9/4/25

ALGORITHM: (mechanized)

1: correctness on every possible input

2: efficiency (metrics: time, memory,)

asymptotic, $O(n)$, hidden constants

3: optimality

ADDING INTEGERS: two ints x & y, compute $x + y$

preschool: $x + \overbrace{1+1+\dots1}^{y \text{ times}}$

ex: $n=3 \rightarrow 123$ (n & #digits is exponential)

$O(10^n)$

grade-school: $\begin{array}{r} 123 \\ +\,456 \\ \hline 579 \end{array}$ adding 3 digits n times

1 → need $2n$ digits (2 n-digit #s) 2: $O(n)$

2 → $n+1$ digit # an output

MULTIPLICATION: $x * y$

$\begin{array}{r} 123 \\ \times\,456 \\ \hline \end{array}$ single dig. mult. $x_i * y_j$ for $\forall i,j$ (multiplies each pair)

$\Theta(n^2)$ multiples

is it optimal?? → $O(n)$ is possible (read / write)

DIVIDE & CONQUER (assume n = even) (no floors/ceil)

Ex: $x = 1234$ … same for y

$x_{hi} = 12$

$x_{lo} = 34$

multiply (x, y)

base case: $n=1$

recursion: $x_{lo} = 10^0 x_0 + \dots + 10^{n/2-1} x_{n/2-1}$ lower order (R)

higher order (left) $x_{hi} = 10^0 x_{n/2} \dots + 10^{n/2-1} x_{n-1}$ $n/2$ digs

$x * y = (x_{hi} \cdot 10^{n/2} + x_{lo})(y_{hi} \cdot 10^{n/2} + y_{lo})$

$= (x_{hi} x_{lo})\, 10^n + (x_{hi} y_{lo} + x_{lo} y_{hi})\, 10^{n/2} + x_{lo} y_{lo}$

$A = \text{mul}(x_{lo}, y_{lo})$

$B = \text{mul}(x_{lo}, y_{hi})$

$C = \text{mul}(x_{hi}, y_{lo})$

$D = \text{mul}(x_{hi}, y_{hi})$

→ out: $A + 10^{n/2}(B+C) + 10^n D$

BASIC OPS: +, −, *, shift

$T(n)$: # basic ops for multiply (x, y) (n digits)

$T(n) = 4T(n/2) + c_1 n$

RECURRENCE SOLN: masters thm → $\Theta(n^2)$

KARATSUBA: 1st math algo. better than $\Theta(n^k)$ $k<2$

Master theorem: Let T be a recurrence of the form $T(n) = aT(n/b) + g(n)$.
Case 1: If $g(n) = O(n^{\log_b(a)-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b(a)})$.
Case 2: If $g(n) = \Theta(n^{\log_b(a)} \log^k(n))$ for some constant $k \geq 0$, then $T(n) = \Theta(n^{\log_b(a)} \log^{k+1}(n))$
Case 3: If $g(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some constant $\epsilon > 0$ and $ag(n/b) < cg(n)$ for some constant $c < 1$ and sufficiently large n, then $T(n) = \Theta(g(n))$.

KARATSUBA:

A = multiply (x_{lo}, y_{lo})

B = (x_{lo}, y_{hi})

E = multiply ($x_{lo} + x_{hi}$, $y_{lo} + y_{hi}$)

only getting needed vals (3 mults instead of 4) - don't need B & C seperately

$B + C = E - A - D$

$A + (B+C)\,10^{n/2} + 10^n D$

$= (x_{hi} + x_{lo})(y_{hi} + y_{lo}) = A + B + C + D$

$B + C = (x_{hi} + x_{lo})(y_{hi} + y_{lo}) - A - D$

$T(n) = 3T(n/2) + c_2 n$ ← apply master thm

$\Theta(n^{\log_2 3})$ $\log_2 3 \approx 1.58$

3 multiplications, each on size $n/2$

HIGH PRECISION DIVISION:

- requires many bits
- need large integer multiplication

CORRECTNESS: right answer on all inputs

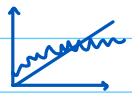
EFFICIENCY: if it makes least/small use of resources (time, space, ...)

WORST CASE ANALYSIS:

- $T(n)$ = number of "operations" algorithm takes on worst input
- Asymptotics to summarize $T(n)$ → { $O(f(n))$ ← upper, $\Omega(f(n))$ ← lower } $\Theta(f(n))$

DEF: BIG-O

$O(f(n))$ algorithm A has (worst-case) runtime $O(f(n))$ if $\exists$ constants $c, n_0 > 0$ s.t. on all inputs of length $n > n_0$ it holds worst case runtime of A $\leq c \cdot f(n)$



$\Omega(f(n))$ algorithm A has (worst-case) runtime $\Omega(f(n))$ if $\exists$ $c, n_0 > 0$ s.t. an input for all $n > n_0$ s.t. A takes time $\geq c \cdot f(n)$ on that input

CAVEAT: alg A $T(n) \leq 2^{1000} \cdot n \in O(n)$ asymptotically: $O(n)$ beats $O(n^2)$

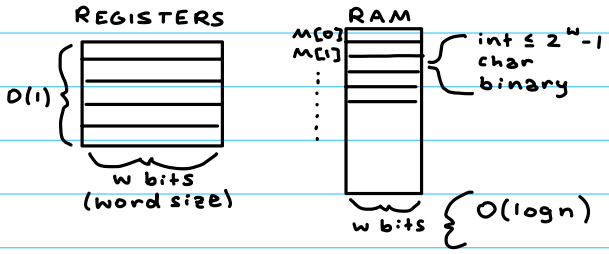
alg B $T(n) \leq n^2 \in O(n^2)$ practically: constants matter. for $n \leq 2500$, $O(n^2)$ alg. might be faster.

