# Theory of Algorithms

1-5 *math.scu.edu/~bwalden/alg

★algorithm: a sequence of steps that solves a problem/does something for you ★
↑finite

*analysis of algorithms
- does it work?
- how well does it work?
- how fast? (should mean fewer steps)
  - most basic notion of "faster" is fewer steps
  - big O differences matter

*Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
$F_n = F_{n-1} + F_{n-2}$ (recursive)

1-7 *Big O Notation
- *$f(n) = O(g(n))$
- *$f(n)$, $g(n)$ are fcns whose input variable is generally a positive integer that (mostly) measures the problem, and generally have an integer value output
- typically $f(n)$ is # of steps for a certain problem w/ input of size $n$, and $g(n)$ would be # of steps for a different algorithm
  - problem: it's possible to get different outputs from same $n$
- Write $f(n) \leq g(n)$ to say one has fewer steps of the other
  - to allow for big O, add constant multiple: $f(n) \leq Cg(n)$
  - take out all small cases: $f(n) \leq Cg(n)$, with $n \geq N$
  - ★If there exists constants $C$ and $n$ such that $f(n) \leq Cg(n)$ whenever $n \geq N$, then we say $f(n) = O(g(n))$★
    ↑eventually ↑essentially

- Ex: $f(n) = \sqrt{n^3 + 10^9}$ $g(n) = 0.001n^2$

  $n=1$ 30,000 .001 $\longrightarrow f(n) = O(g(n))$

  $\sqrt{n^3 + 10^9} \leq C(0.001n^2)$

  if $n > 1000$, $n^3 > 10^9 \rightarrow \sqrt{n^3 + 10^9} \leq \sqrt{n^3 + n^3} \Rightarrow \sqrt{2}n^{3/2} \leq \sqrt{2}n^2 \rightarrow \sqrt{2} \cdot 1000\, g(n)$

  so $C = \sqrt{2} \cdot 1000$, $n = 1000$

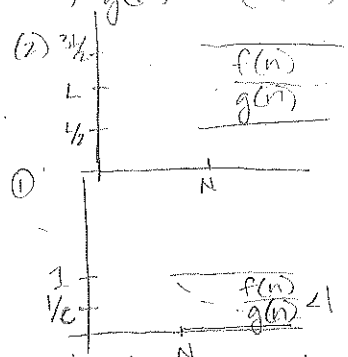
  [End ex]

- Chain of Being: $1$, $\log n$, $n^k$, $a^n$, $n!$

| constant fcns | usually base 2 logarithmic fcns | polynomial/power fcns: $k>0$ | exponential fcns: $a>1$ | factorial |
|---|---|---|---|---|

- in the list above, any fcn $f(n)$ to the left of any fcn $g(n)$ satisfies $f(n) = O(g(n))$ but $g(n) \neq O(f(n))$
  - if $g(n) \neq O(f(n))$, $f(n)$ is distinctly smaller to $g(n)$
- Notation: $f(n) = \Omega(g(n))$ means $g(n) = O(f(n))$
  - if both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, we say $f(n) = \Theta(g(n))$
    - cannot/do not find distinction when $\Theta$ b/c too small
- constant fcns: eat single cherry of sundae of all sizes
- $\log n$: much better than $n^k$
- $n^k$: more typical; smaller power is better
- $a^n$: really really big (like recursive Fibonacci)
- $n!$: rare except for combinations & permutations

- lemma - suppose $f(n)$, $g(n)$ satisfy $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L$
  1) if $L = \phi$, $f(n) = O(g(n))$ but $g(n) \neq O(f(n))$
  2) if $(0, \infty)$, $f(n) = \Theta(g(n))$
  3) if $L = \infty$, $g(n) = O(f(n))$ but $f(n) \neq O(g(n))$




$\frac{L}{2} < \frac{f(n)}{g(n)} < \frac{3L}{2} \rightarrow f(n) < \frac{3L}{2} g(n)$

$\frac{L}{2} g(n) < f(n) \rightarrow g(n) < \frac{2}{L} f(n)$

- to show $g(n) \neq O(f(n))$, want to show for any $C$, there are infinitely many cases where $\frac{f(n)}{g(n)} < \frac{1}{C}$ ← negation of big O
  - if $L = \infty$, $\frac{g(n)}{f(n)}$ is $\phi$ and use #1

- Ex: $\lim_{n\to\infty} \frac{1}{\frac{1}{\log n}} = \phi$

1-9

★Last time: $f(n) = O(g(n))$ means $\exists\, c, N$ such that $f(n) \le Cg(n)$ for all $n \ge N$★

- f, g eventually positive functions
- if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L$ and 1) $L = \phi$, then $f(n) = O(g(n))$ but $g(n) \neq O(f(n))$
  2) $0 < L < \infty$, then $f(n) = \Theta(g(n))$
  3) $L = \infty$, then $g(n) = O(f(n))$ but $f(n) \neq O(g(n))$
- If f is to the left of g in the list $(1, \log n, n^k, a^n, n!)$, then $f(n) = O(g(n))$ but $g(n) \neq O(f(n))$. Proof:

$$\lim_{n\to\infty} \frac{\log n}{n^k} = \lim_{x\to\infty} \frac{\log x}{x^k}$$

$$\text{L'H} = \lim_{x\to\infty} \left( \frac{\frac{1}{\ln b}\frac{1}{x}}{k x^{k-1}} \right)$$

$$= \lim_{x\to\infty} \frac{1}{\ln b \, k \, x^k} = \phi \quad \blacksquare$$

Note: $\log_b x = \frac{\ln x}{\ln b}$ ← All $\log n$'s are constant multiples, so doesn't matter

$\frac{d}{dx} \log_b x = \frac{1}{\ln b}\frac{1}{x}$

- Ex: Compare $f(n) = n \log n$, $g(n) = n^2$

$$\frac{f(n)}{g(n)} = \frac{n \log n}{n^2} = \frac{\log n}{n} \to \lim_{n\to\infty} \frac{\log n}{n} = \phi \quad \text{[End ex]}$$

★big O slows down as you move right in list★

- Ex: $\lim_{n\to\infty} \frac{n^k}{a^n} = \lim_{x\to\infty} \frac{x^k}{a^x}$

Note: $\frac{d}{dx} a^x = a^x \ln a$

$$\text{L'H} = \lim_{x\to\infty} \frac{k x^{k-1}}{a^x \ln a}$$

$$\text{L'H} = \lim_{x\to\infty} \frac{k(k-1)x^{k-2}}{a^x (\ln a)^2} \to \frac{x \text{ drops by 1 eventually to } \phi}{a^x} \to \phi$$

$\vdots \Rightarrow (k+1)$ times $\to \phi$

[End ex]

- Ex: $\lim_{n\to\infty} \frac{a^n}{n!}$ Note: $e^a = \sum_{n=0}^{\infty} \frac{a^n}{n!} = 1 + a + \frac{a^2}{2!} + \frac{a^3}{3!} + \ldots \to \phi$ (lazy proof)

$$n! = \underbrace{1 \cdot 2 \cdot 3 \cdot 4 \cdots n}_{\#\, < n/2} \ge \underbrace{1 \cdot 1 \cdot 1 \cdot 1 \cdots}_{\#\, > n/2} \underbrace{\frac{n}{2} \cdot \frac{n}{2} \cdot \frac{n}{2}}_{\#\, > n/2} = \left(\frac{n}{2}\right)^{n/2}$$

$$= \left(\sqrt{\frac{n}{2}}\right)^n$$

When $\sqrt{\frac{n}{2}} > 2a = (2a)^n$,

so $\frac{a^n}{n!} < \frac{1}{2^n} \to \phi$

★How big is $\log n!$ ?

- Prop: $\log n! = \Theta(n \log n)$

$$\log n! \ge \log \left(\frac{n}{2}\right)^{n/2} = \frac{n}{2} \log_2 \frac{n}{2} = \frac{n}{2}(\log_2 n - 1) > \frac{n}{2} \frac{\log_2 n}{2} \quad \text{(for n large)}$$

$n^n \ge n!$

$\log n! = \log n^n = n \log n$

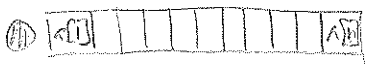
- log functions take massive functions and collapses them

*Sorting an Array:

- input: a[1], ..., a[n]
- output: sorted array (in general, a[1] ≤ a[2] ≤ a[3] ≤ ... ≤ a[n])
- have to be able to compare any two entries and, if we choose, want to be able to swap the two entries
  + can count ① comparison steps and/or ② swaps
    - In general, there are more comparisons than swaps

*Insertion Sort*

+ insert on end of array and compare to previous term
  - don't have to swap if tie
  - *sentinal → a[0] — add something smaller than first entry to avoid checking for out-of-bounds conditions
  - How many steps required?
    + depends on initial array
    + worst case scenario: backwards order

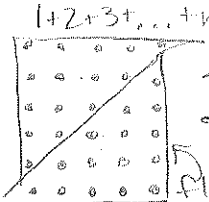


| | Comps | Swaps |
|---|---|---|
| 1 | 1 | 0 |
| 2 | 2 | 1 |
| 3 | 3 | 2 |
| ... | ... | ... |
| n | n | n-1 |

+ need to be able to add up numbers 1 to n

total # of comparisons: 1+2+3+...+n

swaps: 0+1+2+3+...+(n-1)



flip over to make other part

*# comparisons = $\frac{n(n+1)}{2}$*

*# swaps = $\frac{(n-1)n}{2}$ or $\frac{n(n+1)}{2} - n$*

+ both $\frac{n^2+n}{2}$ and $\frac{n^2-n}{2}$ $= \Theta(n^2)$ → ORDER $N^2$*

- In general, if p(n) is a polynomial of degree d,

*$p(n) = a_d n^d + a_{d-1} n^{d-1} + a_{d-2} n^{d-2} + \ldots + a_1 n + a_0$* with $a_d \neq 0$,

then $p(n) = \Theta(n^d)$

+ $\lim_{n\to\infty} \frac{p(n)}{n^d} = \lim_{n\to\infty} \frac{a_d n^d + a_{d-1} n^{d-1} + \ldots + a_1 n + a_0}{n^d}$

$= \lim_{n\to\infty} a_d + \frac{a_{d-1}}{n} + \frac{a_{d-2}}{n^2} + \ldots + \frac{a_0}{n^d}$

*Selection Sort:

- move left to right, looking for smallest entry, then swap with first (then second, third, ..., n-1) spot
- number of comparisons = $\frac{n^2-n}{2}$ → ORDER $n^2$ ★
- number of swaps = n

---

1-12 *Divide-and-Conquer: you have a new problem, split it up and work in smaller pieces, then put back together

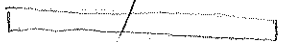
*Merge Sort:

- recursively split in half and sort - sort left half, sort right half, merge together
- when merging problem could happen if two parts are physically swapped
  + rather, after each compare, add to new array, repeat compare using the remaining component

  - requires a lot of extra space
  - need to make sure index of first list doesn't reach used second list indexes. Also index of second list can be exhausted first and grab stuff from space
  + to fix this, sort second half in opposite direction and decrement index

- final definition:
  + sort the left half (smallest to largest) in extra space with merge sort
  + sort the right half (largest to smallest) in extra space with merge sort
  + merge the two halves into one sorted list in original space
- How many steps does merge sort take?
  + worst case = best case because of recursion - independent of initial ordering
  + number of comparisons (and assignments)
    + let T(n) = # comparisons required in a merge sort of an array w/ n entries

$T(n)$ = # comp left + # comp right + # comp merge ($n$)

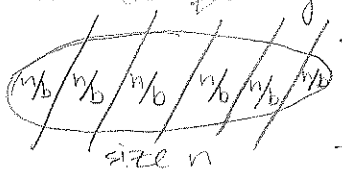


$\lceil n/2 \rceil$ $\lfloor n/2 \rfloor$ $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$

★ $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$, with $T(1) = \emptyset$ ★

- Assuming $n = 2^k$, the recurrence becomes: ★ $T(n) = 2T(n/2) + n$, $T(1) = \emptyset$ ★
  + if $n = 2^k$, arrays will always have even n entries

- divide-and-conquer general




- let b = # of subproblems ($n = b^k$ guarantees you can continue)
- let a = # of times to solve the subproblem, size $\frac{n}{b}$
- Do $T(n/b)$ a times, total of $aT(n/b)$
- let $f(n)$ = number of steps pre/postprocessing

→ $T(n) = aT(n/b) + f(n)$ → depends on sizes of $a, b, f(n)$

- Consider special case: $f(n) = \emptyset$

$T(n) = aT(n/b)$

$T(b^k) = aT(b^{k-1})$

$= a(aT(b^{k-2})) = a^2T(b^{k-2})$

$= a^3T(b^{k-3})$

$\vdots$

$= a^kT(b^0) = a^kT(1)$ for smallest sized subproblem

So $T(b^k) = Ca^k$ with $n = b^k \rightarrow k = \log_b n$

$T(b^k) = Cn^{\log_b a}$

$a^k = a^{\log_b n}$

$a^k = n^{\log_b a} = d$ ★

- Generally, the answer will depend on how big $f(n)$ is

★ MASTER THEOREMS: given the recurrence $T(n) = aT(n/b) + f(n)$,

if 1) $f(n) = O(n^{d-\varepsilon})$, then $T(n) = \Theta(n^d)$ $\quad \varepsilon > \emptyset$

