, 11, 7]$, then

$$\pi = [4, 2, 1, 3, 6, 5, 0].$$

For a sorting permutation π , we define its inverse π^{-1} to be the array which entries have exactly the same "relative order" as in A .

eg $\pi^{-1} = [6, 2, 1, 3, 0, 5, 4]$ (if A is as above)

We denote " Π_n " to be the set of all sorting permutations of $\{0, \dots, n-1\}$.

AVERAGE-CASE RUNTIME: $T_A^{\text{avg}}(n)$

The "average-case run-time" of an algorithm is defined to be

$$T_A^{\text{avg}}(n) := \underset{I \in \mathcal{X}_n}{\text{avg}} T_A(I) = \frac{1}{|\mathcal{X}_n|} \sum_{I \in \mathcal{X}_n} T_A(I),$$

where \mathcal{X}_n is the set of instances of size n .

In particular, if we can map each instance I to a permutation $\pi \in \Pi_n$, where Π_n is the set of all permutations of $\{0, \dots, n-1\}$, then we may alternatively state that

$$T^{\text{avg}}(n) = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi)$$

where $T(\pi)$ is the number of comparisons on instance π^{-1} .

EXAMPLE: AVGCASEDEMO

Consider the algorithm

```

avgCaseDemo(A, n)
// array A stores n distinct numbers
1. if n≤2 return
2. if A[n-2] ≤ A[n-1] then
3.   avgCaseDemo(A[0... $\lfloor \frac{n}{2} \rfloor - 1$ ,  $\frac{n}{2}$ ]) // good case
4. else
5.   avgCaseDemo(A[0...n-3], n-2) // bad case

```

We claim $T^{\text{avg}}(n) \in O(\log n)$.

Proof. To avoid constants, let $T(\cdot) := \# \text{ of recursions}$; the run-time is proportional to this.
As all numbers are distinct, we may associate each array with a sorting permutation.
So for $\pi \in \Pi_n$, let $T(\pi) = \# \text{ of recursions done if the input array has sorting permutation } \pi$.
Note we have two kinds of permutations:
 ① "Good" permutations — if $A[n-2] < A[n-1]$; or
 ② "Bad" permutations — if $A[n-2] > A[n-1]$.

Denote
 $\Pi_n^{\text{good}} = \# \text{ of good permutations of size } n$
 $\& \Pi_n^{\text{bad}} = \# \text{ of bad permutations of size } n$.

Then, we claim that

$$\sum_{\substack{\pi \in \Pi_n \\ \text{good}}} T(\pi) \leq |\Pi_n^{\text{good}}| (1 + T^{\text{avg}}(\lfloor \frac{n}{2} \rfloor))$$

$$\& \sum_{\substack{\pi \in \Pi_n \\ \text{bad}}} T(\pi) \leq |\Pi_n^{\text{bad}}| (1 + T^{\text{avg}}(n-2)).$$

Proof. We only prove this for good permutations; the other claim is similar.

Fix $\pi \in \Pi_n^{\text{good}}$, and let π_{half} be the permutation of the recursion;
ie $\pi_{\text{half}} = \text{the sorting perm of } \pi[0, \dots, \lfloor \frac{n}{2} \rfloor]$ & $T(\pi) = 1 + T(\pi_{\text{half}})$.

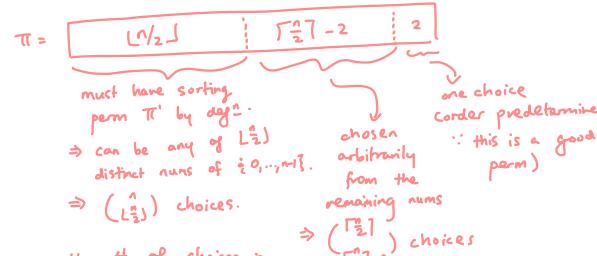
Note that $\pi_{\text{half}} \in \Pi_{\lfloor \frac{n}{2} \rfloor}$. Then see that

$$\begin{aligned} \sum_{\pi \in \Pi_n^{\text{good}}} T(\pi) &= \sum_{\pi \in \Pi_n^{\text{good}}} (1 + T(\pi_{\text{half}})) \\ &= |\Pi_n^{\text{good}}| + \sum_{\pi \in \Pi_n^{\text{good}}} T(\pi_{\text{half}}) \\ &= |\Pi_n^{\text{good}}| + \sum_{\substack{\pi' \in \Pi_{\lfloor \frac{n}{2} \rfloor} \\ \exists \pi \in \Pi_n^{\text{good}} \text{ for which } \pi_{\text{half}} = \pi'}} |\cdot T(\pi')|. \end{aligned}$$

We next prove the following claim:

Claim If $n \geq 3$, then $|\Pi_n^{\text{good}}(\pi')| = \frac{n!}{2!(\lfloor \frac{n}{2} \rfloor)!} \quad \forall \pi' \in \Pi_{\lfloor \frac{n}{2} \rfloor}$.

Proof. Fix $\pi' \in \Pi_{\lfloor \frac{n}{2} \rfloor}$. See that



Hence, the # of choices is $\binom{n}{2} \cdot \binom{\lfloor \frac{n}{2} \rfloor}{\lfloor \frac{n}{2} \rfloor - 2} (\lfloor \frac{n}{2} \rfloor - 2)! = \dots = \frac{n!}{\lfloor \frac{n}{2} \rfloor! \cdot 2}$,

as needed. \blacksquare

Then, since $|\Pi_n^{\text{good}}| = \frac{1}{2} |\Pi_n| = \frac{n!}{2}$, it follows that

$$|\Pi_n^{\text{good}}| / |\Pi_{\lfloor \frac{n}{2} \rfloor}|, \text{ and so}$$

$$\begin{aligned} \sum_{\pi \in \Pi_n^{\text{good}}} T(\pi) &= |\Pi_n^{\text{good}}| + \sum_{\pi' \in \Pi_{\lfloor \frac{n}{2} \rfloor}} |\Pi_n^{\text{good}}(\pi')| \cdot T(\pi') \\ &= |\Pi_n^{\text{good}}| + \sum_{\pi' \in \Pi_{\lfloor \frac{n}{2} \rfloor}} \frac{|\Pi_n^{\text{good}}|}{|\Pi_{\lfloor \frac{n}{2} \rfloor}|} \cdot T(\pi') \\ &= |\Pi_n^{\text{good}}| \left(1 + \frac{1}{|\Pi_{\lfloor \frac{n}{2} \rfloor}|} \sum_{\pi' \in \Pi_{\lfloor \frac{n}{2} \rfloor}} T(\pi') \right) \\ &= |\Pi_n^{\text{good}}| (1 + T^{\text{avg}}(\lfloor \frac{n}{2} \rfloor)) \end{aligned}$$

as needed. \blacksquare

Next, see that $|\Pi_n^{\text{good}}| = |\Pi_n^{\text{bad}}|$, since we can map any bad perm to a good one (and v.v.) by swapping $A[n-2]$ & $A[n-1]$.