PROGRAM	⟷	algorithm
C/C++/RUST	⟷	pseudo code / english pose
ASSEMBLY (x86/RISCV)	⟷	base ops
COMPUTER	⟷	model as computation

WORD-RAM MODEL:

- mem
- registers
- processor

int math: *, +, //, %/
logic: and, or...
bitwise: &, |, ~
load: R[j] ← M[R[i]]
store: M[R[i]] ← R[j]
branch
halt



EXECUTION ON WORD-RAM:

1) programs input starts out in mem locations M[0],...,M[n-1]

2) processor steps through alg until HALT

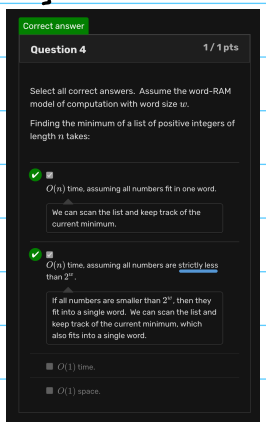
3) output must be in first 17 mem. locations

2^w addresses ≥ n input size — input len n, word len $w = O(\log n)$

$w \geq \log_2 n$ w=word

$w \geq 2\log_2 n \rightarrow n^2$ mem. locs.

*use Pythonic model instead & under the hood is word RAM


Correct answer

Question 4 — 1 / 1 pts

Select all correct answers. Assume the word-RAM model of computation with word size w.

Finding the minimum of a list of positive integers of length n takes:

- ✓ $O(n)$ time, assuming all numbers fit in one word.
  - We can scan the list and keep track of the current minimum.
- ✓ $O(n)$ time, assuming all numbers are strictly less than 2^w.
  - If all numbers are smaller than 2^w, then they fit into a single word. We can scan the list and keep track of the current minimum, which also fits into a single word.
- $O(1)$ time.
- $O(1)$ space.

COMMON python objects & running times:

INTEGER	x + y	Schoolbook	$O(\lvert x\rvert + \lvert y\rvert)$
	x * y	karatsuba	$O((\lvert x\rvert + \lvert y\rvert)^{\log 3})$ ← $T(m) = 3T(\frac{m}{2}) + O(m)$
			$\approx O(m^{\log_2 3})$
LIST	L.append(x)		?
	len(L)		O(1)

PEAK FINDING

INPUT: a list "A" of n integers

OUTPUT: a "peak" of list A

index is "peak" of A if A[i] is at least as large as neigh.

[1,3,4,2]
0 1 2

[1,1,1,1,1]

ALG FOR PEAK FINDING:

- find_peak(A):

```
n = len(A)          // Θ(1)
for i = 0,...,n-1:  // Θ(n)
  if i is peak return i  // Θ(1)
```

runs in worst-case time linearly

O(n) worst case

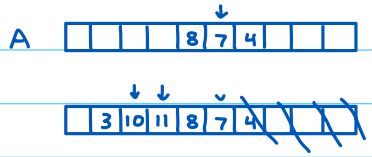
STEPS TO SUCCESS in algs:

1) write alg. in English prose
   ↳ pseudocode if necessary

2) prove alg. is correct on all inputs

3) analyze running time

DIVIDE & CONQUER ALG (more efficient peak finding)



$T(n) \leq O(1) + T(\lceil n/2 \rceil)$

$\leq O(\log n)$

```
find_peak(A):
  n = len(A)
  if n == 1: return 0
  j = n/2
  if A[j] peak: return j
  if A[j] < A[j-1]: return find_peak(A[:j])
  else:             return find_peak(A[j+1:])
```

9/11/25 SEQUENCES + AMORITIZATION

DATA STRUCTURES

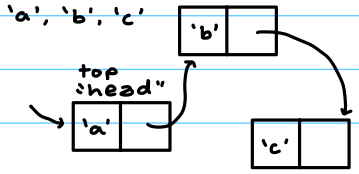
2 COMPONENTS:

- interface (api, adt)
- implementation

STACK: LIFO access pattern

interface:
- push(x): stores x on "top" of stack
- pop: remove & output "top" of stack
- peek: return "top" of stack w/o removing
- is empty: check if empty

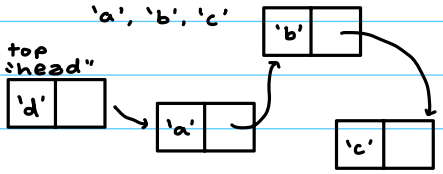
} constant time!!



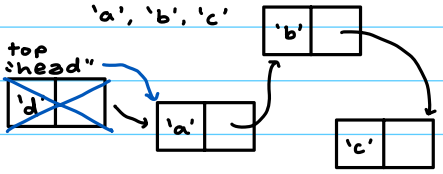
LL:

LINKED LIST:

- pointer-based data structure

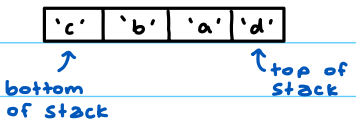


push('d') — constant time



pop() — constant time

ARRAY:



STACK:

d
a
b
c

pop()

push('e')

* problem: fixed-len arrays. we need to reallocate a new array & will take O(n) time

HOWEVER, if we copy into much larger array, can do many operations

AMORTIZATION:

- charge expensive operations to the cheaper ones

Ex: teaholic buy 9 get 1 free

- averaging cost over sequence of operations & bounding
- amortized cost T if any sequence of m ops has total cost $\leq m \cdot T$
  - T: function of state of data structure
  - start from empty data structure

Ex: what is amortized cost of array if we make additional n slots? → constant time

cheap: total cost of m pushes (starting from empty data struct): $m \cdot \Theta(1)$

expensive: total cost of copy operations: $\sum_{i=1}^{\lfloor \log m \rfloor} \Theta(2^i)$ ← happens whenever you hit a power of 2

Sum: $m \cdot \Theta(1) + \sum_{i=1}^{\lfloor \log m \rfloor} \Theta(2^i) = \Theta(m)$

amortized cost: $\Theta(m) / m \rightarrow \Theta(1)$ amortized (constant)

* PROBLEM: push many values then pop them all, we end up with a massive empty array (lots of memory)

MEMORY OVERHEAD:

additional space used beyond what's actually used to store the data.