2) $f(n) = \Omega(n^{d+\varepsilon})$, then $T(n) = \Theta(f(n))$ $\quad \Omega$ = more than

3) $f(n) = \Theta(n^d)$, then $T(n) = \Theta(n^d \log n)$ ★

---

1-14 ★ Proofs:

- Case 1 →

$$\frac{T(n)}{n^d} = \frac{aT(n/b) + f(n)}{n^d}$$

$$= \frac{aT(n/b)}{n^d} + \frac{f(n)}{n^d}$$

$$= \frac{b^d T(n/b)}{n^d} + \frac{f(n)}{n^d}$$

$$= \frac{T(n/b)}{(n/b)^d} + \frac{f(n)}{n^d}$$

replace n by n/b → $\frac{T(n/b^2)}{(n/b^2)^d}$ → plug in → $= \frac{T(n/b^2)}{(n/b^2)^d} + \frac{f(n/b)}{(n/b)^d} + \frac{f(n)}{n^d}$

$= \frac{T(n/b^3)}{(n/b^3)^d} + \frac{f(n/b^2)}{(n/b^2)^d} + \frac{f(n/b)}{(n/b)^d} + \frac{f(n)}{n^d}$

↓ keep going

$= \frac{T(n/b^4)}{(n/b^4)^d} + \frac{f(n/b^3)}{(n/b^3)^d} + \frac{f(n/b^2)}{(n/b^2)^d} + \frac{f(n/b)}{(n/b)^d} + \frac{f(n)}{n^d}$

↓

$\frac{T(n)}{n^d} = \text{constant} + \frac{f(n/b^k)}{(n/b^k)^d} + \frac{f(n/b^{k-1})}{(n/b^{k-1})^d} + \ldots + \frac{f(n)}{n^d} \leq C\left(\text{const} + \ldots + \left(\frac{n}{b^2}\right)^{-\varepsilon} + \left(\frac{n}{b}\right)^{-\varepsilon} + \frac{n}{b}\right)$

eventually will equal constant

- In case 1, $f(n) = O(n^{d-\varepsilon})$ or $\frac{f(n)}{n^d} = O(n^{-\varepsilon})$ —
- by the time you hit small powers, gets thrown into constant

$\leq C\left(b^{k\varepsilon} + \ldots + b^{2\varepsilon} + b^{\varepsilon} + 1\right) n^{-\varepsilon}$

$\leq C\left(\text{const} + \frac{(b^{\varepsilon})^{k+1} - 1}{b^{\varepsilon} - 1}\right) n^{-\varepsilon}$

$\frac{T(n)}{n^d} \leq O(1)$ (bounded by a constant)

↳ $O(1) \leq \frac{T(n)}{n^d} \leq O(1)$ ①

so $T(n) = \Theta(n^d)$

Note: $S = 1 + r + r^2 + \ldots + r^k$

$rS = r + r^2 + r^3 + \ldots + r^{k+1}$

$rS - S = r^{k+1} - 1$

$S = \frac{r^{k+1} - 1}{r - 1}$ or $\frac{1 - r^{k+1}}{1 - r}$

$k = \lfloor \log_b n \rfloor$ ↔ k is # of times ratio added up, so $\frac{T(n)}{n^d} = \Theta(\log n)$ ③

→ conclude $T(n) = \Theta(n^d \log n)$

- Case 3:
- Case 2: $f(n) = \Omega(n^{d+\varepsilon})$

↳ $T(n) = f(n) + b^d f(n/b)$ → convert $b^d = a$

$T(n) = \Theta\left(f(n) + af(n/b) + a^2 f(n/b^2) + a^3 f(n/b^3) + \ldots + a^k f(n/b^k)\right)$

- need to weed out larger ones (would outstrip value of f)
  - ↳ Extra hypothesis: regularity condition:

★ $af(n/b) \leq rf(n)$ for some $0 \leq r < 1$ ★

(In divide and conquer, make sure # subsets ≥ process w/o split)

$T(n) = \Theta(f(n))$

- Ex: Solve the recurrences: ① $T(n) = 4T(n/8) + 5n$ ② $T(n) = 8T(n/4) + 10n$ ③ $T(n) = 8T(n/2) + 7n^3$

① $f(n) = 5n$ vs $n^{2/3}$

$a = 4,\ b = 8 \rightarrow d = \log_8 4 \rightarrow 8^d = 4 \rightarrow (2^{3d} = 2^2) \rightarrow 3d = 2 \rightarrow d = 2/3$

regularity con: $4(5 \cdot n/8) \leq r \cdot 5n \rightarrow \frac{1}{2} \leq r$

case 2 $\Rightarrow T(n) = \Theta(f(n)) = \Theta(n)$

② $f(n) = 10n$ vs $n^{3/2}$

$a = 8,\ b = 4,\ d = 3/2$ → case 1 $\Rightarrow T(n) = \Theta(n^{3/2})$

© $\log_2 8 = 3 \rightarrow$ case 3 $\Rightarrow T(n) = \Theta(n^3 \log n)$

[End ex]

- Merge sort: $T(n) = 2T(n/2) + n \rightarrow T(n) = \Theta(n \log n)$

$\log_2 2 = 1 \rightarrow d = 1 \rightarrow n^d = n^1$

+Test $\rightarrow T(1) = \emptyset$

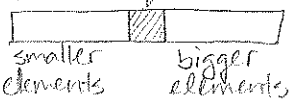
| | $n \log_2 n$ |
|---|---|
| $T(2) = 2T(1) + 2 = 2$ | $2 \cdot 1 = 1 \cdot 2^1 = 2$ |
| $T(2^2) = 2T(2) + 2^2 = 8$ | $4 \cdot 2 = 2 \cdot 2^2 = 8$ |
| $T(2^3) = 2T(2^2) + 2^3 = 24$ | $8 \cdot 3 = 3 \cdot 2^3 = 24$ |
| $T(2^4) = 2T(2^3) + 2^4 = 64$ | $16 \cdot 4 = 4 \cdot 2^4 = 48$ |

$T(2^k) = k \cdot 2^k$, so $T(n) = n \log_2 n$ if $n = 2^k$

★ Goal → find sorting algorithm of $O(n)$, where $n$ is necessary to go through whole array

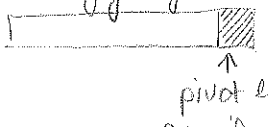
★ Quick Sort ★

- "perfect quicksort" finds middle element, with smaller elements on left and larger elements on the right, then sort left and right




+ in order to find middle element, you have to pretty much sort the whole list

- strategy — guess middle element and hope for the best. Place pivot element wherever it goes, then sort left & right side




+ benefit: finding smaller/bigger elements is the same process of figuring out where pivot belongs

---

1-16

- pick pivot point (first), then work in from left (grab element ≥ pivot) & right (grab element < pivot), then swap. Repeat until indicies meet.
  + make sure you don't go out of bounds (left side, use sentinal or check out of bounds; right side use pivot as sentinal)
- worst case = $\Theta(n^2)$ → thankfully, doesn't happen that often

★ Average Case Behavior:

+ measures what happens randomly, without weight
+ X = random variable which tells # of steps for a given input

+assume all orders are equally likely for inputs

+Ex: $X$ = the role of a die

space of outcomes is $\{1,2,3,4,5,6\} = i$

★ $\text{Prob}(X=i) = p_i$ ★

★ Assumptions on $\{p_i\}$: ★

① $0 \leq p_i$

② $\sum p_i = 1$

For a fair die, $p_1 = p_2 = p_3 = p_4 = p_5 = p_6$

$E(X)$ = expected value of $X$

★ $E(X) = \sum_{i=1} p_i x_i \quad (p_i = \text{Prob}(x = x_i))$ ★

(for fair die) $= \frac{1}{6}\cdot 1 + \frac{1}{6}\cdot 2 + \frac{1}{6}\cdot 3 + \frac{1}{6}\cdot 4 + \frac{1}{6}\cdot 5 + \frac{1}{6}\cdot 6 = \frac{21}{6} = 3.5$

→Average case behavior for insertion sort

+# of orders for $n$ entries $= n!$

+probability $= \frac{1}{n!}$ (if all are equally likely)

+ $E(X) = \sum \frac{1}{n!} \cdot$ (# swaps), where $X$ = # swaps = # transpositions

F
FR
FRI
*FIR
FIRD
*FIDR
*FDIR
*DFIR
*DFIRA
*DFIAR
*DFAIR
*DAFIR
*ADFIR
ADFIR4 (done)

→ 8 swaps

- Easier to count transpositions than swaps

+ if you add the inverse, they transpose in one or the other

$E(X) = \sum \frac{1}{n!}$ (# transpositions) $+ \frac{1}{n!} \sum$ reversed orders

$E(X) = \frac{1}{n!} \frac{\sum(\text{\# of pairs})}{2}$

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

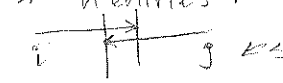
★ $E(X) = \frac{n(n-1)}{4}$ ★

---

1-21 ★ Quicksort

- worst case: $\Theta(n^2)$
- average case: ??

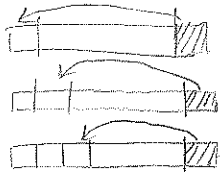
+ We want an estimate for average case # of comparisons


← stopping point

- all elements left of original pivot get compared
- after i and j overlap, those elements are compared twice
  - → (n−1) + 2 = n+1 comparisons to place pivot

* T(N) = expected number of comparisons of running quicksort on an array of n entries
  - → T(N) = N+1 + expected # comparisons on recursive calls
    - possibilities:



→ ∅ + T(N−1)
→ T(1) + T(N−2)
→ T(2) + T(N−3)
⋮
→ T(N−1) + ∅
→ T(N−2) + T(1)
→ T(N−3) + T(2)

- on average, should be about the same on both sides

→ $T(N) = N+1 + \frac{2}{N}\left(T(N-1)+T(N-2)+\dots+T(2)+T(1)\right)$

$NT(N) = N(N+1) + 2\left(T(N-1)+T(N-2)+\dots+T(2)+T(1)\right)$ ↘ replace N w/ N−1

$-\left[(N-1)T(N-1) = (N-1)N + 2\left(T(N-2)+T(N-3)+\dots+T(1)\right)\right]$ ↘ subtract

$\Rightarrow NT(N) - (N-1)T(N-1) = 2N + 2T(N-1)$

$NT(N) - (N+1)T(N-1) = 2N \quad N \geq 3$

→ 2T(2) − 3T(1) = 4, but original eqn has T(2) = 3

- solving ideas:

→ $NT(N) - (N+1)T(N-1) = \emptyset$

$T(N) = \frac{N+1}{N}T(N-1) = \frac{N+1}{N}\frac{N}{N-1}T(N-2) = \frac{N+1}{N}\frac{N}{N-1}\frac{N-1}{N-2}T(N-3)$

$T(N-1) = \frac{N}{N-1}T(N-2)$

$T(N-2) = \frac{N-1}{N-2}T(N-3)$

plug ② T(N−1) into ① T(N−1)

(keep stepping down to) $T(N) = \frac{N+1}{N}\frac{N}{N-1}\frac{N-1}{N-2}\cdots\frac{4}{3}T(2) \rightarrow N+1$

→ compare: $\frac{T(N)}{N+1} = \frac{NT(N)}{N(N+1)}$

$= \frac{(N-1)T(N-1)}{N(N+1)}$ (plugging in *)

$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2}{N+1}$

↓ stepping down again

$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2}{N+1} = \frac{2}{N+1} + \frac{2}{N} + \frac{T(N-2)}{N-1}$

$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2}{N}$

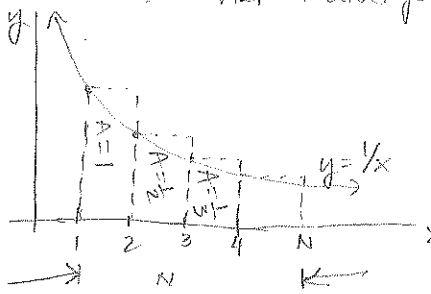
$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2}{N-1}$

$\frac{2}{N+1} + \frac{2}{N} + \frac{2}{N-1} + \frac{T(N-3)}{N-2}$

⋮

$\frac{T(N)}{N+1} = \frac{2}{N+1} + \frac{2}{N} + \frac{2}{N-1} + \dots + \frac{2}{4} + \frac{T(2)}{3}$ ← harmonic series

→ How fast does $\sum_{n=1}^{\infty} \frac{1}{n}$ diverge? (not very fast)




$1 + \frac{1}{2} + \frac{1}{3} + \dots \geq \int_1^{\infty} (1/x)\,dx$

$\ln x \Big|_1^{\infty} = \infty$

$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{N} \geq \int_1^{N} \frac{1}{x}\,dx + \frac{1}{N}$

$\geq \ln N + \frac{1}{N}$

Shift all boxes 1 to the left, so now boxes are under the graph

└ $\ln N + 1$

$1 + \ln N \leq \int_1^N 1/x\,dx \leq \ln N + 1/N$

→ So $1 + \frac{1}{2} + \dots + \frac{1}{N} = \Theta(\ln N)$

- From top of page, $\frac{T(N)}{N+1} = 2 \ln N + O(1)$

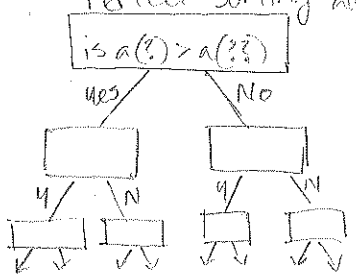
$T(N) = 2(N+1)\ln N$

★ $T(N) = \Theta(N \log N)$ ★ for Quicksort

*Can we beat $\Theta(N \log N)$ behavior?