Thus $|\Pi_n^{\text{good}}| = |\Pi_n^{\text{bad}}| = \frac{n!}{2}$, and so

$$\begin{aligned} T^{\text{avg}}(n) &\leq \frac{1}{|\Pi_n|} \sum_{\pi \in \Pi_n} T(\pi) \leq \frac{1}{|\Pi_n|} \left(\sum_{\pi \in \Pi_n^{\text{good}}} T(\pi) + \sum_{\pi \in \Pi_n^{\text{bad}}} T(\pi) \right) \\ &\leq \frac{1}{|\Pi_n|} (|\Pi_n^{\text{good}}| (1 + T^{\text{avg}}(\lfloor \frac{n}{2} \rfloor)) + |\Pi_n^{\text{bad}}| (1 + T^{\text{avg}}(n-2))) \\ &= 1 + \frac{1}{2} T^{\text{avg}}(\lfloor \frac{n}{2} \rfloor) + \frac{1}{2} T^{\text{avg}}(n-2). \end{aligned}$$

Finally, we show $T^{\text{avg}}(n) \leq 2 \log n$ by induction.

Clearly, this holds for $n \leq 2$.

So, assume $n \geq 3$. Assume the inductive hypothesis. Then see that

$$\begin{aligned} T^{\text{avg}}(n) &\leq 1 + \frac{1}{2} T^{\text{avg}}(\lfloor \frac{n}{2} \rfloor) + \frac{1}{2} T^{\text{avg}}(n-2) \\ &\leq 1 + \frac{1}{2} (2 \log(\lfloor \frac{n}{2} \rfloor)) + \frac{1}{2} (2 \log(n-2)) \\ &\leq 1 + \log(n-1) + \log(n) \\ &= 2 \log(n), \end{aligned}$$

which suffices to prove the claim. \blacksquare

EXPECTED-CASE RUNTIME: $T_A^{\text{exp}}(n)$

The "expected-case runtime" of an algorithm is defined to be

$$T_A^{\text{exp}}(I) = E[T_A(I, R)] = \sum_{\text{all } R} T_A(I, R) P(R)$$

where R is a sequence of random outcomes, and $P(R)$ denotes the probability the random variables in the algorithm A have outcomes R .

AMORTIZED RUN-TIME: T_0^{amort}

Let \mathcal{O} be an operation, and let $T^{\text{actual}}(\mathcal{O})$ be the actual run-time of \mathcal{O} .

Then, we say \mathcal{O} has "amortized run-time $T^{\text{amort}}(\mathcal{O})$ " if for any sequence of operations $\mathcal{O}_1, \dots, \mathcal{O}_k$ that could occur, we have

$$\sum_{i=1}^k T^{\text{actual}}(\mathcal{O}_i) \leq \sum_{i=1}^k T^{\text{amort}}(\mathcal{O}_i).$$

POTENTIAL FUNCTION [OF A DATA STRUCTURE]: Φ

A "potential function" is a function $\Phi(\cdot)$ that depends on the status of the data structure.

In particular, for any sequence $\mathcal{O}_1, \dots, \mathcal{O}_k$ of operations:

- ① $\Phi(i) \geq 0 \quad \forall i \geq 0$, where Φ_i is the value of Φ after $\mathcal{O}_1, \dots, \mathcal{O}_i$ have been executed; &
- ② $\Phi(0) = 0$.

$$T^{\text{amort}}(\mathcal{O}) := T^{\text{actual}}(\mathcal{O}) + \Phi_{\text{after}} - \Phi_{\text{before}}$$

UPPER BOUNDS ACTUAL RUN-TIME

For any potential function Φ , the function

$$T^{\text{amort}}(\mathcal{O}) := T^{\text{actual}}(\mathcal{O}) + \Phi_{\text{after}} - \Phi_{\text{before}}$$

upper-bounds the actual run-time,

where Φ_{before} & Φ_{after} denote the state of the potential function before & after \mathcal{O} .

Proof. Fix a sequence of operations $\mathcal{O}_1, \dots, \mathcal{O}_k$.

Summing up the amortized times and using a telescoping sum, we get

$$\begin{aligned} \sum_{i=1}^k T^{\text{amort}}(\mathcal{O}_i) &= \sum_{i=1}^k (T^{\text{actual}}(\mathcal{O}_i) + \Phi(i) - \Phi(i-1)) \\ &= \sum_{i=1}^k T^{\text{actual}}(\mathcal{O}_i) + \sum_{i=1}^k (\Phi(i) - \Phi(i-1)) \\ &= \sum_{i=1}^k T^{\text{actual}}(\mathcal{O}_i) + \Phi(k) - \Phi(0) \\ &\geq \sum_{i=1}^k T^{\text{actual}}(\mathcal{O}_i). \quad \blacksquare \end{aligned}$$

STEPS TO PERFORM AMORTIZED ANALYSIS USING THE POTENTIAL FUNCTION METHOD

To do amortized analysis using potential functions, we do the following:

- ① Define a "time unit", so that an operation with run-time $\Theta(k)$ takes at most k time units.
- ② Define a potential function Φ and verify $\Phi(0) = 0$ & $\Phi(i) \geq 0 \quad \forall i \geq 0$.
- ③ For each operation \mathcal{O} , compute

$$T^{\text{amort}}(\mathcal{O}) := T^{\text{actual}}(\mathcal{O}) + \Phi_{\text{after}} - \Phi_{\text{before}},$$

and find an asymptotic upper bound for it.

EXAMPLE: DYNAMIC ARRAYS

Consider dynamic arrays with two operations:

- ① insert; &
- ② rebuild, where we "lengthen" the array by a factor of 2.

Hence, insert takes $\Theta(1)$ time, & rebuild takes $\Theta(n)$ time (where n is the size of the array).

Define a "time unit" such that insert takes "one unit of time" & rebuild takes " n units of time".

We claim $T^{\text{amort}}(\text{insert}) = 3$ & $T^{\text{amort}}(\text{rebuild}) = 0$.

Proof. Let the potential function Φ be defined by $\Phi(i) = \max\{0, 2 \cdot \text{size} - \text{capacity}\}$.

Clearly $\Phi(i) \geq 0$. Also initially size=0 & cap=0, so $\Phi(0) = 0$ as desired.

Now, the amortized run-time for insert is $T^{\text{amort}}(\text{insert}) = T^{\text{actual}}(\text{insert}) + \Phi_{\text{after}} - \Phi_{\text{before}} \leq 3$, as the actual time is ≤ 1 unit, the size increases by 1 & the capacity does not change.