- dynamic arrays: linear $\Theta(n)$
- linked list: linear

allowed to use several diff. ops, each of which we can use diff. T?

POTENTIAL:

if $\exists$ Φ...

- Φ: function
- data struct

$\Phi(A) \geq 0$

operation that takes $A_0 \rightarrow A_1$ takes time t: $t + \Phi(A_1) - \Phi(A_0) \in O(f(|A_0|))$

$\Rightarrow O(f)$ amortized cost

array potential func:

$\Phi(A) = c|n - m/2|$

n = # elements in stack

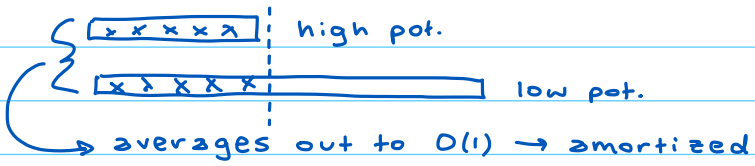
m = capacity of array

- potential decrease by linear amount

Ex: push w/o copy $\Theta(1) + \Theta(1) = \Theta(1)$ cost

Ex: push with copy $\Theta(n) - \Theta(n) = \Theta(1)$

- $\Theta(n)$: cost
- $\Theta(n)$: potential difference



SEQUENCE: maintains sequence $x_0, x_1, \ldots, x_{n-1}$

ITER-SEQ(): iterates over sequence

BUILD(A): build sequence out of A ← container op

GET-AT(i): output x_i

SET-AT(i, x): set $x_i = x$ } static op

INDEX-OF(x): output i s.t. $x_i = x$

INSERT (at, first, last)

DELETE (at, first, last)

SEQUENCE	Operations					
	Container	Static		Dynamic		
	build	get_at set_at	index_of	insert_first delete_first	insert_last delete_last	insert_at delete_at
Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Dynamic Array	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$ (Am.)	$\Theta(n)$
2-End Dynamic Array	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$ (Am.)	$\Theta(1)$ (Am.)	$\Theta(n)$

(Am.) = **amortized** running time

9/16 **SETS (key-value maps)**

EX: (38, Henry)
(63, Srini)
(35, Brynmor)

{(key, value)} pairs

keys are comparable, unique

OPERATIONS:

$\Theta(n)$ • build(A)

• length()

$\Theta(n)$ • find(key) → returns value associated w/ key

$\Theta(n)$ • insert(key, value) might be able to do in amortized $\Theta(1)$

$\Theta(n)$ • delete(key)

• find_min() → returns value w/ lowest key

• find_max()

• find_next(key) → returns value w/ smallest key larger than key

• find_prev(key) → returns value w/ largest key smaller than key

$\Theta(n^2)$ • iter_ord() → each find takes n, so all finds will be n^2 time

* if we sort array, time could be $\Theta(\log n)$

ARRAY:

SORTING:

input: list or array, A, of comparable elements

output: list or array, B, permutation of A that is sorted.

$\forall i < |A|, B[i] \leq B[i+1]$

PERMUTATION SORT: $\Omega(n!) >>> 2^n$

- generates permutations of elements until sorted

INSERTION SORT *stable & in place

- goes through 1 by 1 each element & sorts into right place

EX: 8 6 7 5 3 0 9

1: 6 8 7 5 3 0 9

2: 6 7 8 5 3 0 9

⋮

```
insert_last(A, i)                    WORST CASE: every iteration
    while i ≥ 0 and A[i] > A[i+1]:   O(n)   Ω(n) ← @ ith position, A[i] smaller than anything in prefix
        swap A[i] with A[i+1]        O(1)
        i--                          O(1)

insertion_sort(A):                   Θ(n²)
    for i in 1...|A|:                O(n)
        insert_last(A, i)            O(n)
```

in-place: $\Theta(1)$ memory overhead (modifying input directly - no extra mem.)

STABLE SORT: preserves order of equal items

in: (2,3), (1,1), (2,4)

out 1: (1,1), (2,3), (2,4) ← valid & stable

out 2: (1,1), (2,4), (2,3) ← another valid but unstable out

MERGE SORT: divide + conquer

EX: 8 6 7 5 3 0 9 • has many edge cases

6 7 8 0 3 5 9 } merge $\Theta(n)$

0 3 5 6 7 8 9

```python
def merge_sort(A, a = 0, b = None):                 # T(b - a = n)
    '''Sort A[a:b]'''
    if b is None: b = len(A)                        # O(1)
    if 1 < b - a:                                   # O(1)
        c = (a + b + 1) // 2                        # O(1)
        merge_sort(A, a, c)                         # T(n / 2)
        merge_sort(A, c, b)                         # T(n / 2)
        L, R = A[a:c], A[c:b]                       # O(n)
        merge(L, R, A, len(L), len(R), a, b)        # S(n)

def merge(L, R, A, i, j, a, b):                     # S(b - a = n)
    '''Merge sorted L[:i] and R[:j] into A[a:b]'''
    if a < b:                                       # O(1)
        if (j <= 0) or (i > 0 and L[i - 1] > R[j - 1]): # O(1)
            A[b - 1] = L[i - 1]                     # O(1)
            i = i - 1                               # O(1)
        else:                                       # O(1)
            A[b - 1] = R[j - 1]                     # O(1)
            j = j - 1                               # O(1)
        merge(L, R, A, i, j, a, b - 1)              # S(n - 1)
```