Step 1: comparison, Step 2: ???, Step 3: Profit.

Perfect Sorting algorithm




□□□□□□□□ → N entries ($N!$ possible orders)

let $q$ = # questions → $2^q$ is most # of cases possibly covered

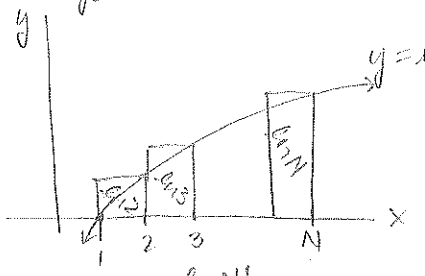
If it's going to work, $2^q \geq N!$ or $q = \log_2 N!$

$= \Theta(N \log N)$

-no matter the algorithm for sorting, minimum is $N \log N$ comps

*Prop: any sorting algorithm requires $\Omega(N \log N)$ comparisons ★

$\log_2 N!$ vs $2N \ln N + O(N)$




$= \int_1^N \ln x \, dx$ + error (triangles)

$= x \ln x - x \Big|_1^N + E$

$= N \log N - N + O(\log N)$

$\log_2 N! = \frac{\ln N!}{\ln 2}$

$= \frac{N \log N - N + O(\log N)}{\ln 2}$

★ average based behavior for quicksort is roughly $2 \ln 2$ * the best theoretically possible # of comparisons for any sorting algorithm.

~39 % worse than the best possible sorting algorithm

(really, really good in average case; really really bad on worst case)

---

1-23 *Merge sort

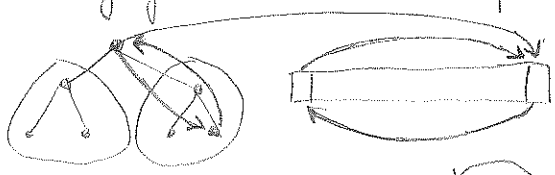
- $\Theta(n \log n)$ in worst case, but requires extra space
- Is it possible to get $\Theta(n \log n)$ worse case behavior w/o extra space?
  + Heapsort

*Heapsort

- a heap is a data structure with the properties

1)  binary tree whose nodes contain some data that can be compared
2) the binary tree is complete down to the last generation, which may have missing nodes
3) at the bottom level, all 2-child nodes are to the left of all no-child nodes and either no 1-child node or one 1-child node between the 2- and no-child nodes (1-child node will always be a left child)
4) the datum for any parent is greater than or equal than that of its child or its children

- assuming you can build a heap, how can you use it to sort the data?

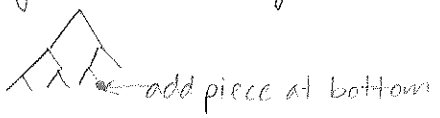


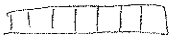
swap largest element to last slot, then fix first element, next, next...

- rough outline: 1) build heap
  2) decapitate heap
  3) fix heap

- how to build heap:


- fix heap: children compete → winner competes with parent → if parent loses, swap → repeat at lower level
- Building the Heap
  + top down → start at beginning of array. At the $i^{th}$ stage, a[1]...a[i] is a heap

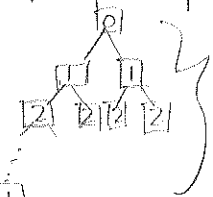



  + bottom up → work from back to front, that leaves you with several heaps  find all children, then parents (which you have to compare and possibly swap) to start merging original heaps
  + # of comparisons: top down

| # nodes | top | bottom |
|---|---|---|
| 1 | ∅ | |
| 2 | 1 | |
| 3 | 1 | |
| ⋮ | | |
| n | | |

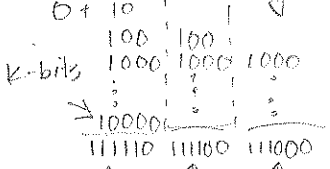
top down: $k^{th}$ generation → k steps to top

$2^k \le n < 2^{k+1}$ ($n^{th}$ node)



$0 \cdot 1 + 1 \cdot 2 + 2 \cdot 4 + 3 \cdot 8 + \dots + (k-1)2^{k-1}$

easiest to do in binary (see below)




$n - 2^k + 1$ leaves in last generation (1 to $2^k$)

$2^{k-2} + 2^{k-4} + 2^{k-8} + \dots + 2^k - 2^{k-1} \to (k-2)2^k + 2$

  + overall, total is $k(n - 2^k + 1) + (k-2)2^k + 2$

$$\bigstar = kn - 2^{k+1} + k + 2 = \Theta(n \log n) \bigstar$$

+ # comparisons: bottom up
- # of children for that node is # of comparisons if bottom node
- big numbers on top, smaller items on bottom → leaves less work on hardes cases
- $\Theta(n)$ ☆

-Heap sort = $\Theta(n \log n)$

---

1-26

- building the heap:
  + top down = $\Theta(n \log n)$ steps
  + bottom up = $\Theta(n)$ steps
- entire algorithm = $\Theta(n \log n)$
  + worst case scenario, fall down $\log_2(n-1) + \log_2(n-2) + \dots + \log_2(1)$
    $= \log_2((n-1)!)$
    $= n \log n$
- as efficient as possible with regards to big O + doesn't need any extra space
- advantage: heap is used to build a Priority Queue

☆ Searching Arrays

- basic problem: given an array a[] and a key, output ① i (index value) if a[i] = key; ② "not found" if no such i exists

* Sequential Search

- go through an array from start to end
- worst case: not there → n comparisons (not there will always be the longest)
  $\Theta(n)$
- average case: assume any location in the array is equally likely; assume the probability that the key is found at all is some known quantity P

| | |
|---|---|
| 1 comparison with probability | $\frac{P}{n}$ |
| 2 | $\frac{P}{n}$ |
| 3 | $\frac{P}{n}$ |
| ⋮ | ⋮ |
| n | $\frac{P}{n}$ |
| not found (n) | $1-P$ |

Expected # of comp = $E(x)$

$E(x) = 1\frac{P}{n} + 2\frac{P}{n} + 3\frac{P}{n} + \dots + n\frac{P}{n} + n(1-P)$

$= \frac{P}{n}(1+2+3+\dots+n) + n(1-P)$

$= \frac{P}{n}\left(\frac{n(n+1)}{2}\right) + n(1-P)$

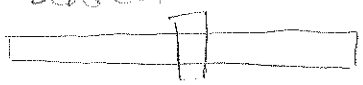
$= \frac{P(n+1)}{2} + n(1-P) = \Theta(n)$

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

Midterm: Wed, Feb 11

→ If the array is sorted, we can do better

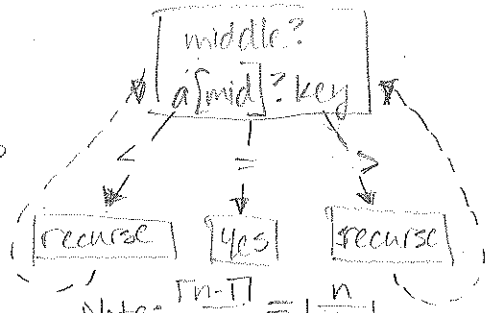
→ Binary Search



- worst case: not found

$T(n)$ = worst case # comps

mid + $\lfloor \frac{n-1}{2} \rfloor$ + $\lceil \frac{n-1}{2} \rceil$




$T(n) = 1 + T(\lceil \frac{n-1}{2} \rceil)$

$= 1 + T(\lfloor \frac{n}{2} \rfloor)$

Note: $\lceil \frac{n-1}{2} \rceil = \lfloor \frac{n}{2} \rfloor$

n in binary: $n = b_k \ldots b_2 b_1$

★ $T(n)$ = # of bits in binary rep of $n$ ★ $\frac{n}{2} = b_k \ldots b_2$

$2^k \le n < 2^{k+1}$

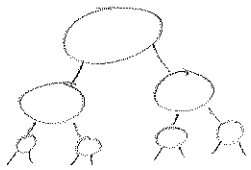
$1\, 0\, \overbrace{\ldots}^{k}\, 0 \le n < 1\, \overbrace{\ldots}^{(k+1)}\, 1$'s

$k \le \log_2 n < k+1$

★ $T(n) = k+1 = \lfloor \log_2 n \rfloor + 1 = \Theta(\log n)$ ★

→ Is it possible to do better?

- currently, no! (maybe in the future)
- imagine "perfect" searching algorithm
  + still has worst case of not-found



needs to cover (n+1) cases (not-found case)

If d is # comparisons, $2^d \ge n+1$

$d = \lceil \log_2 (n+1) \rceil$

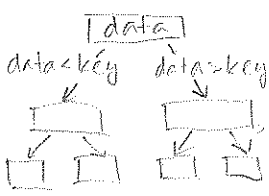
$= \lfloor \log_2 n \rfloor + 1$

- binary search is the most efficient (Homework #2 due Feb 6)

* In many cases, don't have sorted array (b/c array is changing a lot)

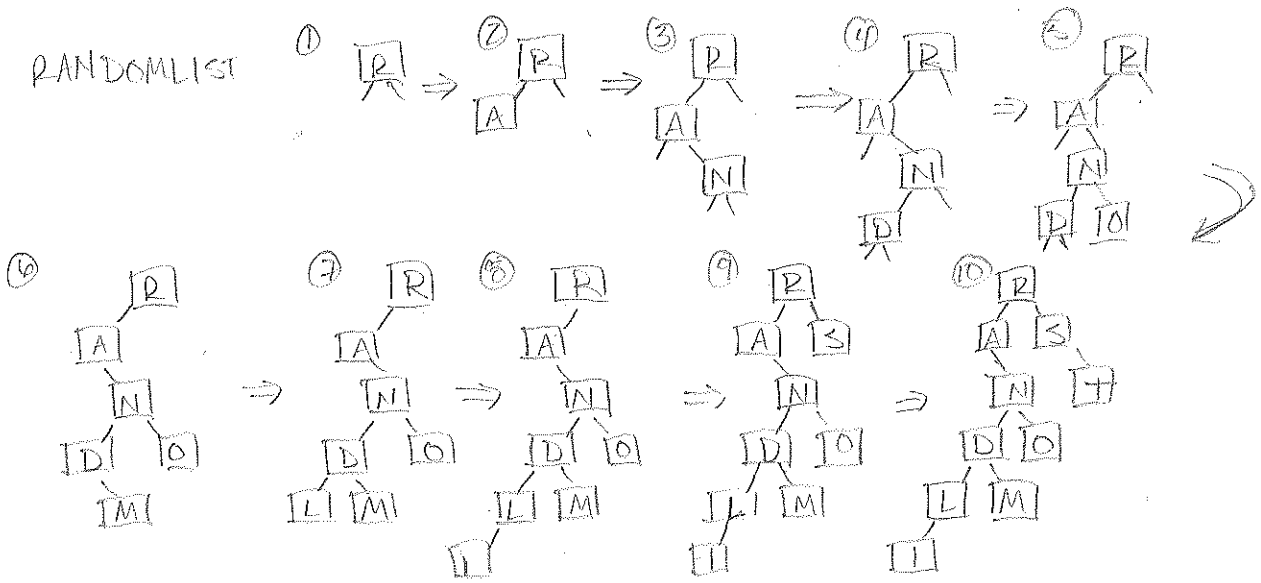
---

1-28

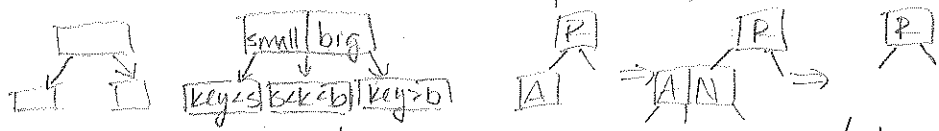



- Need to make process to add things in and keep it ordered
- Rather than having a pre-sorted array, build a dynamic tree based on the order that you're given