Similarly,
 $T^{\text{amort}}(\text{rebuild}) = T^{\text{actual}}(\text{rebuild}) + \Phi_{\text{after}} - \Phi_{\text{before}} \leq n + (0-n) = 0. \quad \blacksquare$

Chapter 2: Priority Queues and Heaps

ADT PRIORITY QUEUES

Q₁: A "priority queue" stores items that have a priority, or key.

Q₂: Operations:

- ① insert (a given item & priority as a k-v pair)
- ② deleteMax (return item with largest priority)
- ③ size, isEmpty

Q₃: We can use a PQ to sort:

PQ-Sort (A[0, ..., n-1])

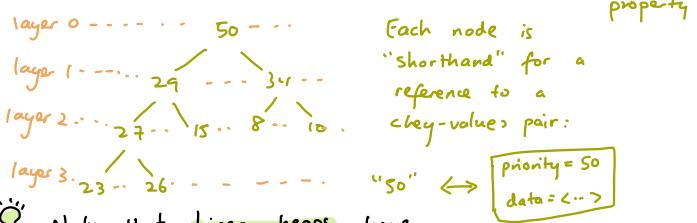
```
init PQ to an empty PQ  
for i=0 to n-1 do  
    PQ.insert(A[i])  
for i=n-1 down to 0 do  
    A[i] ← PQ.deleteMax()
```

Q₄: The run-time of the above algorithm is O(initialization + n·insert + n·deleteMax).

BINARY HEAPS

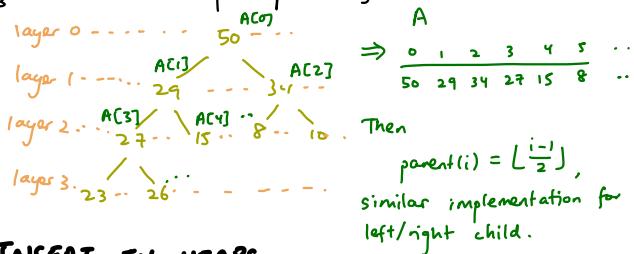
Q₁: "Binary heaps" are binary trees with the following properties:

- ① Each level is filled except the last, which is filled from the left; } "structural" property
- ② key(i) ≤ key(parent(i)) } "heap-order property"



Q₂: Note that binary heaps have height $O(\log n)$.

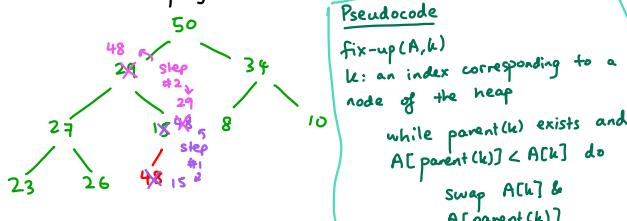
Q₃: We store binary heaps using an array:



INSERT IN HEAPS

Q₁: To insert into a heap, we just place the new key at the first free leaf.

Q₂: We also employ "fix-up":



Pseudocode

fix-up(A, k)

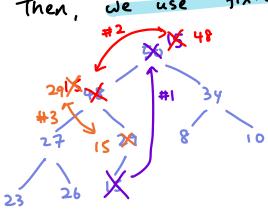
k: an index corresponding to a node of the heap
while parent(k) exists and $A[\text{parent}(k)] < A[k]$ do
 swap $A[k]$ & $A[\text{parent}(k)]$
 $k \leftarrow \text{parent}(k)$

DELETMAX IN HEAPS

The maximum item of a heap is just the root node.

We replace the root by the last leaf, which is taken out.

Then, we use "fix-down":



* we "swap down" from the root-node until the heap-ordering is satisfied.

Run-time: $O(\text{height}) = O(\log n)$

HEAPSORT

If we use a heap as a PQ and sort using it, the run-time of said algorithm is

$$\begin{aligned} T(n) &\in O(\text{initialization} + n \cdot \text{insert} + n \cdot \text{deleteMax}) \\ &= O(\Theta(1) + n \cdot O(\log n) + n \cdot O(\log n)) \\ &= O(n \log n). \end{aligned}$$

Pseudocode:

HeapSort(A, n)

```
// heapify
n ← A.size()
for i ← parent(last()) down to 0 do
    fix-down(A, i, n)
// repeatedly find maximum
while n > 1
    // 'delete' maximum by moving to end and
    // decreasing n
    swap item at A[root()] and A[last()]
    n --
    fix-down(A, root(), n)
```

* Heapify:

Given: all items that should be on the heap

It builds the heap all at once.

How? → by fixing-down incrementally.

Heapify has run-time $\Theta(n)$.

Why? → analyze a recursive version:

```
heapify(node i)
    if i has left child
        heapify(left child i)
    if i has right child
        heapify(right child i)
    fix-down(i)
```

Now, let

$T(n)$:= runtime of heapify on n items.

(Assume n divisible as needed.)

Then:

$$\text{size(left child)} = \text{size(right child)} = \frac{n-1}{2}.$$

$$\therefore T(n) = \begin{cases} \Theta(1), & n \leq 1 \\ T\left(\frac{n-1}{2}\right) + T\left(\frac{n-1}{2}\right) + \Theta(\log n), & \text{otherwise} \end{cases}$$

$$\Rightarrow T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(\log n) \in O(n).$$

Note this uses $O(1)$ auxiliary space, since we use the same input-array A for storing the heap.



OTHER PQ OPERATIONS

We can also support the following operations:

① "findMax" — finds the max element without removing it;

- in a bin heap this takes $\Theta(1)$

- since it is just the root node

② "decreaseKey" — takes in a ref i to the location of one item of the heap and a key k_{new} , and decreases the key of i to k_{new} if $k_{\text{new}} < \text{key}(i)$

- does nothing if $k_{\text{new}} \geq \text{key}(i)$

- easy to do in a bin heap; just need to change the key & then call fix-down on i to its children

③ "increaseKey" — "opposite" of decreaseKey.

- easy in bin heap, but just call fix-up instead of fix-down

④ "delete" — delete the item i (which we have a ref to).

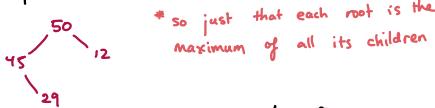
- for bin heap: increase value of key at i to ∞ (or $\text{findMax}().\text{key}() + 1$);

- then call deleteMax

- takes $O(\log n)$

MELDABLE HEAP

A "meldable heap" is the same as a "binary heap", except it drops the structural property.