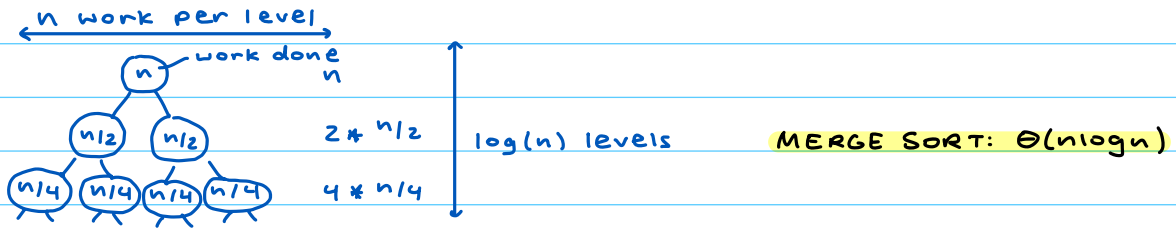
*not in place

*can make stable by preferring left array

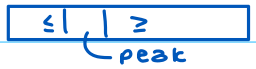
merge_sort(A): pseudocode

```
    size of A
if |A| < 2: return A   (already sorted)

mid = |A|//2

L = merge_sort(A[:mid])

R = merge_sort(A[mid:])

return merge(L, R)
```

$T(n) = \Theta(n) + 2T(n/2)$ each level is n work! equal!!

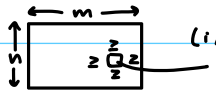


2D PEAK FINDING:



* looking for ANY peak, not max peak

· find_peak_1D(A) = Θ(logn)

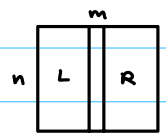


(i,j) is peak if $A[i,j] \geq A[i \pm 1, j]$ &
$A[i,j] \geq A[i, j \pm 1]$

GOAL: algorithm that returns (i,j) a peak in A

BRUTE FORCE: $\Theta(n \times m)$

IDEA: **DIVIDE + CONQUER** ← THIS IS A BUGGY SOLUTION



$jmid = \lfloor \frac{m}{2} \rfloor$

col_peak = find_peak_1D

col_peak returns some value i s.t. (i,j) is a peak in column j.

Check if (i,j) is peak in A.

CASE 1: it is. return (i,j)

CASE 2: it isn't. didn't fail N or S b/c we used col_peak(). must have failed E or W. recurse in uphill direction (L or R).

PROOF: induction on m.

INDUCTIVE HYPOTHESIS (preposterous): assume the alg works for arrays up to m-1. INDUCTION: show it works for m columns

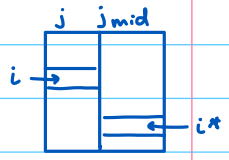
BASE CASE: m=1: 1D peak finding, which we know to be correct & returns a peak in the column, which is a peak in A b/c 1D.

acquire jmid & call col_peak(A[i, jmid]) → i*

CASE 1: (i*, jmid) is peak in A. ✓

CASE 2: A[i*, jmid] has greater value in L or R

WLOG (without loss of generality), higher value is in L. recursively call on L.



(i, j). A[i, j] is peak in L. b/c of preposterous hypothesis

CASE 2a: j < jmid-1. by inductive hypothesis (i, j) is peak in A.

CASE 2b: j = jmid-1. show (i,j) is peak in A.

BUG: we don't know relationship b/t i & i*. need to establish a relationship & can't use 1D peak finder — must find max peak of columns

NEW INDUCTIVE HYPO: assume alg. works up to m-1. ALSO assume it returns a max in col j.

9/18 **FASTER SORTING**

so far...

- insertion $\Theta(n^2)$
- selection $\Theta(n^2)$
- merge $\Theta(n\log n)$

COMPARISON MODEL:

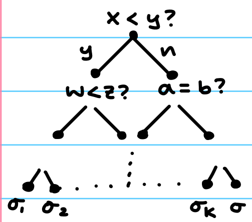
comparable: black box that supports binary comparison operations

$<, =, >, \leq, \geq, \neq$

INPUT: list $x_1, x_2, \ldots, x_n$. key is comparable

OUT: $\sigma \in S_n$ s.t. $x_{i\sigma}$ key are increasing with i.

DECISION TREE:



*need @ least 1 leaf for each permutation

every permutation must have ≥ 1 leaf

POSSIBLE σ's $\geq n!$ leafs

ROOT-TO-LEAF PATH ↔ runtime

HEIGHT: worst case runtime $\geq \log(n!) \geq n\log n$

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

$= \frac{n}{2}\log\frac{n}{2} \in \Omega(n\log n)$

*branches always binary (2 per level)

*need superconstant branching to beat $n\log n$

REFERENCING / DEREFERENCING: $A[B[i]]$

(direct access array)

DAA-SORT: assume all keys are $\mathbb{N} \cap [0, u)$ & DISTINCT

$A[x.\text{key}] = x$

- stable by default b/c nothing can appear twice

(no duplicate keys to preserve man...)