RANDOMLIST

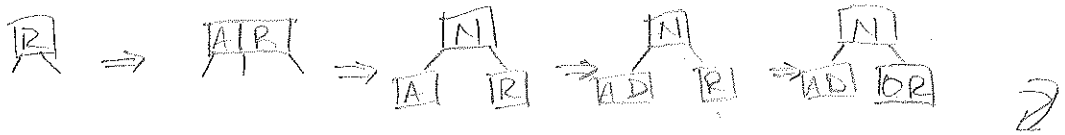


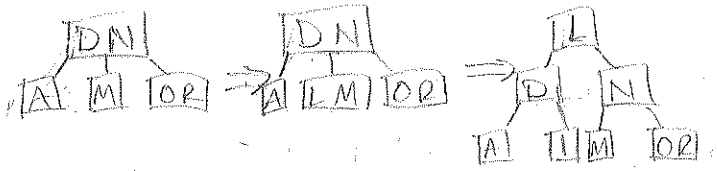

- Problem: worst case is a vine, $\Theta(n)$
  - + need a strategy to make it look more like a tree
  - + rather than one data w/ 2 comparisons, use more data w/ 3 comps



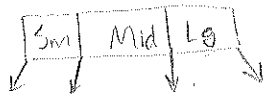

2-node 3-node

both combined is a 2-3 tree ★ (option: have top as 2-node or 3-node)



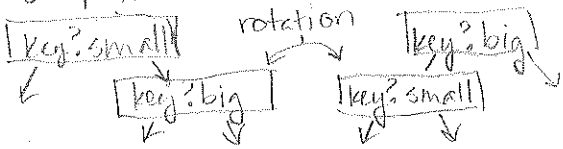


- + if keep balanced, $\Theta(n \log n)$
- + since 2-3 is good, how about 2-3-4? ★
  - - not as often to need promotions → still going to look like $\log n$

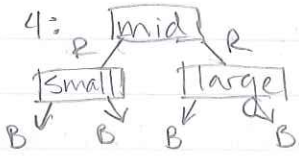


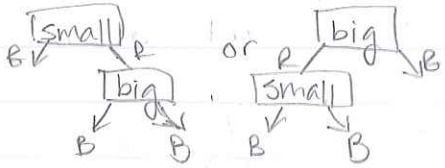

  - - both 3- & 4-nodes only need 2 comparisons:


  - - much better to use 2-3-4 trees
- + 2-3 trees: 2 comparisons:



*red-black tree: red-internal, black-external

4: 


3: 


- can never have a red followed by a red

---

1-30* 2-3-4 Trees

- uses red-black trees

+ rules: - standard binary tree (no multiple nodes)

- if color is stored on node itself, the color refers to the link that it's stored on


+ head of tree is always **black**, also null points are **black**

1) **black** depth of the tree is dependent of path

+ no matter which path you take to a final leaf, there will always be the same # of **black** nodes

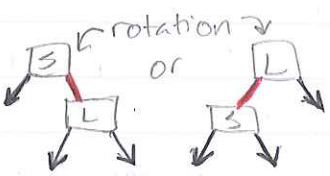
2) no two consecutive **red** nodes in any path

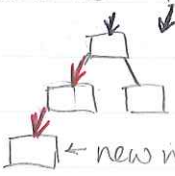
+ shortest path: all **blacks**

+ longest path: twice as much as all **blacks** (alternative RBRB...)

- any path from root to leaf is no more than double any other path
- locating any new node is still $\Theta(\log n)$ steps

+ to add a node, start at top + sort until find where new node belongs, then color it **red** + adjust colors above

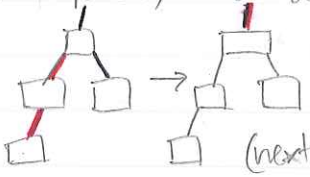

3-node ⇒ 


Problem: 


If these are 2 consecutive reds, fix it.

2 cases:

1) red parent, black uncle



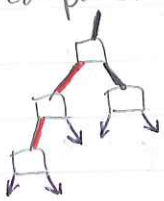
(next page)

2) red parent, red uncle

worst case: $\Theta(\log n)$ fixes

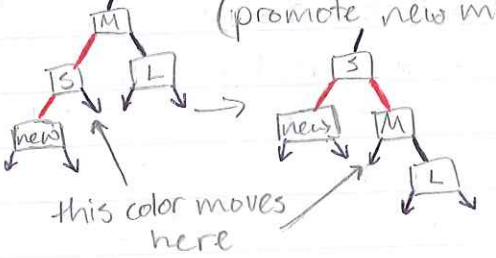
change red head to black

1) red parent, black uncle



cannot color swap b/c change in black depth

need to rotate




(promote new middle case to grandparent)

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

-deleting a **black** node between 2 **reds** has similar fix as above

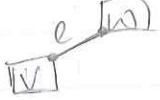
---

2-2 * Graph Algorithms

* Def: a graph is a collection G composed of two sets $G(V,E)$, where V = set of vertices = $\{v_1, v_2, v_3, \ldots v_n\}$ + E = set of edges = $\{(v_{i_1}, v_{j_1}), (v_{i_2}, v_{j_2}), \ldots, (v_{i_n}, v_{j_n})\}$ where the edges are pairs of vertices.
  - the pairs can be ordered, so $(v,w)$ is different from $(w,v)$
    + can also be unordered, so $\{v,w\} = \{w,v\}$
  - when using ordered pairs, it makes a directed graph, or digraph
    + if unordered, undirected graph


Directed

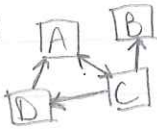
$e = (v,w)$




Undirected

$e = (v,w) = (w,v)$

- Ex:




$V = \{A,B,C,D\}$

$E = \{(A,C), (C,D), (D,A), (C,B), (C,A)\}$

[End ex]

* Possible Representations

1) Adjacency matrix → 2D array indexed by vertices

Directed

| | A | B | C | D |
|---|---|---|---|---|
| A | ∅ | ∅ | 1 | ∅ |
| B | ∅ | ∅ | ∅ | ∅ |
| C | 1 | 1 | ∅ | 1 |
| D | 1 | ∅ | ∅ | ∅ |

$m[i,j] = \begin{cases} 1 \text{ when } [i \to j] \text{ (when there's an edge} \\ \emptyset \text{ when not} \end{cases}$

| | A | B | C | D |
|---|---|---|---|---|
| A | ∅ | ∅ | 1 | 1 |
| B | ∅ | ∅ | 1 | ∅ |
| C | 1 | 1 | ∅ | 1 |
| D | 1 | ∅ | 1 | ∅ |

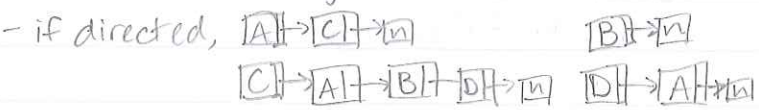
⇒

| | i | j |
|---|---|---|
| i | ∅ | 1 |
| j | 1 | ∅ |

will be symmetric (about the diagonal)

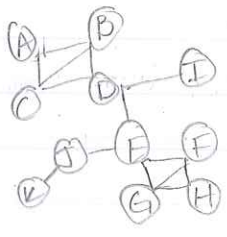
If I have color on here, the scanner will scan in color + make it easier to read :)

- an edge connecting a vertex to itself is called a loop 
- often the case where most entries are Ø - lots of wasted space
  + sparse graph

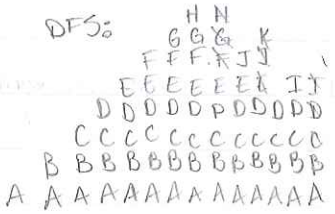
2) Adjacency list: for each vertex,  , form a list of all vertices that are adjacent to to v
   - if directed,  (Λ = null pointer)
   - only have nodes if they're listed, no wasted space
   - big disadvantage, lots of jumping around to search

1) Ø-1 Adjacency Matrix ⇒ Unweighted Graph
   Other #s in Adjacency Matrix ⇒ Weighted Graph
   + Ø might become ∞ or MAX_INT

* Algorithms for Graphs
  - Search the graph (visit the whole thing)
    + Depth-first search → find an unvisited vertex adjacent the current location, repeat → RECURSION! → uses a stack
      - If get stuck, backtrack one step, look for new things. If new thing exists, continue forward. If not, backtrack.
    + Breadth-first seach → visit a new vertex and visit all neighbors before you move on, then visit first neighbor (repeat), second (repeat),..., last neighbor (repeat), continue → uses a queue
    + Recursive DFS(v){
      visit(v)
      mark it as visited
      find unvisited neighbor w
      DFS(w)}
  - can be implemented with a stack
    + pop stack when there are no unvisited neighbors (backtrack)
    + Ex:

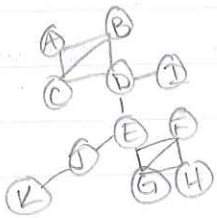



DFS:



Order visited: A B C D E F G H J K I

4.1 #7,8
5.1 #6
5.2 # 1
(3rd or online edition)




BFS: ~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~E~~ ~~I~~ ~~F~~ ~~G~~ ~~J~~ ~~H~~ ~~K~~

Order: A B C D E I F G J H K

In Adjacency Matrix (undirected)

DFS: BFS

| | A | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A | | 1 | 1 | | | | | | | | |
| B | | | | | | | | | | | |
| C | | | | | | | | | | | |
| D | | | | | | | | | | | |
| E | | | | | | | | | | | |
| F | | | | | | | | | | | |
| G | | | | | | | | | | | |
| H | | | | | | | | | | | |
| I | | | | | | | | | | | |
| J | | | | | | | | | | | |
| K | | | | | | | | | | | |

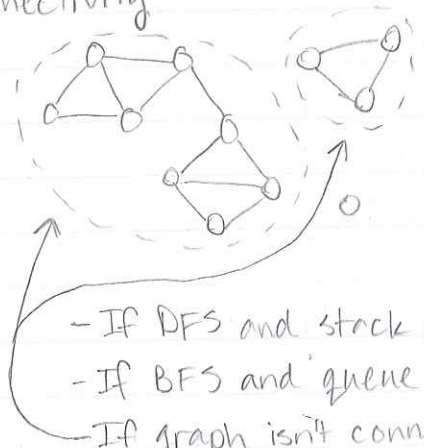
→ # things checked = $\Theta(|V|^2)$

→ for adjacency list, $\Theta(|V|+|E|)$

---

24

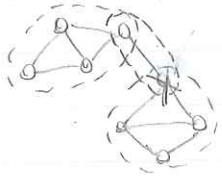
* Connectivity



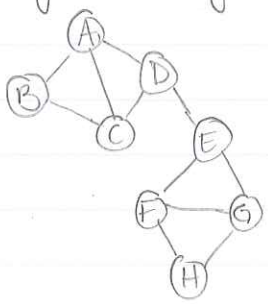
* Assume working with undirected graphs
- Path: a sequence of vertices $v_1, v_2, \ldots, v_k$ where any 2 consecutive vertices $v_i, v_{i+1}$ are adjacent
+ If any two vertices are connected by a path, say the graph is connected

- If DFS and stack empties midway through, graph isn't connected
- If BFS and queue empties before end, graph isn't connected
- If graph isn't connected, any maximal connected subgraph is called a connected component
  + no max number of connected components
  + to count # of connected components, count # of times stack/queue empties

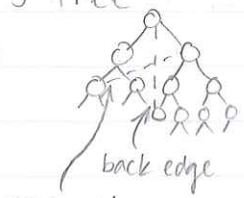
* Biconnectivity
- a connected graph is called biconnected if has ① only one biconnected component (maximal biconnected subgraph) / ② it has no articulation nodes (def next page)



- if a vertex disconnects the graph when it's removed from the graph (vertex and edges) we call the vertex an <u>articulation node</u>
- easy to modify DFS to find articulation nodes

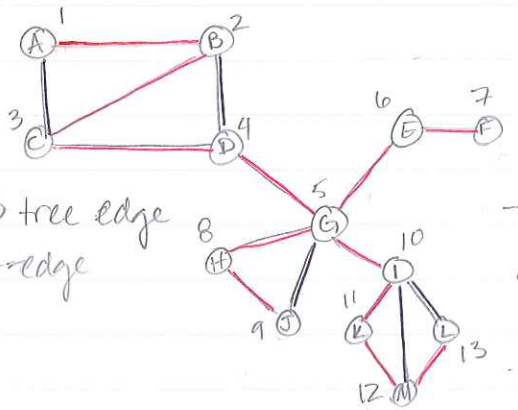



- DFS tree




+ any edge in a tree connecting a vertex to an ancestor is called a <u>back edge</u>

+ any other edge is called a cross edge




— DFS tree edge
— Back-edge

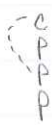
- With DFS tree, all non-tree edges are back edges (w/ no cross edges)