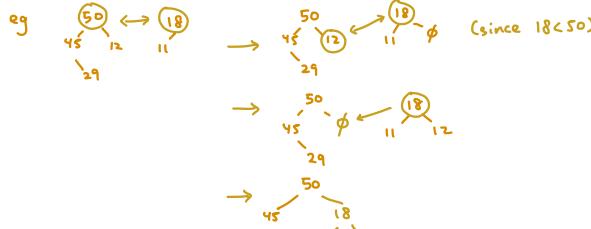
Operations for meldable heaps:

- ① insert
 - create 1-node meldable heap P' w/ new k-v pair we want to add
 - call $\text{merge}(P, P')$ w/ existing & newly-created meldable heap

- ② deleteMax
 - max is root, so just remove that
 - then return $\text{merge}(P_L, P_R)$, where P_L & P_R are the "subheaps" of the original heap

③ merge

```
meldableHeap::merge(r1, r2)
input: r1, r2 (roots of two meldable heaps)
      - ri ≠ NIL
output: returns root of merged heap
if r2 is NIL then return r1
if r1.key < r2.key then swap r1, r2
randomly pick one child c of r1
replace c by result of merge(r2, c)
return r1
```



$$\text{AVG. RUN-TIME (MERGE)} = O(\log n_1 + \log n_2)$$

(L2.3)

Note that

avg run-time (merge) = $O(\log n_1 + \log n_2)$
where n_1, n_2 are the sizes of the heaps to be merged.

BINOMIAL HEAPS

FLAGGED TREES

- 1 A "flagged tree" is one where every level is full, but the root node only has a left child.
- 2 Note that a flagged tree of height h has 2^h nodes.



BINOMIAL HEAP (C02.3)

A "binomial heap" is a list L of binary trees such that

- any tree in L is a flagged tree (structural property); &
- for any node v , all keys in the left subtree of v are no bigger than $v.key$. (order property).



PROPER BINOMIAL HEAP

- We say a BH is "proper" if no two flagged trees in L have the same height.



A PBH OF SIZE n CONTAINS $\leq \log(n)+1$

FLAGGED TREES (C02.2)

Let a PBH have size n .

Then it contains at most $\log(n)+1$ flagged trees.

Proof: Let T be the tree w/ max height, say h , in the list L of flagged trees.

Then T has 2^h nodes, so $2^h \leq n$, ie $h \leq \log(n)$.

Since the trees in L have distinct heights, we have at most one tree for each height $0, \dots, h$, and so at most $h+1 \leq \log(n)+1$ trees. \square

MAKING A BH PROPER

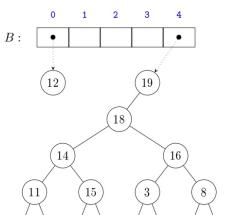
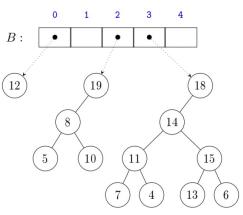
- To make a BH proper, we do the following: if the BH has two flagged trees T, T' of the same height h , then they both have 2^h nodes.

We want to combine them into one flagged tree of height $h+1$.

To do so, we use the following algorithm:

(A2.1)

```
binomialHeap::makeProper()
n <-- size of the binomial heap
for (l=0; n>1; n <-> [n/2]) do l++ // compute [log n]
B <- array of size l+1, initialized at NIL
L <- list of flagged trees
while L is non-empty do
    T <- L.pop(), h <- T.height
    while T' <- B[h] is not NIL do
        if T.root.key < T'.root.key then swap
        T & T'
        T'.right <- T.left, T.left <- T', T.height <- h+1 // merge T with T'
        B[h] <- NIL, h++
    B[h] <- T
for (h=0; h<l; h++) do
    if B[h] ≠ NIL then L.append(B[h]) // copy B back to the list
```



PQ OPERATIONS FOR PBHS

Each of the PQ operations can be performed in $O(\log n)$ time with makeProper.

① merge(P_1, P_2)

- concat lists of P_1, P_2 into one
- then call makeProper
- takes time $O(\log n_1 + \log n_2) \leq O(\log n)$

② insert(k, v)

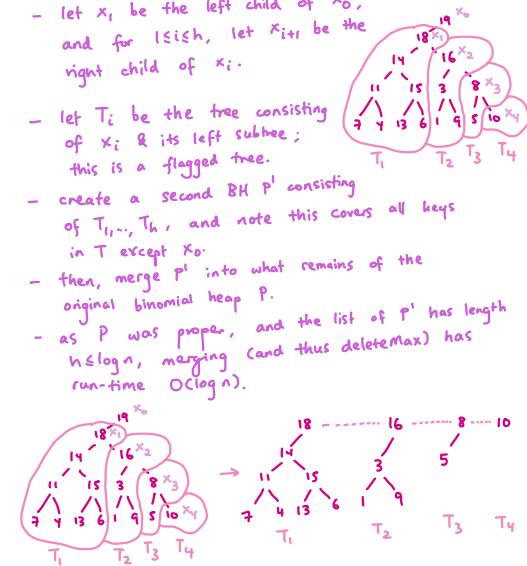
- like w/ meldable heaps: create a single-node binomial heap, then call merge
- takes time $O(\log n)$

③ findMax()

- Scan through L and compare keys of the roots
- largest is just maximum
- takes time $O(|L|) \leq O(\log n)$

④ deleteMax()

- assume we found the max at root x_0 of flagged tree T , say w/ height h (this takes $O(\log n)$ time if heights not already known)
- remove T from L and split it as follows:
 - let x_1 be the left child of x_0 ,
 - and for $i \in [1:h]$, let x_{i+1} be the right child of x_i .
- let T_i be the tree consisting of x_i & its left subtree; this is a flagged tree.
- create a second BH P' consisting of T_1, \dots, T_h , and note this covers all keys in T except x_0
- then, merge P' into what remains of the original binomial heap P .
- as P was proper, and the list of P' has length $h \leq \log n$, merging (and thus deleteMax) has run-time $O(\log n)$.



Chapter 3:

Sorting

THE SELECTION PROBLEM

The "selection problem" is:

"Given an array $A[0, \dots, n-1]$ and an index $0 \leq k \leq n$, $\text{select}(A, k)$ should return the element in A that would be at index k if we sorted A ".

e.g. if $A[0, \dots, 9] = [30, 60, 10, 0, 50, 80, 90, 10, 40, 70]$

then the sorted array would be

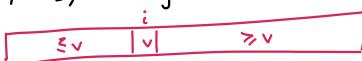
$[0, 10, 10, 30, 40, 50, 60, 70, 80, 90]$

so

$\text{select}(3) = 30$.

PARTITION [THE FUNCTION]

The "partition" function does the following:
given the pivot-value $v = A[p]$ and an array $A[0, \dots, n-1]$, rearrange A s.t.



and return the pivot-index i .