[5, 7, 2, 0, 4]
↓
u = 7+1 = 8

[0 - 2 - 4 5 - 7] array of size 8
0 1 2 3 4 5 6 7

[0 2 4 5 7] copy array back

typically, just use this instead → ★

what if keys not distinct??

COUNTING SORT

→ allocate a sequence within list

sorting by #s

CHAR	0 0 6
PIKA	0 2 5
SNOR	1 4 3
MCHAR	0 0 6

key collision for diff. values

[0 1 2 ... 6 ... 143]
[CHAR, MCHAR]

- maintain stability: must ensure sequence memorizes insertion order (queue, etc)

- store "chain" or queue of all elts. w/ same key
- concat all chains together to create output list
- stable, but not in place (uses extra mem. to sort)

assume keys are tuples of equal length.

sort by Col A, Col B lexicographically

TUPLE LENGTH: sort by key components individually in order of significance

Ex: Sort cards by suit (clubs, diamonds, hearts, spades) then #

- counting sort: put cards into suit order, sort within suit, put all back together OR sort by #s, then by suit ⇒ same output!

TLDR: doesn't matter whether we sort by less/more significant key; if sort by less sig. first, need to make sure we preserve order (stability)

RADIX SORT: what if u is larger? e.g. $u = n^2$

$3n + 5 \rightarrow (3, 5)$

TUPLE SORT using **COUNTING SORT** as our auxillary algorithm

keys in base n

MUST SAY: "use tuple sort with auxillary counting sort alg"

Ex: $[17, 3, 24, 22, 12] \rightarrow [(3,2), (0,3), (4,4), (4,2), (2,2)]$

0	1	2	3	4
(0,3)		(2,2)	(3,2)	(4,2)
				(4,4) ← 4·5+4=24

$[3, 12, 17, 22, 24]$

$\Theta(n + n\log_n u)$

when $\log_n u \in O(1)$ if u is small:

→ sorting takes linear time

n^4 ✓, $n\log n$ ✓, 2^n ✗

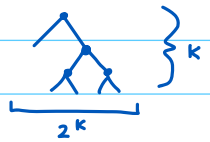
SORTING ALGS:	RUNTIME:	STABLE?	IN PLACE?
insertion	$\Theta(n^2)$	✓	✓
selection	$\Theta(n^2)$	✗	✓
merge	$\Theta(n \log n)$	✓	✗
DAA	$\Theta(n+u)$	✓	✗
counting	$\Theta(n+u)$	✓	✗
radix	$\Theta(n+u)$	✓	✗

9/23

	BUILD	FIND	INSERT/DELETE	FIND-PREV / FIND-NEXT
ARRAY	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
SORTED ARRAY	$\Theta(n \log n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$
DAA	$\Theta(u)$	$\Theta(1)$	$\Theta(1)$	$\Theta(u)$
HASH TABLE	$\Theta(n)$	$\Theta(1)$ exp.	$\Theta(1)$ exp./amor.	$\Theta(n)$

bottom line: know when to use which

- SORTED ARRAY: range queries
- HASH TABLES: lookups by key
- ARRAYS: almost never



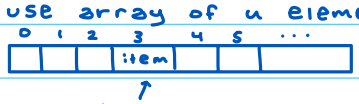
find(key) → item

$2^k \geq n \Rightarrow k = \log_2 n$

*dict is implemented w/ hash table

WARM-UP: O(1)-time find w/ huge memory overhead

use array of u elements: if all keys are in $\{0, 1, \ldots, u-1\} = [u]$



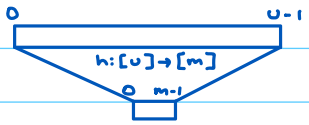
← direct-access array

insert(3, item)

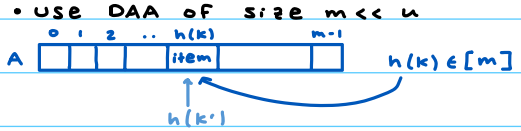
find(5)

Ex: student id's $n < 10^9$ id #S → n students, $u = 10^9$ → lots of wasted space

HASHING:

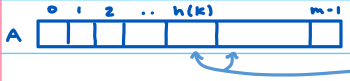


- USE DAA of size $m << u$



- collision when $k \neq k' \in [u]$ s.t. $h(k) = h(k')$

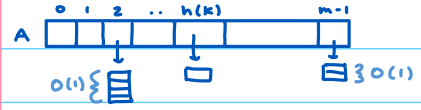
HANDLING COLLISIONS: open addressing



- store h(k') & item in next available empty bucket
- don't need another data struct
- however, insert/delete makes alg. complicated

SIZING:
n items
m table size
$m \geq n$

HANDLING COLLISIONS: chaining



- each "bucket" (spot) now points to another DAA
- hash table shouldn't be too small b/c we want DAAs in each index to be small

find(k): $\Theta(1+L)$

```
c = A[h(k)]      Θ(1)
return c.find(k) Θ(L)
```

insert(item):

```
c = A[h(item.key)]
c.insert(item)
```

Ex: Find a collision

$h: \mathbb{Z} \to [10]$ $\quad h_1(x) = x \bmod 10$

$h_2(x) = x \bmod 10000$ *seems like it's doomed b/c of Pigeonhole

$h_2(1) = h_2(10001)$

$h(x) = \lfloor \cos(\tan^{-1}(x+15)) \rfloor$

*pick hash function in secret after given keys.