* val(v) = # in the sequence of visited vertices using DFS
- consider all paths from a vertex of the form $v_1 \hat{v}_2 \hat{v}_3 \hat{v}_4 \ldots \hat{v} \ldots \hat{w}$ where ^ are tree edges and ⌒ are back edges
  + define the back value of v to be the minimum of value of w for any path $v \hat{v} \hat{v} \ldots \hat{w}$ or val(v) itself
    - ex: find back-value(G) = 5
    - ex: find back-value(D) = 2
    - ex: find back-value(C) = 1
    - ex: find back-value(F) = 7

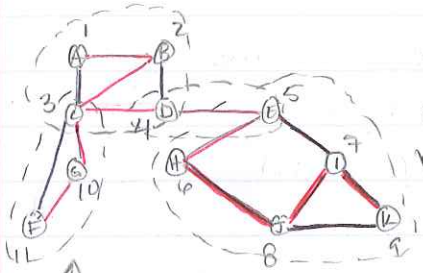
Note: cannot go back on tree-edge (no child to parent)

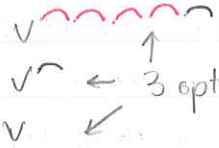
- back-values can be computed "in real time" during DFS


back-value($V_5$) ≤ back-value($V_2$)
- when you push, initialize back-value = value
- as you pop, if back-value(c) < back-value(p), update back-value(p)



2-6

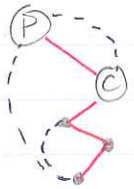



back-value (H) = 3
back-value (G) = 3
back-value (E) = 5

- To make back-value smaller,
  + when you find a back-edge that goes to an ancestor, update parent's back-value if child's < parent's
  + when you have  it makes 
  + if knock out middle edges, it does nothing

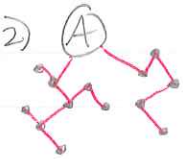
- to find articulation node with parent to child setup, $\forall P \neq A$, if back-value(C) $\geq$ val(P), for some child C of P, then P is an articulation node, separating root A from C




+ if at root:

1) 


If only one child, not an articulation node

2) 


If root has multiple children, it's an articulation node.

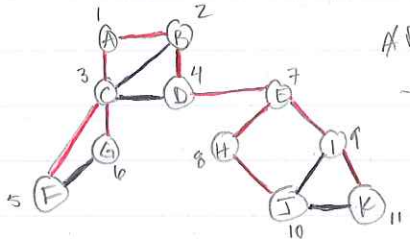
+ If no such child that fits criteria, P is <u>not</u> an articulation node because you can always connect to an ancestor


- find articulation nodes: C, D, E
  + where biconnected components intersect

* Exercise: modify DFS to print out connected components

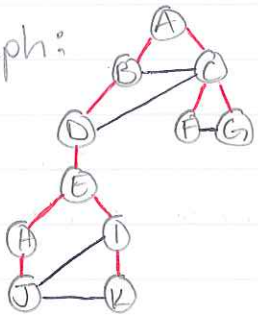

one component

* If you take k-1 shots and still connected, say you are k-connected
* Bread-first search




A B C D E F G H I J K — cross-edge
- now all cross-edges and no back-edges

- Layered graph:




- cross-edges go between cousins, siblings, or 1 layer difference
  + only cross-edges with generation difference ≤ 1

- What's the shortest path in the graph from A to any other vertex?
  + can never go less than the path in the tree.

*Midterm: big O, master theorem, searching/sorting, maybe red-black question, no graphs

---

2-9 *Midterm Review

- Big O
  + ex: $\log n! = \Theta(n \log n)$
  + Master Theorem $\Rightarrow T(n) = aT\left(\frac{n}{b}\right) + f(n)$

4) a) $T(n) = 1$ and $T(n) = 8T\left(\frac{n}{2}\right) + 3n^2$

$f(n) = 3n^2 \quad a = 8 \quad b = 2$

$d = \log_b a = \log_2 8 = 3 \rightarrow \Theta(n^3)$

$\rightarrow f(n) = \Theta(n^2) < \Theta(n^3)$ → so $T(n) = \Theta(n^3)$

b) $T(n) = 1$ and $T(n) = 4T\left(\frac{n}{4}\right) + 4n$

$f(n) = 4n^1 = \Theta(n) \quad a = 4, \; b = 4$

$d = \log_b a = \log_4 4 = 1$

$f(n) = \Theta(n) = \Theta(n) \rightarrow T(n) = \Theta(n \log n)$

- GENERAL:
  1) if $f(n) = O(n^{d-\varepsilon})$ then $T(n) = \Theta(n^d)$
  2) if $f(n) = \Theta(n^d)$ then $T(n) = \Theta(n^d \log n)$
  3) if $f(n) = \Omega(n^{d+\varepsilon})$ (and regularity condition $af\left(\frac{n}{b}\right) < kf(n)$ $(k<1)$, then $T(n) = \Theta(f(n))$, else $T(n) = \Omega(f(n))$

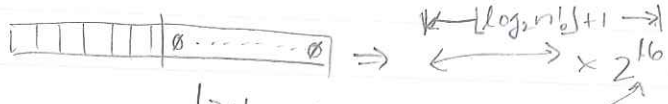
4) c) $T(n) = 1 + T(n) = 2T\left(\frac{n}{8}\right) + 5\sqrt{n}$

$d = \log_8 2 = \frac{1}{3}$

$f(n) = 5n^{1/2} \rightarrow 2f\left(\frac{n}{8}\right) < kf(n) \rightarrow 2\left(5\sqrt{\frac{n}{8}}\right) < k5\sqrt{n} \rightarrow \frac{2}{\sqrt{8}} < k < 1$

so $T(n) = \Theta(\sqrt{n})$

3) 18!




↳ store binary places

$\lfloor \log_2 n! \rfloor + 1$ is the # of bits before you chop off the Øs

the number of trailing Øs is given n-bitsum(n)

ex: n=18

18-bitsum(18)

10010 → bitsum=2

18-2 = 16

| f(n) | n-bitsum(n) |
|---|---|

$f(n) = \lfloor \log_2 n! \rfloor + 1 - (n\text{-bitsum}(n)) \Rightarrow \Theta(n \log n)$ since $\log_2 n! = \Theta(n$

$g(n) = \lfloor \log_2 (n\text{-bitsum}(n)) \rfloor + 1 \Rightarrow \Theta(\log n)$ since $\text{bitsum}(n) \approx \Theta(n)$

- ex: $f(18) = 37$

$g(18) = 5$

$\frac{f(n)}{g(n)} = \Theta(n)$

$cn\log n \le f(n) \le C n \log n$

$k \log n \le g(n) \le K \log n$

$\frac{cn\log n}{K \log n} \le \frac{f(n)}{g(n)} \le \frac{k \log n}{cn \log n}$

so $\frac{f(n)}{g(n)} = \Theta(n)$

2?) Insertion sort (last 2) of CRAZYSECTION

ACEIOQRSTUYZ | N

N+0 → swap

N+1 → no swap

(6) a) $\frac{1}{n} \mid \frac{1}{n} \mid \frac{1}{n} \mid \frac{1}{n} \mid \frac{1}{n} \mid \frac{1}{n} \mid \frac{1}{n} \mid \frac{1}{n}$

w/ n entries

1 2 3 4 5 6 7 . . . n

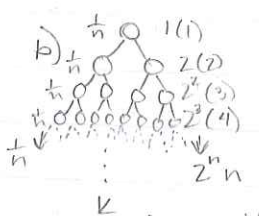
↖ # tries to find with probability $\frac{1}{n}$

$= \frac{1}{n}1 + \frac{1}{n}2 + \frac{1}{n}3 + \ldots \frac{1}{n}n$

$= \frac{1}{n}(1+2+\ldots n)$

$= \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$

b)




Expected # comparisons:

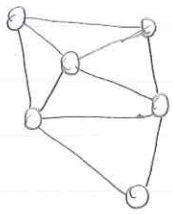
$\frac{1}{n}(1 + 2\cdot 2 + 3\cdot 2^2 + 4\cdot 2^3 + \ldots n\cdot 2^n)$

→ even if top generations are Øs, bulk is $> \frac{k}{2}$

+ avg # looks like k (or k-1)

2-13 * Breadth first search

- finds shorest paths in a graph
- more general problem:
  + in a weighted graph, find the path of the smallest total weight

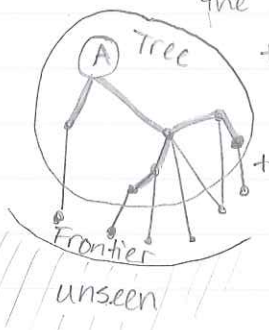


- for each edge there is some weight $w(e)$ which gives time/distance/cost/etc... of travelling via that edge

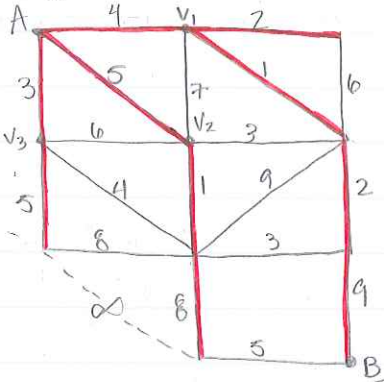
$A \xrightarrow{w(e_1)} V_1 \xrightarrow{w(e_2)} V_2 \xrightarrow{w(e_3)} V_3 \cdots B \Rightarrow w(e_1)+w(e_2)+\ldots w(e_k) =$ total weight of path

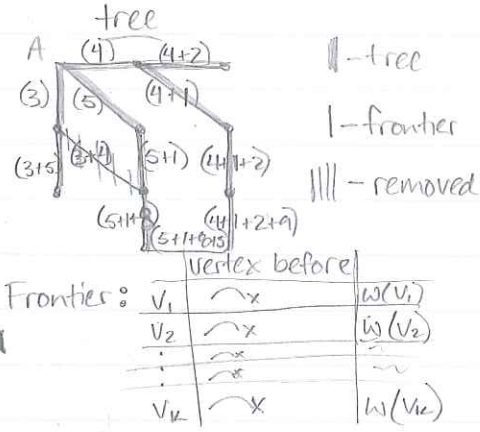
  + need to go through the entire graph in case shortest path is the last one

- Goal: find the shortest path from one vertex to another in no more time (constant multiple) than searching the graph
  ↳ really asking for the shortest path to get from A to anywhere
  + one thing to consider as we go: does it work with negative weights?
    - as long as there are no negative weights, we can immediately find the closest vertex to A




  + shortest known paths structure is a tree, where a tree is a connected graph with no cycles
  + assumption: graph is undirected, so we can travel in two directions

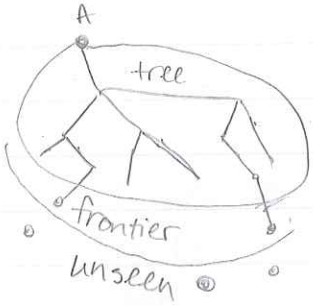



- Dijkstra Shortest Path Algorithm
- greedy strategy - picking smallest path to continue

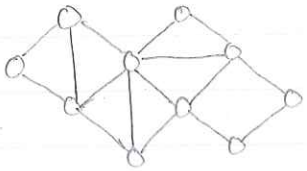
shortest path may not get you to the treasure chest

2-20



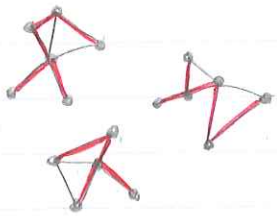

- assuming all weights are positive, this will provide the shortest path to any vertex
- cummulative weight to add new path

* Minimum Spanning Tree



- passenger wants shortest route
- bus company owner needs to cut back options, making paths longer but still keeping all places available
  + want to cut back redundancies (no more straight round trips or cycles)
  + going to make a connected acyclic graph (aka a tree)

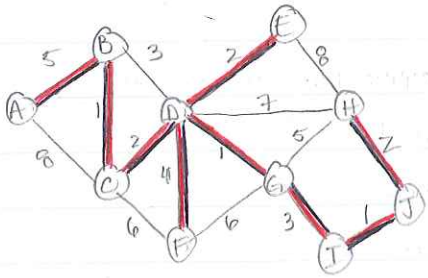
* A spanning tree for a subgraph G is a subgraph of G which is a tree (connected acyclic) on all the vertices of G
  - G has to be connected to have a spanning tree

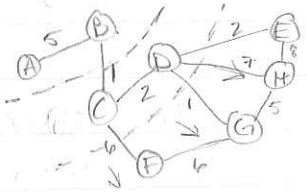


← a spanning forest

* minimum spanning tree problem: find the spanning tree (forest) for G with the minimum total weight
  - update tree → update frontier
  - no cummulative weight - just pick weight

- Ex:




+ Don't necessarily pick edge connected to last vertex added - just find path of smallest weight

- identical weights creates multiple options
- Prim's Algorithm ↗
- Kruskall's Algorithm - pick smallest path + that it