Partition algorithm ("efficient in-place partition — Hoare"):

```
partition(A, p)
// A: array of size n
// p: integer s.t.  $0 \leq p \leq n$ 
1. swap(A[n-1], A[n])
2. i ← 1, j ← n-1, v ← A[n-1]
3. loop
4.   do i ← i+1 while  $A[i] < v$ 
5.   do j ← j-1 while  $j \geq 1$  &  $A[j] > v$ 
6.   if i ≥ j then break (goto 9)
7.   else swap(A[i], A[j])
8. end loop
9. swap(A[n-1], A[i])
10. return i
```

* we keep swapping the outer-most wrongly-positioned pairs

QUICK-SELECT

The "quick-select" algorithm:

```
quick-select(A, k)
// A: array of size n
// k: integer s.t.  $0 \leq k \leq n$ 
1. p ← choose-pivot(A) // for now, p=n-1
2. i ← partition(A, p)
3. if i=k then
4.   return A[i]
5. else if i < k then
6.   return quick-select(A[0, ..., i-1], k)
7. else if i > k then
8.   return quick-select(A[i+1, ..., n-1], k-i-1)
```

* intuition:

Have: $\begin{array}{c|c} \leq v & | v | \\ \hline \end{array}$

Want: $\begin{array}{c|c} \leq m & | m | \\ \hline \end{array}$

* Run-time analysis of quick-select:

We analyze the # of key-comparisons.
↳ so we don't mess with constants.

In particular, partition uses n key comparisons.

Then, the run-time on an instance I is

$$T(I) = \underbrace{n + T(I')}_\text{partition subarray}$$

How big is I' ?

best case: don't recurse at all $\rightarrow O(n)$

worst case: $|I'| = n-1$

$$\begin{aligned} \therefore T^{\text{worst}}(n) &= \max_I T(I) \\ &= n + T^{\text{worst}}(n-1) \\ &= n + (n-1) + T^{\text{worst}}(n-2) \\ &= \dots \\ &= \frac{n(n+1)}{2} \in O(n^2). \end{aligned}$$

Average-case:

$$\begin{aligned} T^{\text{avg}}(n) &= \frac{1}{|\mathcal{X}_n|} \sum_{I \in \mathcal{X}_n} T(I) \\ &= \frac{1}{n} \sum_{k=0}^{n-1} \frac{1}{n!} \sum_{\pi} T(\pi, k). \end{aligned}$$

We will do

- ① Creating a random quick select;
- ② Analyze exp. runtime; then
- ③ Argue that this implies bound on avg-case of quick-select.

RANDOMIZED QUICK-SELECT

Consider the "randomized quick-select" algorithm:

```
quick-select(A, k)
  // A: array of size n
  // k: integer s.t. 0 <= k < n
  1. p ← random(n) // key step
  2. i ← partition(A, p)
  3. if i = k then
  4.   return A[i]
  5. else if i > k then
  6.   return quick-select(A[0, ..., i-1], k)
  7. else if i < k then
  8.   return quick-select(A[i+1, i+2, ..., n-1], k-i+1)
```

Then, what is $P(\text{pivot-index} = i)$?

\Rightarrow pivot-value is equally likely to be any of $A[0], \dots, A[n-1]$

$\Rightarrow \therefore$ pivot-index is equally likely to be any of $0, \dots, n-1$

$$\Rightarrow P(\text{pivot index} = i) = \frac{1}{n}$$

We claim $T^{\text{exp}}(n) \in O(n)$.

Proof. Recall that

$$T^{\text{exp}}(n) = \max_I \sum_{\substack{\text{random} \\ \text{outcomes } R}} P(R) \cdot T(I, R)$$

In particular, note that

$$T(I, R) = \begin{cases} n + T(A[0, \dots, i-1], k, R') & i > k \\ n + T(\text{right subarray}, k-i-1, R'), & i < k \\ n & i = k \end{cases}$$

(we count # of comparisons)

Then,

$$\begin{aligned} \sum_R P(R) T(I, R) &= \sum_R P(R) T(A, k, R) \\ &= \sum_{(i, R')} \underbrace{P(i)}_{\frac{1}{n}} \underbrace{P(R')}_{\sum_{R'} P(R')} \underbrace{T(A, k, R')}_{\begin{cases} n + \dots & i > k \\ \dots & i < k \\ 0 & i = k \end{cases}} \\ &= n + \frac{1}{n} \sum_{i=0}^{n-1} \left\{ \begin{array}{l} \sum_{R'} P(R') T(A_2, k, R') \\ \sum_{R'} P(R') T(A_r, k-i-1, R') \\ 0 \end{array} \right\}_{\begin{array}{l} i > k \\ i < k \\ i = k \end{array}} \\ &\leq n + \frac{1}{n} \sum_{i=0}^{n-1} \left\{ \begin{array}{l} \max_{k'} \max_{A_2} \sum_{R'} P(R') T(A_2, k', R') \\ \max_{k'} \max_{A_r} \sum_{R'} P(R') T(A_r, k', R') \\ 0 \end{array} \right\}_{\begin{array}{l} i > k \\ i < k \\ i = k \end{array}} \\ &= n + \frac{1}{n} \sum_{i=0}^{n-1} \left\{ \begin{array}{l} T^{\text{exp}}(i) \\ T^{\text{exp}}(n-i-1) \\ 0 \end{array} \right\}_{\begin{array}{l} i > k \\ i < k \\ i = k \end{array}} \\ &\leq n + \frac{1}{n} \sum_{i=0}^{n-1} \max_i \{ T^{\text{exp}}(i), T^{\text{exp}}(n-i-1) \} \end{aligned}$$

Then, we show $T^{\text{exp}}(n) \in 8n$.

Proof. By induction.

$$n=1: \text{Comp} = 0 < 8(1) = 8.$$

Step:

$$T(n) \leq n + \frac{1}{n} \sum_{i=0}^{n-1} \max_i \{ 8i, 8(n-i-1) \}$$

bad	good	bad
$n/4$		$3n/4$

$$\begin{aligned} &\leq n + \frac{1}{n} \sum_{i \text{ good}} 8\left(\frac{3}{4}n\right) + \frac{1}{n} \sum_{i \text{ bad}} 8n \\ &= n + \frac{1}{n} \cdot \frac{n}{2}(6n) + \frac{1}{n} \cdot \frac{n}{2}(8n) \\ &= n + 3n + 4n = 8n. \quad \blacksquare \end{aligned}$$

$\therefore T^{\text{exp}}(n) \in O(n)$. \blacksquare

QUICK-SORT

Pseudocode:

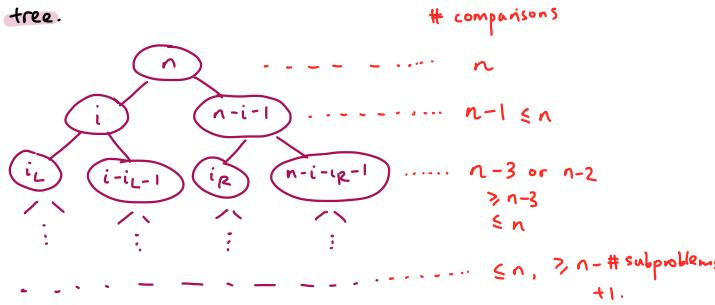
```
quick-sort(A)
  // A: array of size n
  1. if n ≤ 1 then return
  2. p ← choose-pivot(A)
  3. i ← partition(A, p)
  4. quick-sort(A[0, ..., i-1])
  5. quick-sort(A[i+1, ..., n-1])
```

There are better implementations of this idea.