SIMPLIFYING ASSUMPTION: have data structure pick truly random hash function

↳ hash is picked independently of the keys $h: [u] \to [m]$

- SUHA

"EXPECTED" TIME for find/insert:

- imagine we've inserted items w/ distinct keys $k_1, k_2, \ldots, k_n \in [u]$
- insert $k_{new} \notin \{k_1, k_2, \ldots, k_n\}$
- look @ prob. that k hashes to same bucket as k_{new}

$$Pr[h(k_j) = h(k_{new})] = \sum_{i=0}^{m+1} Pr[h(k_j) = i \wedge h(k_{new}) = i]$$

$$= \sum_{i=0}^{m+1} Pr[h(k_j) = i] \cdot Pr[h(k_{new}) = i]$$

$$= m \cdot \left(\frac{1}{m}\right) \cdot \left(\frac{1}{m}\right) = \frac{1}{m}$$

collisions @ $h(k_{new})+1$

the expected chain length @ $h(k_{new})$ = $\mathbb{E}[L] = 1 + \sum_{j=1}^{n} Pr[h(k_j) = h(k_{new})] = 1 + \frac{n}{m}$ ($\frac{n}{m}$ = LOAD FACTOR = α; $\frac{\#\text{ items}}{\text{table size}}$)

$= 1 + \alpha$

EXPECTED:

time to insert: $1 + \alpha = 1 + \frac{n}{m} = 1 + \frac{n}{3n} \in O(1)$

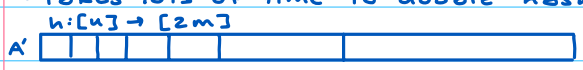
STORAGE FOR TABLE: $m+n$ $\quad m > n$, $m = \Theta(n)$

m = table size

n = # total items inserted

L = size of the bucket

- takes lots of time to double hash table size like dynamic array.



Ex: python: hash("cow") → always get same # until you quit python.

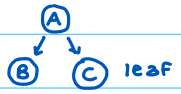
- starting Python over → get a different #
- good for preventing hackers

9/25 **BINARY TREES & BINARY SEARCH TREES:**

DATA STRUCTURE:	BUILD	FIND	INSERT/DELETE	FIND_MIN/FIND_MAX	FIND_PREV/FIND_NEXT	. . . .
SORTED ARRAY:	n log n	log n	n	1	log n	
HASH TABLE:	n (e)	1 (e)	1 (a) (e)	n	n	
GOAL:	n log n	log n	log n	log n	log n	

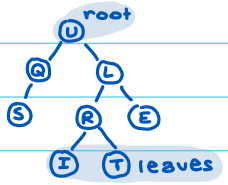
EX: key prime
x o/o P

BINARY TREE:



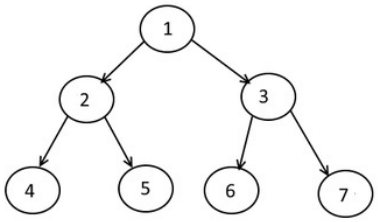
B = left child of A
C = right child of A

A = parent of B & C



DEPTH of node: d(x) = # of ancestors
→ root = 0, leaf = 3

HEIGHT of node: h(x) = # edges of longest path from node to a leaf
→ root = 3, leaf = 0
→ h = O(log n)



Inorder Traversal: 4 2 5 1 6 3 7
Preorder Traversal: 1 2 4 5 3 6 7
Postorder Traversal: 7 6 3 5 4 2 1
Breadth-First Search: 1 2 3 4 5 6 7
Depth-First Search: 1 2 4 5 3 6 7

TRAVERSALS: O(n), n = # nodes

IN-ORDER: visits first, recursively the left child, root, right child
→ S, Q, U, I, R, T, L, E

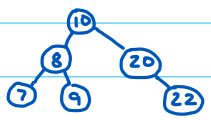
PRE-ORDER: visits recursively the root, then left, then right nodes
→ U, Q, S, L, R, I, T, E

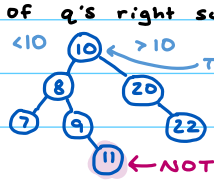
POST-ORDER: visits recursively the left, then right, then node.
→ SQITRELU

BINARY SEARCH TREES: # distinct keys

for every node q, all the keys stored in the left subtree of q are smaller than q's key & all the keys stored in the nodes of q's right subtree are greater than q's key.

valid BST:





T.item = 10
T.left, T.parent, T.right

←NOT BST b/c everything on left of root should be strictly smaller, but 11 !< 10

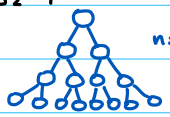
find(T, k): find node in T whose item has # key k if it exists, else None

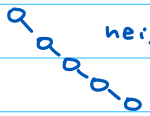
O(h) h = height

```
if T is None: return None
if k < T.item.key: return find(T.left, key)
if k > T.item.key: return find(T.right, key)
return T
```

$O(h)$

$h = O(\log_2 n)$

Ex: height = 3  n = 15

 height = 4

`find_min(T)`: keep going left until can't anymore

`find_max(T)`: keep going right until can't anymore

`find_next(T, x)`: next node after x in T's inorder traversal

CASE 1: if x has right child, successor of x is smallest element in the right subtree of x. `find_min(T.right, x)`

CASE 2: successor of x is 1st ancestor which we reach through left child.

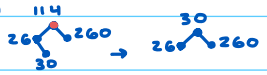
```
def find_next(T):
  # Finds next node after T in the in-order traversal
  if T is None:
    return None
  if T.right:
    return find_min(T.right)
  child = T
  while child.parent and child.parent.right == child:
    child = child.parent
  return child.parent
```

`delete(T, x)`: $\Theta(h)$

CASE 1: if node is a leaf, just delete 

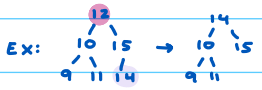
CASE 2: node only has one child, replace with child 

CASE 3: node is internal. Swap with successor (find_next)



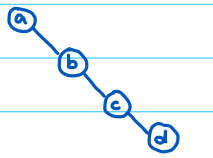
Successor will not have a left child, or else it would not be the successor.

predecessor will not have right child.