2-23 * Minimum Spanning Tree

- Prim's Algorithm
  - + greedy
  - + count cost of the edge
- Kruskal's Algorithm
  - + super greedy
  - + add in edges as cheaply as possible to the subgraph, never forming a cycle until you're stuck
    - disconnected sections eventually join together
  - + steps:
    - sort all edges by weight
      - + $\Theta(|E|\log(|E|))$, $|E|$ = # of edges in graph

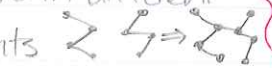
edges {

a) if both vertices are new, start a new component with that edge

b) if edge between an old + new vertices, connect them

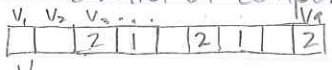


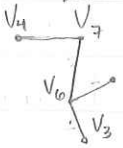
c) if edge connects two old vertices, then either ① the vertices are in different components, where you add the edge to merge the components



② or it will make a cycles, so you ignore it and do nothing

- make a list of components




Add $V_4 \to V_7$. Add $V_6 \to V_9$.
Add $V_6 \to V_3$. Add $V_6 \to V_7$

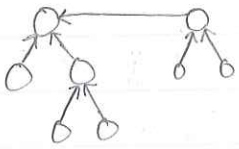
- to update for C1 case ↑, it will need a lot of steps
- need a more efficient strategy for C1

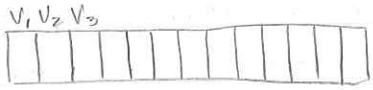
* Union-Find Problems

- objects are disjoint sets (For Kruskal's, elements ↔ vertices + sets ↔ components

$S_1 = \{\quad\}$
$S_2 = \{\quad\}$
$S_3 = \{\quad\}$
⋮

- Find operation: Find(i)
  - + finds the set that contains the i element
- Union operation: Union(i,j)
  - + if find(i) = find(j) → skip it
  - + if find(j) = ∅ but find(i) is a set → join j to find(i)
  - + if find(i) = find(j) = ∅ → create new set with i + j
  - + if find(i) ≠ find(j) (not ∅), then union (i,j)



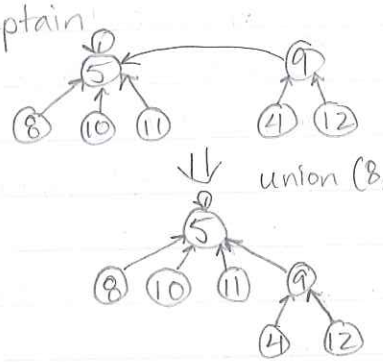



- array strategy for union-find
  - + make reverse pointing tree where head points to itself
  - + Ex: $S_1 = \{5, 8, 10, 11\}$
    $S_2 = \{4, 9, 12\}$
    $S_3 = \{6, 7\}$
    $S_4 = \{13, 15, 16\}$

| 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 5 | | | 5 | 9 ↓ 5 | 5 | 5 | 9 | | | |

union (8, 12)

  - - only update the team captain → other members of team point to captain




  - - longer depth means longer find ()
    - + the one with the bigger depth should be captain (join short to long)
    - + captain holds a depth counter for comparisons
- + path compression
  - - want to compress long depths ("long stringy thing")
  - - reassign members to point directly to captain

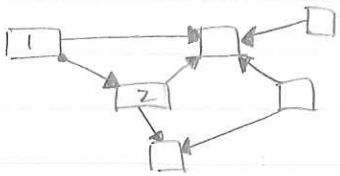
| 5 | 12 | 11 | 15 | 14 |
|---|---|---|---|---|
| 5 | 11 ↓ 5 | 15 ↓ 5 | 5 ↓ 5 | 12 ↓ 5 |

find(14)

  - - will (normally) give you shorter trees
  - ~ $\Theta(\log n)$

2-25 * Topological Sorting

- want to sort events based on necessary order of some events




want a list with all arrows going in the same direction: 

- a **topological sort** of the graph is a listing of vertices from left to right with all directed edges pointing rightward.
- given a directed graph, determine whether it has a topological sort + if so, find it

+ a **directed cycle** is a cycle in a directed graph
+ no graph with a directed cycle can have a topological sort
+ suppose we have a directed acyclic graph (aka a bag)

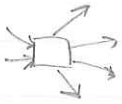
- any directed finite acyclic graph has a topological sort
  + Proof: by induction of the size of the graph

    base: size 1  (cannot have  b/c it's a cycle)

    hypothesis part: Assume we can sort all directed acyclic graphs of less than n vertices.

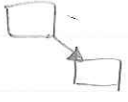
    We want to show we can do it for n vertices.

    → We have to start with a vertex that has nothing pointing to it (indegree zero)



- **outdegree**: # of edges coming out of the vertex
- **indegree**: # of edges pointing into the vertex

1) Find a vertex of indegree zero



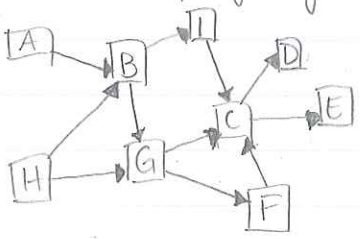
+ pick a vertex: if it is indegree zero, done; if not work back, repeat

2) Put it as # 1 in your list

3) Delete it + its outward arrows.

4) Topologically sort what's left.

- Ex:




A H B G I F C E D (all arrows going →)

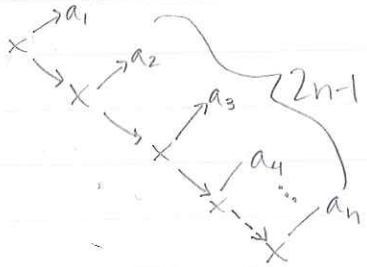
- to implement, reverse direction of arrows then do DFS
  + if you have an adjacency matrix of the graph, transpose it (swap rows and columns
  + if you have a linked-list representation of the graph, it's going to be very messy

## ✱ COMPUTATIONAL ALGORITHMS

- Ex: given a polynomial, compute its value at some point x
- Ex: given two matrices, find their product

✱ Polynomial evaluation problem:

- def: a **polynomial** is an expression of the form $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$, $a_n \neq \emptyset$
- def: the **degree** is the highest power, n
- computer's view:
  + input: x (some value) and a[ ] (a[ø], a[1], ... a[n])
    - polynomials of degree n has n+1 coefficients.
  + output: $a[n] x^n + a[n-1] x^{n-1} + \ldots + a[1] x + a[ø]$
  + how many steps does this take?
    - # of multiplications $= n + (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n+1)}{2} = \Theta(n^2)$
    - # of additions = n
  + Ex:




2n-1 multiplications then n additions → $\Theta(n)$

  + Ex: $3x^4 + 2x^3 - 9x^2 + 5x - 12$

    $3x^4 + 2x^3 \rightarrow x^3(3x+2) - 9x^2 \rightarrow x^2(x(3x+2) - 9) + 5x \rightarrow x(x(x(3x+2) - 9) + 5) - 12$

    ↳ 4 additions, 4 subtractions

- **Horner's Method** ↑
  + n multiplications + n additions
  + you cannot do any better than this

2-27 *$p(z) = ((((a[n]z + a[n-1])z + a[n-2])z + \dots a[1])z + a[0]$

- **Straight line program**
  - + statements are all of the form: $s = x$ op $y$ (where op is an operation taken from an approved list: $+, -, *, \div$)
    - $x$ + $y$ can be constants or input variables or previously calculated values
- Horner's Method as straight line program

$s_1 = a[n] * z \longrightarrow a[n]z$

$s_2 = s_1 + a[n-1] \longrightarrow a[n]z + a[n-1]$

$s_3 = s_2 * z \longrightarrow (a[n]z + a[n-1])z$

$s_4 = s_3 + a[n-2] \longrightarrow (a[n]z + a[n-1])z + a[n-2]$

$\vdots$

$s_{2i-1} = s_{2i-2} * z$

$s_{2i} = s_{2i-1} + a[n-i]$

$\vdots$

$s_{2n-1} = s_{2n-2} * z$

$s_{2n} = s_{2n-1} + a[0]$

+ produces n additions and n multiplications

- Ex: $p(z) = z^{15} + z^{14} + z^{13} + z^{12} + \dots + z^2 + z + 1$

$$(z-1)p(z) = (z-1)(z^{15} + \dots + 1)$$
$$= (z^{16} + z^{15} + z^{14} + \dots + z) - (z^{15} + z^{14} + z^{13} + \dots + 1)$$
$$= z^{16} - 1$$
$$p(z) = \frac{z^{16}-1}{z-1}$$

$s_1 = z * z$

$s_2 = s_1 * s_1$

$s_3 = s_2 * s_2$

$s_4 = s_3 * s_3$

$s_5 = s_4 - 1$

$s_6 = z - 1$

$s_7 = s_5 \div s_6$

↑ done in 7 steps rather than 30

* Theorem: any straight line program that calculates the value of a degree-n polynomial

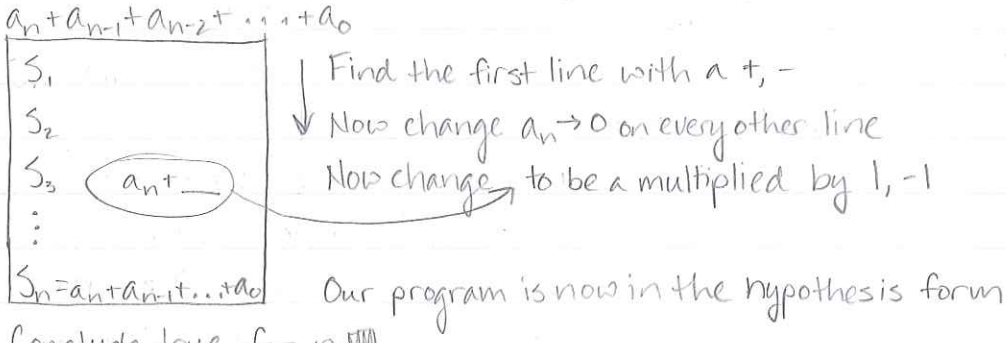
(with user inputted coefficients/variable) requires at least $n$ +, − and at least $n$ *, ÷

- For a general case, Horner's method is most efficient
- Stronger statement for +, −
  - + Any straight line program which calculates $a_n+a_{n-1}+a_{n-2}+\dots+a_0$ requires at least $n$ additions/subtractions
  - + Proof by induction:

    Basis: $n=0$ (trivial)

    Hypothesis: Assume the result for $n-1$. Show that it is true for $n$.

    Inductive step: Assume we have a straight line program that calculates

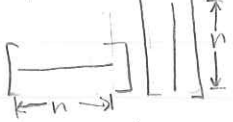



    Conclude true for $n$ ∎

- For now, we pause on polynomial evaluation

* Matrix multiplication
  - $A=[a_{ij}]$, $1\le i\le m$, $1\le j\le n$, with $m$ rows + $n$ colums

    $m \times n$ matrix
  - $B=[b_{ij}]$ $1\le i\le m$, $1\le j\le n$.
  - matrix addition: if $A$ + $B$ are both $m \times n$ matices, then $A+B=[a_{ij}+b_{ij}]$ $1\le i\le m$, $1\le j\le n$
  - matrix multiplication:




$[c_{ij}]=[a_{i_1}b_{j_1}+a_{i_2}b_{j_2}+a_{i_3}b_{j_3}+\dots+a_{in}b_{jn}]$ where $C$ is $m \times p$

+ Ex: $\begin{bmatrix}2&3&6\\4&5&7\end{bmatrix}\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}=\begin{bmatrix}\ &\ &\ \\ \ &\ &\ \end{bmatrix}$

  + Have $mnp$ multiplications and $m(n-1)p$ additions
  + For square matrices, have $n^3$ multiplications and $n^2(n-1)$ additions
    - $\Theta(n^3)$

3-2 ✱Last time:

- Have A, B n×n matrices
- C = A·B is defined using $n^3$ *'s and $n^2(n-1)$ +'s
  + $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{jk}$

✱In particular for 2×2 matrices:

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

✱ **Strassen's Algorithm** – works on both numbers + matrices themselves

- $x_1$ through $x_7$ involve 1 or 2 additions/subtractions and 1 multiplication
- $c_{11}, c_{12}, c_{21}, c_{22}$ are additions of certain elements of $x_1$ through $x_7$
- Ex: $c_{12} = x_3 + x_5 = a_{11}(b_{12} - b_{22}) + (a_{11} + a_{12})b_{22}$

$$= a_{11}b_{12} - a_{11}b_{22} + a_{11}b_{22} + a_{12}b_{22}$$

$$= a_{11}b_{12} + a_{12}b_{22}$$

↳ 7 *'s, 18 +/-'s versus 8 *'s, 4 +'s

- only one * fewer than before, so not much better

- maintains order of multiplication (a's on left, b's on right)
  + note that matrices cannot commute ($A \cdot B \neq B \cdot A$)
- Cut any n×n matrix into quarters to produce 4 smaller matrices