Runtime:

$$T(n) = n + T(i) + T(n-i-1).$$

But there's a simpler method: the recursion tree.



Thus

$$\# \text{ comparisons} \leq n \cdot \# \text{ layers} = n \cdot \text{height of the recursion tree.}$$

So what can we say about the recursion tree's height?

① Worst case: height = $\Theta(n)$

- * tight if array is sorted.
- we need n recursions.

Thus run-time $\in \Theta(n^2)$.

② Best case: if the pivot-index is $\sim \frac{n}{2}$ always.



Thus run-time $\in \Theta(n \log n)$.

We can improve QuickSort's run-time by

① Not passing sub-arrays, but instead passing "boundaries";

② Stopping recursion early;

- stop recursing when array of subproblem is ≤ 10
- then use insertion-sort to sort the array
- which has $\Theta(n)$ best-case run time if the array is (almost) sorted

③ Avoid recursions;

④ Reduce auxiliary space;

- in the worst case (currently), the auxiliary space is $|S| \in \Theta(n)$.
- but if we put the bigger subproblem on the stack, the space can be reduced to $|S| \in O(\log n)$.

⑤ Choose the pivot-index efficiently;

- don't let $p=n-1$. (run-time = $\Theta(n^2)$)
- use "median-of-3": use the median of $\{A[0], A[\frac{n}{2}], A[n-1]\}$ as the pivot-value

AVERAGE-CASE QUICK-SORT

① We accomplish this via randomization of the algorithm:

```
RandQS(A)
  // A: array of size n
  1. if n ≤ 1 then return
  2. p ← random(n)
  3. i ← partition(A, p)
  4. quick-sort(A[0, ..., i-1])
  5. quick-sort(A[i+1, ..., n-1])
```

② Then,

$$T^{\exp}(n) = \exp \# \text{ of comparisons of the algo.}$$

③ Note that

$$T(A, R) = n + T(\text{left subarray, } R') + T(\text{right subarray, } R').$$

↑ instance ↑ outcomes
size i size $n-i$

④ Hence

$$\begin{aligned} \sum_R P(R) T(A, R) &= n + \frac{1}{n} \sum_{i=0}^{n-1} T^{\exp}(i) + \frac{1}{n} \sum_{i=0}^{n-1} T^{\exp}(n-i-1) \\ &= n + \frac{2}{n} \sum_{i=2}^{n-1} T^{\exp}(i), \end{aligned}$$

and so

$$T(n) = \begin{cases} 0, & n \leq 1 \\ n + \frac{2}{n} \sum_{i=2}^{n-1} T(i), & \text{otherwise.} \end{cases}$$

⑤ We claim $T(n) \in O(n \log n)$, so the expected run-time of RandQS is in $O(n \log n)$, and so the average-case run time of QS is in $O(n \log n)$.

Proof. Specifically, we prove $T(n) \leq 2 \cdot n \cdot \ln(n)$.

By induction. Base ($n=1$) is trivial.

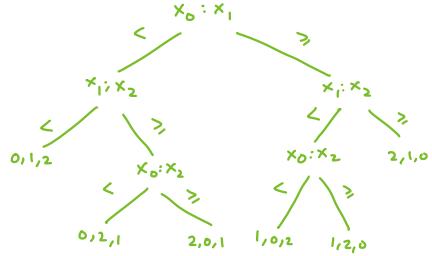
$$\begin{aligned} \text{Step: } T(n) &\leq n + \frac{2}{n} \sum_{i=2}^{n-1} 2i \ln(i) \\ &= n + \frac{4}{n} \sum_{i=2}^{n-1} i \ln(i) \\ &\leq n + \frac{4}{n} \int_2^n x \ln x \, dx \quad (\text{since } x \ln x \text{ is increasing}) \\ &\leq n + \frac{4}{n} \left(\frac{1}{2} n^2 \ln(n) - \frac{1}{4} n^2 \right) \\ &= n + 2n \ln(n) - n \\ &= 2n \ln(n). \quad \blacksquare \end{aligned}$$

ANY COMPARISON BASED SORTING ALGORITHM USES $\Omega(n \log n)$ KEY-COMPARISONS IN THE WORST-CASE

 As above.

Proof. Fix an arbitrary comparison-based sorting algorithm A , and consider how it sorts an instance of size n .

Since A uses only key-comparisons, we can express it as a decision tree T.



A decision tree for an algo to sort 3 elements x_0, x_1, x_2 with ≤ 3 key-comparisons.

For each sorting perm π , let I_π be an instance that has distinct items and sorting perm π (ie $I_\pi = \pi^{-1}$).

Executing A on I_{π} leads to a leaf that stores π .

Note that no two sorting perms can lead to the same leaf, as otherwise the output would be incorrect for one (as the items are distinct).

We then have $n!$ sorting perms, and hence at least $n!$ leaves that are reached for some π .

Let h be the largest layer-number of a leaf reached by some I_{π} .

Since γ is binary, for any $0 \leq h$ there are at most 2^h leaves in layers $0, \dots, h$.

Therefore $2^n > n!$, or

$$\begin{aligned}
 n \geq \log(n!) &= \log(n(n-1) \cdots 1) \\
 &= \log(n) + \log(n-1) + \cdots + \log(1) \\
 &\geq \log(n) + \log(n-1) + \cdots + \log(\lceil \frac{n}{2} \rceil) \\
 &> \log\left(\frac{n}{2}\right) + \log\left(\frac{n}{2}\right) + \cdots + \log\left(\frac{n}{2}\right) \\
 &\geq \frac{n}{2} \log\left(\frac{n}{2}\right) \\
 &= \frac{n}{2} \log(n) - \frac{n}{2} \in \mathcal{L}(n \log n).
 \end{aligned}$$

Finally, consider the sorting perm π that has its leaf on layer h .