Ex: 

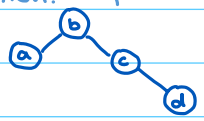
9/30

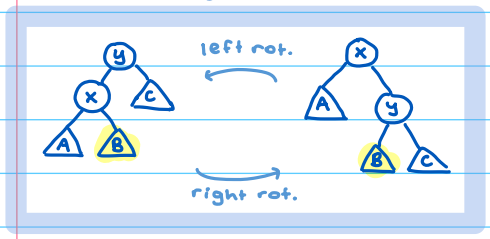
```
def in_order(T):
    if T is None: return
    in_order(T.left)
    print(T.item)
    in_order(T.right)
```



O(h)
h→n

left rotation:






```
def right_rotate(T):
    return Tree(T.left.left, T.left.item,
                Tree(T.left.right, T.item, T.right))
```

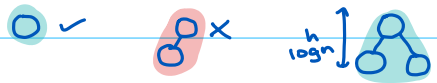
left node right

left node right

```
def left_rotate(T):
    return Tree(Tree(T.left, T.item, T.right.left),
                T.right.item,
                T.right.right)
```

left ← node ← right

PERFECTLY BALANCED TREE: if T is full @ every level! $h = O(\log n)$



✓ ✗ ✓ *hard to achieve

SKEW(n): height(n.right) - height(n.left)

AVL PROPERTY: if $\forall n \in T$, $|skew(n)| \leq 1$

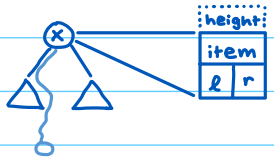
- as long as deepest leaf on one side is ≤ 1 deepest leaf on the other
- a tree with AVL property has $h \in O(\log n)$

n_h nodes:

$n_h = n_{h-2} + n_{h-1} + 1$ ← Fibonacci! exponential growth

$n_h > 2 * n_{h-2}$

$n_h > 2^{\frac{h}{2}} \Rightarrow 2\log n_h > h \Rightarrow h \in O(\log n)$



T.height: insert @ leaves

new node gets height = 0

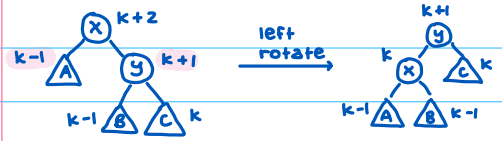
*insert & delete are logarithmic!

Deletions:

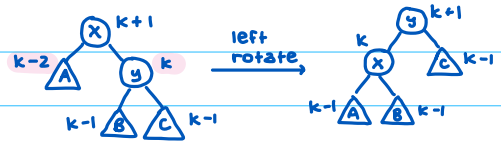
1) leaf: throw it away; update the ancestors

2) internal nodes have ≤ 1 child: replace node w/ child; update ancestors

CASE 1: skew = 2

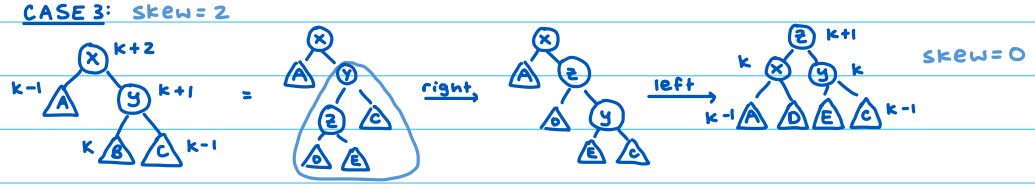


CASE 2: skew = 2



*can't happen during insertion

CASE 3: skew = 2



AVL SORT: use AVL tree to represent our set. Θ(nlogn)

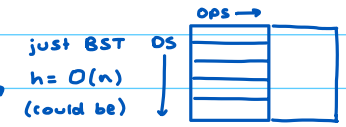
```
def avl_sort(A):
    for a in A: tree.insert(a)   ← each insert = logn, total insert [Θ(nlogn)]
    for x in tree: iter_order(): B.append(x)   ← Θ(n)
    return B
```

10/1

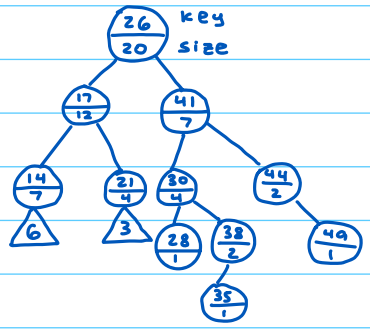
AUGMENTATION OF TREES

- dynamic order statistics ← size field
  O(logn) for dynamic sets for O.S.
- maintainery augmentations
- interval trees

last time: BST → AVL
O(h) O(logn)



ORDER-STATISTIC TREE



- get ith order statistic: element in set with ith smallest key
  give i → return element
- rank: give element → return position i in ascending order
- x.size = x.left.size + x.right.size + 1

RETRIEVING ELEMENT w/ GIVEN RANK:

OS-select(x, i):

```
r = x.left.size + 1
if i == r: return x
else if i < r: return OS-select(x.left, i)
else: return os-select(x.right, i-r)   ← shifted rank
```

OS-select(17, 17-13)
4th
4 - 2 → 2nd

FIND RANK OF ELEMENT: given pointer to node x

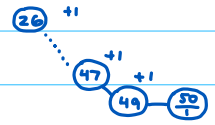
OS-rank (T, x):

```
r = x.left.size + 1
y = x
while y ≠ T:
    if y == y.parent.right:  r = r + y.parent.left.size + 1
    y = y.parent
return r
```