- Strassens on 4×4 Matrices
  + 7 *'s of 2×2 matrices → 7(7 *'s), 7(18 +/-'s)
  + 18 +/-'s of 2×2 matrices → 4(18 +/-'s)

  } 49 *'s and 198 +/-'s total
  + 15 less *'s, 150 more +/-'s
- Strassens on 8×8 Matrices
  + 7 *'s of 4×4 → $7^2$ *'s, 198 +/-'s
  + 16 +/-'s of 4×4 → 18
  + total: $7^3$ *'s, $(7 \cdot 198 + 18 \cdot 16)$
- Note: if you're not a power of 2, zero-fill to next $2^k$
  + Ex: $\begin{bmatrix} \cdot & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$

- If $2^k \times 2^k$ and you multiply with Strassen's, we would get: ① $7(2^{k-1} \times 2^{k-1})$ multiplications
② $7 * 18(2^{k-1} \times 2^{k-1})$ additions/subtractions

+ if only count multiplications, $T(n)$ = # of *'s for $n = 2^k$

$$= 7 \cdot T\left(\frac{n}{2}\right)$$
$$= 7^2 \cdot T\left(\frac{n}{4}\right)$$
$$\vdots$$
$$= 7^k T(1) = 7^k$$

+ so Strassens uses $7^k$ multiplications vs $8^k$ by definition

$n = 2^k$

$k = \log_2 n$

$7^k = 7^{\log_2 n} = n^{\log_2 7}$ $(\log_2 7 \approx 2.81)$

+ For additions (and subtractions), $T(n) = 7 \cdot T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2$

$a = 7$, $b = 2$ $d = \log_2 7 \approx 2.81$

$f(n) = \Theta(n^2) < n^d$

$T(n) = \Theta(n^{\log_2 7})$

* Polynomial Evaluation (cont'd):
- Given $p(z)$, find $p(z_1), p(z_2), \ldots$ in such a way that calculations are re-usable to be more efficient than Horner's
+ possibilities: ① arithmetic sequence → evenly spaced

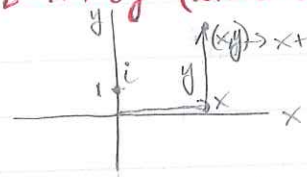
② geometric sequence → powers of r

③ a circle → evenly spaced around a circle

+ both arithmetic + geometric using complex #'s

* Complex numbers:

$z = x + iy$ (where $i = \sqrt{-1}$)




+ i is a martian that is peacefully visiting Earth
- needs to blend in as well as possible, so i will behave under all real # rules

- addition: $z_1 = x_1 + iy_1$, $z_2 = x_2 + iy_2$

$z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2)$ where $(y_1 + y_2)$ is "imaginary"

+ complex addition is vector addition

- multiplication: $z_1 \cdot z_2$

Get real!

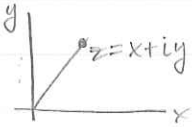
Be rational!

$$z_1 \cdot z_2 = (x_1 + iy_1)(x_2 + iy_2)$$
$$= x_1x_2 + x_2iy_1 + x_1iy_2 + iy_1iy_2 \qquad \text{Note: } i^2 = -1$$
$$= x_1x_2 + i(x_2y_1) + i(x_1y_2) - y_1y_2$$
$$= (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1)$$

---

3-4

★Last time:

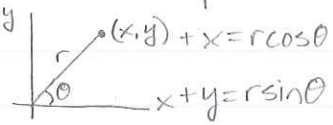



- complex numbers ($i = \sqrt{-1}$)
- addition: $z_1 = x_1 + iy_1$ + $z_2 = x_2 + iy_2$

$$z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2)$$

- multiplication: $z_1 \cdot z_2 = (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1)$

★Polar form for complex numbers




+ $x = r\cos\theta$
+ $y = r\sin\theta$
+ $x + iy = r\cos\theta + ir\sin\theta$
$= r(\cos\theta + i\sin\theta) = |z|(\cos\theta + i\sin\theta)$

- origin: $O = (0,0) = 0 + i0$
+ $r$ = distance from $z$ to $O = |z| = \sqrt{x^2 + y^2}$ (definition)
- Say $f(\theta) = \cos\theta + i\sin\theta$

$f'(\theta) = -\sin\theta + i\cos\theta$
$= i^2\sin\theta + i\cos\theta$
$= i(\cos\theta + i\sin\theta)$
$= if(\theta)$

Note: $f'(\theta) = if(\theta)$

$\frac{f'(\theta)}{f(\theta)} = i$

$\ln(f(\theta)) = i\theta + C$

$e^{f(\theta)} = e^{i\theta + C}$
$= (e^C)e^{i\theta}$

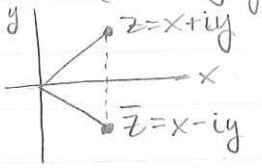
- Definition: $e^{i\theta} = \cos\theta + i\sin\theta$ ☆
  - ↳ $(e^{i\theta})^2 = \cos 2\theta + i\sin 2\theta$
  - ↳ $(\cos\theta + i\sin\theta)^2 = \cos^2\theta - \sin^2\theta + i(2\sin\theta\cos\theta)$, so
  + $\cos 2\theta = \cos^2\theta - \sin^2\theta$ ☆
  + $\sin 2\theta = 2\sin\theta\cos\theta$ ☆

$|\cos\theta + i\sin\theta| = 1 = \sqrt{\cos^2\theta + \sin^2\theta}$

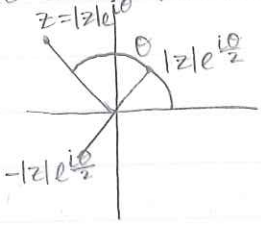
+ $z_1 = |z_1|(\cos\theta_1 + i\sin\theta_1)$ + $z_2 = |z_2|(\cos\theta_2 + i\sin\theta_2)$

$z_1z_2 = |z_1||z_2|(\cos\theta_1\cos\theta_2 - \sin\theta_1\sin\theta_2 + i(\sin\theta_1\cos\theta_2 + \cos\theta_1\sin\theta_2))$
$= |z_1||z_2|(\cos(\theta_1 + \theta_2) + i\sin(\theta_1 + \theta_2))$

- Define $\bar{z}$ (the conjugate of $z$) to be $\bar{z} = x - iy$



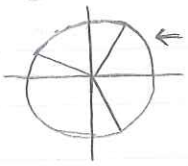

- Note: $-1 = \cos\pi + i\sin\pi$ so $-1$ is a 180° rotation and $\sqrt{-1} = i$ is a 90° rotation
  + also $(-i)^2 = -1$
- we cannot use less-than or greater than when working with complex #'s




$\sqrt{z} = \sqrt{|z|}\, e^{i\frac{\theta}{2}}$

If $\theta = \theta + 2\pi = \theta + 4\pi + \theta + 6\pi$ then

$\frac{\theta}{3} = \frac{\theta + 2\pi}{3} = \frac{\theta + 4\pi}{3} + \frac{\theta + 6\pi}{3}$

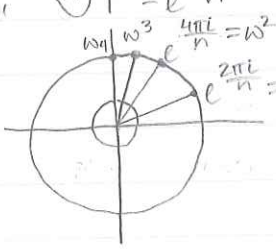


← 3 evenly spaced cube-roots
- what about more pieces?

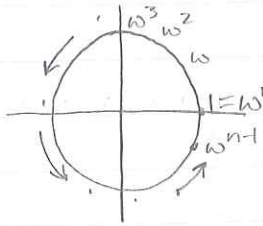
- $n^{th}$ roots of unity:

$1 = e^{i0}$ or $e^{i2\pi}$ or $e^{i4\pi}$ or any $e^{i2\pi k}$

so $\sqrt[n]{1} = e^{\frac{2\pi i k}{n}}$




$\Longrightarrow$




- This is a geometric sequence.
- Because of this we can reuse our calculations if evaluated at $w^n$

* So given a polynomial, we want an efficient algorithm for calculating the polynomial at some particular roots of unity
  - given a polynomial of degree $n = 2^k - 1$, we have $2^k$ coefficients in the polynomial

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + a_{n-2} z^{n-2} + \ldots + a_2 z + a_1 z + a_0$$

(odd, even, odd)

$$= (a_n z^n + a_{n-2} z^{n-2} + a_{n-4} z^{n-4} + \ldots) + (a_{n-1} z^{n-1} + a_{n-3} z^{n-3} + \ldots)$$

- Ex. $p(z) = 2z^7 - 4z^6 + 9z^5 + 10z^4 - 12z^3 + 3z^2 - 5z + 18$

$= (2z^7 + 9z^5 - 12z^3 - 5z) + (-4z^6 + 10z^4 + 3z^2 + 18)$ ← split even + odd

$= 5z(2(z^2)^3 + 9(z^2)^2 - 12(z^2) - 5) + -4(z^2)^3 + 10(z^2)^2 + 3(z^2) + 18$ ← make common terms

* Recursive step: To evaluate $p(z)$ (of degree $n = 2^k - 1$), collect odd powers + even powers

$p(z) = a_n z^n + a_{n-1} z^{n-1} + a_{n-2} z^{n-2} + \dots$

$= \underbrace{a_n z^n + a_{n-2} z^{n-2} + \dots}_{\text{odd} = p_O} + \underbrace{a_{n-1} z^{n-1} + a_{n-3} z^{n-3} + \dots}_{\text{even}}$

$a_{n-1}\ a_{n-3}\ a_{n-5}$

$p_E(z^2)$

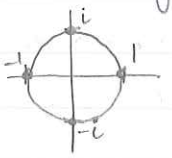
- from ex: $p_O = 2z^3 + 9z^2 - 12z - 5$

$p_E = -4z^3 + 10z^2 + 3z + 18$

$p_E(z^2) = -4z^6 + 10z^4 + 3z^2 + 18$

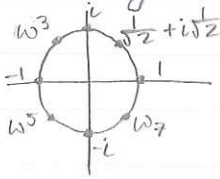
★ $p(z) = z p_O(z^2) + p_E(z^2)$ ★

- $4^{th}$ roots of unity:




- If you square the $4^{th}$ roots of unity, get the $2^{nd}$ roots $(1, -1)$

- $8^{th}$ roots of unity:




- If you square the $8^{th}$ roots of unity, get the $4^{th}$ roots

- Evaluate $p(z)$ at the $(n+1)^{th}$ or $(2^k)^{th}$ roots of unity

3-6 ★ Let $p(z) = a[n] z^n + a[n-1] z^{n-1} + \dots + a[1] z + a[0]$ be a polynomial of degree $n = 2^k - 1$ (so there are $2^k$ coefficients)

- split $p(z)$ into odd/even pairs: $p(z) = z p_O(z^2) + p_E(z^2)$
- coeffs of $p_O \rightarrow a[n], a[n-2], \dots, a[3], a[1]$ } $2^{k-1}$ coeffs, degree $2^{k-1} - 1$

  $p_E \rightarrow a[n-1], a[n-3], \dots, a[2], a[0]$ }
- to evaluate $p(z)$ at $z = 1, \omega, \omega^2, \dots, \omega^{n-1}$ (with $(n+1)^{th}$ roots of unity), evaluate $p_O(z) + p_E(z)$ at $z^2 = 1, \omega^2, \omega^4, \dots, \omega^{n-1}$ (with $(n/2)$ roots of unity)

$\omega = e^{\pi i \frac{2\pi}{n+1}} = e^{\frac{2\pi i}{2^k}}$ (remember $e^{i\theta} = \cos\theta + i\sin\theta$)

$= \cos\left(\frac{2\pi}{2^k}\right) + i\sin\left(\frac{2\pi}{2^k}\right)$ → also can be written as $\text{cis}\left(\frac{2\pi}{2^k}\right)$

$\begin{bmatrix} a[0] \\ a[1] \\ \vdots \\ a[n] \end{bmatrix} \xrightarrow{\text{output}} \begin{bmatrix} p(1) \\ p(\omega) \\ \vdots \\ p(\omega^n) \end{bmatrix}$

Note: $p(z) = z p_O(z^2) + p_E(z^2)$

$p(-z) = -z p_O(z^2) + p_E(z^2)$

because of symmetry of roots of unity

★ How many calculations is this?