Executing algorithm A on Π hence takes $h \in \Omega(n \log n)$ key-comparisons, so the worst-case bound holds. \square

SORTING INTEGERS

BUCKET-SORT

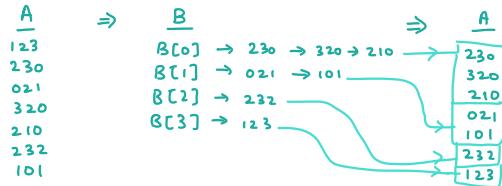
Bucket-sort can be used to sort a collection of integers by a specific "position" of digit.

eg the last digit 12345

Pseudocode:

```
bucket-sort(A, n < A.size, l <= 0, r <= n-1, d)
// A: array of size  $\geq n$  with numbers with m digits in {0,...,R-1}
// d: 1st digit
// output: A[l...r] is sorted by "d" digit.
1. initialize an array B[0...R-1] of empty lists
2. for i < l to r do
3.   append A[i] at the end of // move array-items
      B[dth digit of A[i]] to buckets
4. i < l
5. for j < 0 to R-1 do // move bucket-items
6.   while B[j] is not empty do to array
7.     move first element of B[j]
       to A[i]
8.   i += 1
```

eg



See that

- ① Run-time = $\Theta(n+R)$; and
- ② Auxiliary space = $\Theta(n+R)$.

MSD-RADIX-SORT

MSD (Most Significant Digit) radix sort can be used to sort multi-digit numbers.

Pseudocode:

```
MSD-radix-sort(A, n < A.size, l <= 0, r <= n-1, d=1)
// A: array of size  $\geq n$ , contains numbers with m digits in {0,...,R-1}, m, R are global variables
// l, r: range (ie A[l...r]) we wish to sort
// d: digit we wish to sort by
1. if l < r then
2.   bucket-sort(A, n, l, r, d)
3.   if d < m then
        // find sub-arrays that have the same dth digit and recurse
4.     int l' < l
5.     while l' < r do
6.       int r' < r'
7.       while r' < r & dth digit of A[r'+1] = dth digit of A[l'] do r'++
8.       MSD-radix-sort(A, n, l', r', d+1)
9.     l' += 1
```

Note that

- ① run-time = $\Theta(mRn)$; &
- ② auxiliary space = $\Theta(n+R+m)$.

LSD-RADIX-SORT

"LSD-Radix-Sort" is a better way of sorting multi-digit numbers (than MSD) because

- ① it has a faster runtime;
- ② it uses less auxiliary space; and
- ③ it uses no recursion.

Pseudocode:

```
LSD-radix-sort(A, n < A.size)
// A: array of size n, contains m-digit radix-R numbers, m, R are global
1. for d=m down to 1 do
2.   bucket-sort(A, d)
```

Clearly

- ① run-time = $\Theta(m(n+R))$; &
- ② auxiliary space = $\Theta(n+R)$.

Chapter 4: ADT Dictionaries

DICTIONARIES

💡 Dictionaries store key-value pairs, or "KVP".



In particular,

- ① `search(key)`: return the KVP for this key.
- ② `insert(key, value)`: add the KVP to the dictionary.
 - key is distinct from existing keys
- ③ `delete(key)`: remove KVP with this key.

💡 Assumptions:

- ① we assume all keys are distinct.
- ② we also assume keys can be compared.

💡 Implementations:

① **Unsorted list**

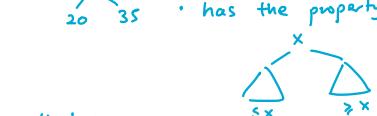
- fast insert, slow search, slow delete

② **Sorted array**

- fast search, slow insert, slow delete

③ **Binary search tree / BST**

- we only show the keys (implied each node is a KVP)
- has the property



In particular,

`insert, delete, search` $\in O(\text{height of tree})$.

* Height is $\Theta(n)$ is worst case, but typically much better ($O(\log n)$)

LAZY DELETION

💡 Consider a sorted array:

`delete(20)`:

10 20 30 40 60 70 80

- usually takes $\Theta(n)$ time to "backtrack" all other elements
- but we can avoid this if we instead just mark the box as "isDeleted".
- thus run-time (`search`) = $O(\log n)$.
- but we don't get any space back!
- however, we can occasionally "clean up":
 - create a new initialization; &
 - move all items into the new array if they were "real".

· clean-up takes $O(n * \text{insert})$.

But we can frequently do better; in this example, we can perform clean-up in $O(n)$ time.

Then, the amortized time is $O(\text{insert} + \text{delete})$ for doing a deletion, and sometimes better.

💡 Why do we do **lazy deletion**?

- ① It is simpler.
- ② It might be faster.
 - for sorted arrays: $\Theta(n)$ worst-case for delete but $\Theta(\log n)$ amortized time for lazy deletion

③ It sometimes is required.

💡 Why not?

- ① It wastes space.
 - we have to allocate space to store the "isDeleted" flag.
- ② Occasionally deletion is very slow.

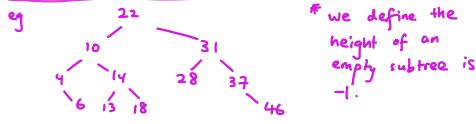
AVL-TREES

Goal: $O(\log n)$ worst case time.

Structural condition:

For any node z , we have

$$\text{balance}(z) = |\text{height}(z.\text{left}) - \text{height}(z.\text{right})| \leq 1.$$



- notice the structural property holds for every node.

We also store the height of the subtree at every node, or we can also store the balance.

(usually we will store the height).

$h \in O(\log n)$

Let h be the height of an AVL tree with n nodes.

Then necessarily $h \in O(\log n)$. # nodes

height 0: 1

height 1: 2

height 2: 4
has $h=1$
to preserve the structural property!

height 3: 7
 $h=2$ $h=1$

In general, let $N(h)$ = smallest # of nodes of

height h .

$N(h) = N(h-1) + N(h-2) + 1$.

height h :

$\therefore N(h) = N(h-1) + N(h-2) + 1$.

Thus

h	0	1	2	3	4	5	...
$N(h)$	1	2	4	7	12	20	...
$N(h)+1$	2	3	5	8	13	21	

We see $N(h)+1$ is the Fibonacci numbers!

(ie $F(0)=0$, $F(1)=1$, $F(i)=F(i-1)+F(i-2)$ $\forall i \geq 2$.)

By a easy induction proof, we can show

$$N(h)+1 = F(h+3).$$

In particular, we know

$$F(i) = \frac{1}{\sqrt{5}} \phi^i + \Theta(1), \quad \phi = \text{the golden ratio}.$$

Therefore

$$N(h) = \frac{1}{\sqrt{5}} \phi^{h+3} + \Theta(1)$$

Hence, for any AVL tree of height h ,

$$\# \text{nodes} = n \geq N(h) \approx \frac{1}{\sqrt{5}} \phi^{h+3},$$

and so

$$n \approx \log_{\phi} (\sqrt{5} n) - 3 \in O(\log n).$$

AVL OPERATIONS

INSERT, PART 1

First, we call `BST::insert`, and then rebalance the ancestors of z .

`AVL::insert(k, v)`

1. $z \leftarrow \text{BST}::\text{insert}(k, v)$
2. while z is not NIL do
3. if $|z.\text{left}.\text{height} - z.\text{right}.\text{height}| > 1$ then
4. let $y \leftarrow$ taller child of z
5. let $x \leftarrow$ taller child of y
6. $z \leftarrow \text{restructure}(x, y, z)$
7. break
8. $z.\text{height} \leftarrow 1 + \max\{z.\text{left}.\text{height}, z.\text{right}.\text{height}\}$ // we alias this as "setHeightFromSubtrees"
9. $z \leftarrow z.\text{parent}$

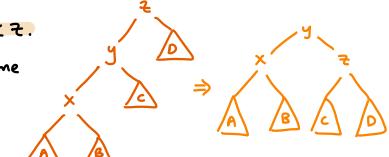
ROTATIONS [OF BSTS]

Let T be a BST with a node z that has a child y and a grandchild x . Then, a "rotation at z with respect to y & x " is a restructuring of T such that the result is again a BST, and sub-tree references have been changed only at x, y, z .

For restoring balances at AVL-trees, we want the four rotations that make the median of x, y, z the new root of the subtree:

① Right rotation; $x < y < z$.

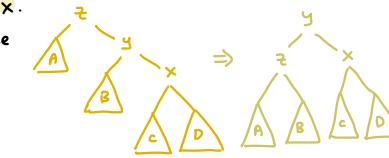
We want y to become the root.
pseudocode similar to left rotation; see below



② Left rotation; $z < y < x$.

We want y to be the root.
rotate-left(z)

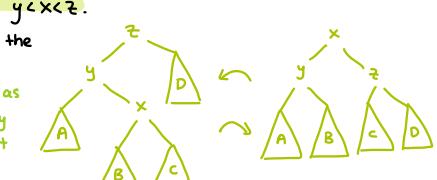
1. $y \leftarrow z.\text{right}$
2. $z.\text{right} \leftarrow y.\text{left}$
3. $y.\text{left} \leftarrow z$
4. `setHeightFromSubtrees(z)`
5. `setHeightFromSubtrees(y)`
6. return y



③ Double-right rotation; $y < x < z$.

We want x to be the root.

can also be written as single-left rotation at y and then a single-right rotation at z (double-rotation)



④ Double-left rotation; $z < x < y$.

We want x to be the root.

can also be written as single-right rotation at y and then a single-left rotation at z (double-rotation)



INSERT, PART 2

We now give the implementation of `restructure` in the `insert` function from the first part;

```
restructure(x, y, z)
  // node x has parent y & grandparent z
  1. if y=z.left & x=y.left    // x-y-z
  2.   return rotate-right(z)  // right rotation
  3. else if y=z.left & x=y.right // y-x-z
  4.   z.left ← rotate-left(y) // y-x
  5.   return rotate-right(z)  // right rotation
  6. else if y=z.right & x=y.left // z-y-x
  7.   z.right ← rotate-right(y) // z-y
  8.   return rotate-left(z)   // double-left rotation
  9. else                      // z-y-x
  10.  return rotate-left(z)  // left rotation
```

DELETE

Pseudocode:

```

AVL::delete(u)
1. z ∈ BST::delete(u) // z is the parent of the
   BST node that was
   removed
2. while z is not NIL do
   if |z.left.height - z.right.height| > 1 then
   let y = taller child of z
   let x = taller child of y
   // (break ties to prefer single rotation)
   6. z ← restructure(x,y,z)
   // do not break — continue up the path &
   rotate if needed.
7. setHeightFromSubtrees(z)
8. z ← z.parent

```

RUNTIME

Both insert & delete have run-time $O(\log n)$.

Why?

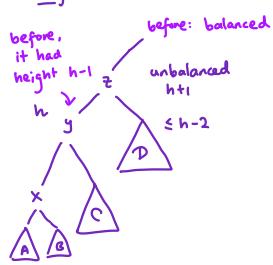
- height of tree = $O(\log n)$
- rotations take $O(1)$ time
- so delete takes $O(\log n)$.
- tight if we insert/delete at lowest level & never rotate.

CORRECTNESS [FOR INSERTION]

If we restructure at z during an insertion, then

- ① the subtree is balanced; and
- ② the subtree has now the height that it had before the insertion.

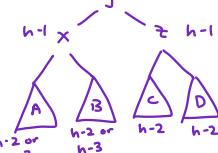
Proof.



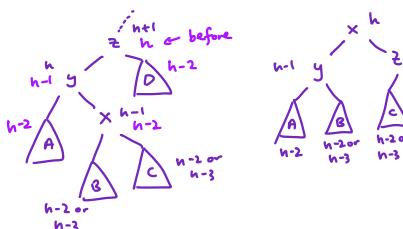
Case 1: "x" is the left node

- x has height $h-2$ before
- c has height $h-2$

After rotation:



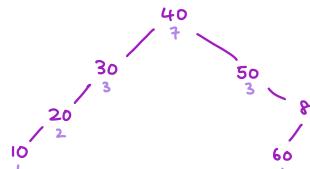
Case 2: x is the right child.



SCAPEGOAT TREES

"Scapegoat trees" are BSTs such that

- ① Each node v stores the size of its subtree; and
- ② $v.size \leq \alpha \cdot v.parent.size$ for all nodes that are not the root.



* Example, with $\alpha = \frac{2}{3}$. Verify that $p.size \geq \frac{3}{2}v.size$ for any parent p of any node v .

HEIGHT = $O(\log n)$

Any scapegoat tree has height $O(\log n)$.

Proof: At any leaf v (which has size 1), the parent has size $\geq \frac{1}{\alpha}$ by defn of a scapegoat tree.

Repeating this argument, the grand-parent has size $\geq (\frac{1}{\alpha})^2$, and so on. As the root has size n , it follows that

$$(\frac{1}{\alpha})^d \leq n,$$

where $d = \max \text{ depth of a leaf} = \text{height}.$
Thus

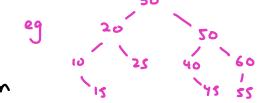
$$d \leq \log_{(\frac{1}{\alpha})}(n) \in O(\log n).$$

PERFECTLY BALANCED BST

A "perfectly balanced BST" is one where for any node v , we have

$$|v.left.size - v.right.size| \leq 1.$$

Given any n -node BST T , we can build a perfectly balanced BST with the same KVPs in $O(n)$ time.



INSERT

Pseudocode:

```

ScapegoatTree::insert(k,v)
1. z ∈ BST::insert(k,v)
2. S ← stack initialized w/ z
3. while p ≠ z.parent ≠ NIL do
4.   p.size++
5.   S.push(p)
6.   z = p
7. while S.size ≥ 2 do
8.   p = S.pop()
9.   if p.size <  $\frac{1}{\alpha} \max\{p.left.size, p.right.size\}$  then
10.    completely rebuild the subtree rooted
        at p as a perfectly balanced BST
11.    break

```

insert has worst case runtime $O(n)$, but amortized runtime $O(\log n)$.