- to calculate $p(1), p(w), p(w^2), \dots p(w^n)$; $n = 2^k - 1$
  we do the $2^{k-1}$ case, $p_0(z^2) + p_E(z^2)$ and then $p(z) = z p_0(z^2) \pm p_E(z^2)$
+ $T(k) = T(k-1) + T(k-1) + 3(2^{k-1})$
  ↓ → 3 operations: addition, subtraction, multiplication
* $T(k) = 2T(k-1) + 3 \cdot 2^{k-1}$ *
  ↳ put in terms of n to reach master theorem

$T(n) = 2T\left(\frac{n+1}{2}\right) + 3\left(\frac{n+1}{2}\right)$ ← $n = 2^k - 1$, $n+1 = 2^k$, $\frac{n+1}{2} = 2^{k-1}$

$a = 2, \ b = 2$

$\log_2 2 = 1$

→ $O(n \log n)$

- if we were to evaluate through Horner's method, expect $O(n^2)$
  + Horner's method is only the best when there's only one calculation

↑ all of this is called Discrete Fourier Transform

* Inverse Fourier Transform
- Discrete Fourier Transform in reverse
- Given the evaluations, find the coefficients
- Want to make a non-recursive version
  + Need to make the step $p(z) = z p_0(z^2) + p_E(z^2)$

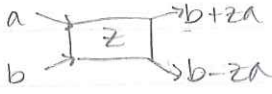
$p_0(z^2)$

$p_E(z^2)$

input ⟶ output: $p_E(z^2) + z p_0(z^2)$

$p_E(z^2) - z p_0(z^2)$

$$\begin{bmatrix} a \\ b \end{bmatrix} \rightarrow \begin{bmatrix} b + za \\ b - za \end{bmatrix}$$




- Use a butterfly switch

- Ex: $a_{15}\ a_{14}\ a_{13}\ a_{12}\ a_{11} \dots a_1\ a_0$
  ↳ Split: $a_{15}\ a_{13}\ a_{11} \dots$
  $a_{14}\ a_{12}\ a_{10} \dots$
  ↳ Split: $a_{14}\ a_{10}\ a_{10}\ a_2$
  $a_{12}\ a_8\ a_4\ a_0$
  ↳ Split: $a_{12}\ a_4$

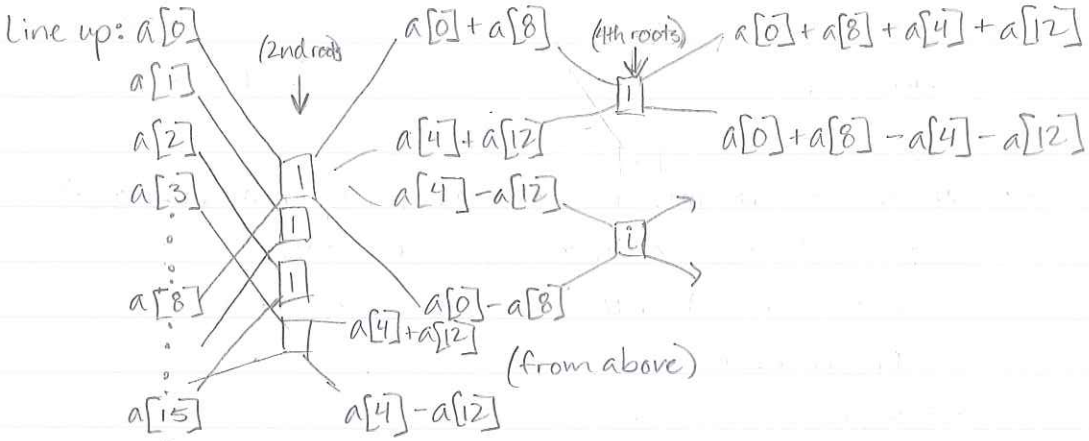
$a_8 a_0$

Look at binary version:

$a_{15}\ a_{14}\ a_{13}\ a_{12}\ a_{11} \ldots a_1\ a_0$
1111 1110 1101 1100 1011 ... 0001 0000

Look at last split from above:

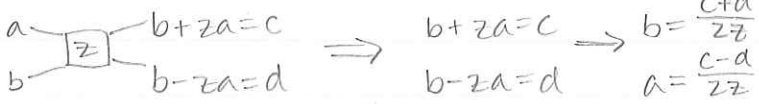
$a_{12}\ a_4 \rightarrow 1100\ \ 0100$ } all end in 2-zeros
$a_8\ a_0 \quad 1000\ \ 0000$ }




+Have: $a[12]z^3 + a[8]z^2 + a[4]z + a[0]$

+Note: Google "butterfly switch fft" so you avoid getting etsy + pinterest

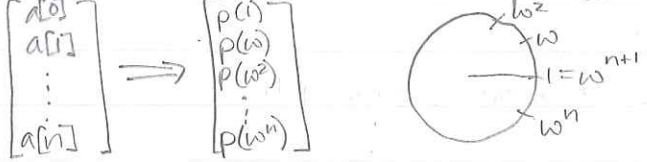
-Invert switch to reverse order




+still $\Theta(n \log n)$

---

3-9 * Fourier Transform takes coefficients as input and outputs $p(z)$ where $n+1=2^k$ for some $k$




-linear transformation

-Ex: $p(z) = a[n]z^n + a[n-1]z^{n-1} + \ldots + a[1]z + a[0]$

$p(1) = a[n] + a[n-1] + \ldots + a[1] + a[0]$

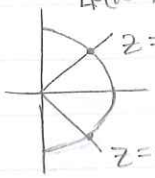
$$\begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & & \omega^n \\ 1 & \omega^2 & \omega^4 & \cdots & & \omega^{2n} \\ \vdots & & & & & \\ 1 & \omega^n & \omega^{2n} & \cdots & & (\omega^n)^n \end{bmatrix} \begin{bmatrix} a[0] \\ \vdots \\ a[n] \end{bmatrix} = \begin{bmatrix} p(1) \\ p(\omega) \\ p(\omega^2) \\ \vdots \\ p(\omega^n) \end{bmatrix}$$

$M$

$M = [\omega^{ij}]$ $0 \le i \le n$, $0 \le j \le n$

* Inverse Fourier Transform is accomplished by multiplying the following:

$M^{-1}\begin{bmatrix} p(1) \\ p(w) \\ \vdots \\ p(w^n) \end{bmatrix}$ - so inverse equation is $\bar{M} = [\bar{w}^{ij}]$ where $w \cdot \bar{w} = 1$ and $\bar{w} = w^{-1}$




$+ M \cdot \bar{M} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \bar{w} & \bar{w}^2 & \cdots & \bar{w}^n \\ 1 & \bar{w}^2 & \bar{w}^4 & \cdots & \bar{w}^{2n} \\ \vdots & & & & \vdots \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & \cdots & w^n \\ 1 & w^2 & w^4 & \cdots & w^{2n} \\ \vdots & & & & \vdots \end{bmatrix} = \begin{bmatrix} (n+1) & 0 & 0 & \cdots & 0 \\ 0 & (n+1) & 0 & \cdots & 0 \\ 0 & 0 & \ddots & & \\ \vdots & \vdots & & & \\ 0 & 0 & & & \end{bmatrix}$

- Note: $S = 1 + w + w^2 + w^3 + \cdots + w^n$

$wS = w + w^2 + w^3 + \cdots + w^n + 1$ ← same as S, so $S = 0$

- Final matrix: $(n+1)I$ where $I$ is the identity matrix

+ so $\frac{1}{n+1}\bar{M} = M^{-1}$ ✱

- Problem: given 2 polynomials $p(z)$ and $q(z)$, find $p(z) \cdot q(z)$

$(a_n z^n + a_{n-1} z^{n-1} + \cdots + a_0)(b_n z^n + b_{n-1} z^{n-1} + \cdots + b_0)$

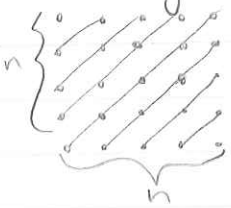
$= a_0 b_0 + (a_1 b_0 + a_0 b_1) z + (a_2 b_0 + a_1 b_1 + a_0 b_2) z^2 + \cdots + (a_n b_{n-1} + a_{n-1} b_n) z^{n-2} + (a_n b_n) z^n$

↑ notice the numbers of a + b add to n

+ multiplications: $1 \quad 2 \quad 3 \quad \cdots \quad (n+1) \quad n \quad \cdots \quad 2 \quad 1 = (n+1)^2$

+ additions: $0 \quad 1 \quad 2 \quad \cdots \quad n \quad (n-1) \quad \cdots \quad 2 \quad 1 = n^2$

+ essentially adding dots in nxn array diagonally




+ Fourier transform: $\begin{bmatrix} a[0] \\ \vdots \\ a[n] \end{bmatrix} \to \begin{bmatrix} p(1) \\ p(w) \\ \vdots \\ p(w^n) \end{bmatrix}$ and $\begin{bmatrix} b[0] \\ \vdots \\ b[n] \end{bmatrix} \to \begin{bmatrix} q(1) \\ q(w) \\ \vdots \\ q(w^n) \end{bmatrix}$

→ $O(n \log n)$

- multiply together to get: $\begin{bmatrix} p(1)q(1) \\ p(w)q(w) \\ \vdots \\ p(w^n)q(w^n) \end{bmatrix}$ ($O(n)$ steps) $\xrightarrow[O(n \log n)]{\text{inverse transform}} \begin{bmatrix} c[0] \\ c[1] \\ \vdots \\ c[n] \end{bmatrix}$

- overall, $O(n \log n)$ steps in either direction

* Is P = NP?

+ P = polynomial time algorithm

+ if the algorithm solves the problem in $O(n^k)$ steps with $k > 0$, then we say the algorithm is polynomial time

+ n = size of input = # of bits

+ N = non-deterministic

+NP = non-deterministic polynomial
- means the answer can be checked in polynomial time
+ outputs will only be true/false
+ can we check our answer in polynomial time?

---

3-11 * $P \subseteq NP$ at least

* Is $P = NP$?


$P \neq NP$?


- there should be problems with no polynomial time solution but with polynomial time verification

- nobody actually knows if $P = NP$

- Ex: The Subset Problem:

- Given the set of negative integers $S$, determine whether there is a subset $T$ such that $\sum_{x \in T} x = 0$.

+ For example, $S = \{5, 2, -8, 3, -4, -6, 12\}$

$T_1 = \{12, -4, -8\}$

$T_2 = \{5, 3, -8\}$

$T_3 = \{5, 2, 3, -4, -6\}$

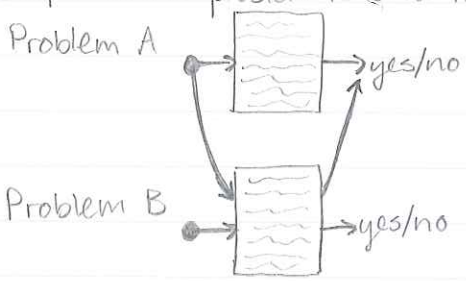
$\vdots$

+ varification is quick but finding a unique T is a long process (exponential time)

+ If $P = NP$, there is no algorithm in polynomial time

+ If $P \neq NP$, there is a solution in P time, but no one has found it yet

* Find a super hard problem to show it's not polynomial time to compare




- If we have a polynomial time solution to B + a polynomial time conversion so that the yes/no answer in B is the yes/no answer for B, then there is a polynomial time solution for A.

- Easiest to show there is no poly time solution for B

- The "hardest" in NP would be one for which there is a polynomial time conversion to solve other NP problem. Such problems are called **NP Complete**

+ If you can show a NP complete problem is polynomial time, then $P = NP$ for all other problems

* Satisfiability is our first NP complete to use as a base

3-13 * HW stuff:

$$x^2-2 \overline{)\, x^3+5x^2-7x+3} \quad x$$

$$(x^3 \quad -2x)$$

$$5x^2-5x+3$$

$$-(5x^2 \quad -10)$$

$$-5x+13 \rightarrow \frac{x^3+5x^2-7x+3}{x^2-2} = x+5+\frac{-5x+13}{x^2-2}$$

$$= x^3+5x^2-7x+3 = (x+5)(x^2-2)+(-5x+13)$$

$\Pi_1^{n/2}(x-x_i)$

$\Pi_{n/2+1}^{n}(x-x_i) \rightarrow p(x) = Q(x)\Pi_1^{n/2}(x-x_i) + R(x)$

Want to calculate $p(x_1) \ldots p(x_n)$

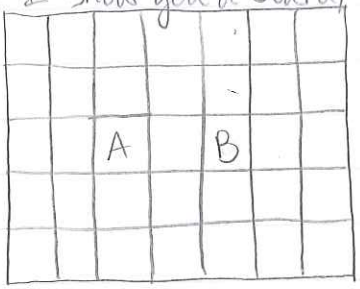
$p(x_i) = R(x)$ ↑ (from above)

Since $R(x)$ is $O(n \log n)$, show why $p(x)$ is $O(n \log^2 n)$

* The harder the NP problem, the easier it is to prove that it is not polynomial time, so P ≠ NP

* Satisfiability
  - boolean expressions given → try to make it true by adding 1s + 0s
  - Ex: $(x_1 \vee x_2 \vee x_3) \wedge (\_\ \_\ \_) \wedge (\_\ \_\ \_\ \_\ \_) = 1$

* Minesweeper Problem (NP Complete)
  - If I show you a board, is it a legitimate minesweeper board?
    + Ex: if A = 8 + B = 0, it is not a board




  - Richard Kaye → showed problem is NP complete by converting satisfiability problem in polynomial time