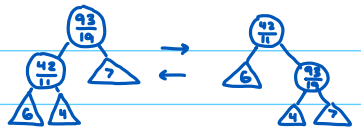
DYNAMIC SETS:

BST:

insert: +1



SECOND PHASE: AVL ROTATIONS



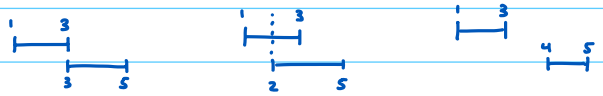
O(1) constant # of nodes with their size changing

INTERVAL TREES: interval-search (T, i) → find a node who's interval intersects i if it exists

low(i) high(i)

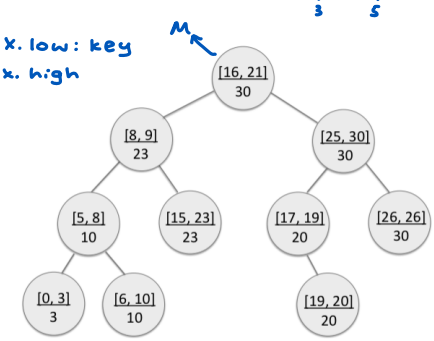
$i \equiv [t_1, t_2]$

high(i) ≥ low(i)



x.low: key

x.high



AVL BST

unsuccessful: [11, 14]

x.M = max(x.high, x.left.M, x.right.M)

Interval_Search (T, i):

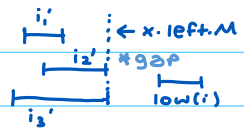
```
x = T
while x ≠ None & i doesn't overlap [x.low, x.high]:
        if x.left ≠ None and x.left.M ≥ low(i):
            x = x.left
        else: x = x.right
return x
```

Interval_Search Proof:

(a) suppose we go right

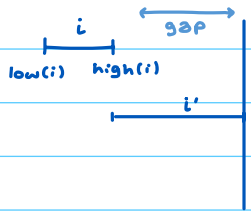
x.left == None

x.left.M < low(i)



(b) Suppose we go left

need to show if no interval in x.left overlaps then no interval in x.right will overlap



x.left.m ≥ low(i)

low(i') > high(i)

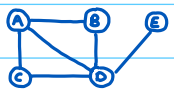
x.right low(i'') > low(i')

MIDTERM 1

10/7

GRAPHS: G = (V, E) is a set of vertices V & a set of pairs of vertices $E \subseteq V \times V$

UNDIRECTED:



- E {u,v} unordered edges
- size of E: $|E| \leq \binom{|V|}{2} = O(|V|^2)$

DIRECTED:

- vertices are ordered pairs
- E (u,v) distinct edges
- $|E| \leq 2\binom{|V|}{2} = O(|V|^2)$

SIMPLE GRAPHS:

× 1:  no duplicate edges (u,v) for unique u,v

× 2:  no self looping (u,v) for u≠v

adj⁺(u) = set of outgoing edges from u.

adj⁻(u) = set of incoming edges to adj(u) = defaults adj⁺(u) (outgoing)

deg⁺(u) = |adj⁺(u)|

deg⁻(u) = |adj⁻(u)|

how to represent in computers? → **adjacency matrix**

	A	B	C	D	E
A		1	1	1	0
B	1		0	1	0
C					
D					
E					

(u,v) index into adjacency matrix & find if there's an edge in Θ(1)

space: uses $\Theta(|V|^2)$

* matrix is <u>useful</u> when G is dense → lots of edges with respect to $|V|^2$
<u>not ideal</u> when G is sparse → not so many edges w/ respect to $|V|^2$

IDEA: use a set with Θ(1) lookup operations (DAA or Hash Table) to store **adjacency lists**

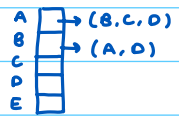
ADJACENCY LISTS: use DAA, hash table, linked list.

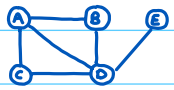
MATRIX: $\Theta(|V|^2)$ space

ADJ. SET: $\Theta(|V|+|E|)$

$\sum_{u \in V} deg^+(u) = |E|$ ← add all out degrees of vertices → # edges!

- set backed by direct array
- each elt has vertices it goes to





PATH: sequence of vertices connected by edges

Ex: path A→B: (A,B), (A,C,D,A,B)

fewest # edges needed to traverse from u to v

. $\delta(u,v)$: distance from u to v following the shortest path

* $\delta(u,v) = \infty$ by convention if there is no path from u to v.

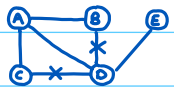
single_pair_reachability(G,s,t): true iff there's a path from s to t in G.

single_pair_shortest_path(G,s,t): return shortest path & its distance $\delta(s,t)$ for s & t in G.

→ solving this problem solves reachability issue

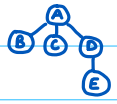
single_source_shortest_path(G,s): return shortest path tree & set of distances for all vertices starting @ s.

→ solving this solves single_pair_shortest_path



how to represent shortest path tree?

Ex: A → D



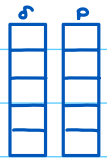
ABCDE $\Omega(|V|^2)$

A	
B	A
C	B
D	C
E	D

idea: store elt. before current

go in layers →

BFS:




```
L0 = {s}
for v ∈ Li,
    for u ∈ Adj(v)
        if δ[u] == ∞:
            Li+1 add(u)
            δ[u] = i+1
            P[u] = v
```

$\sum_{u \in V} \deg^+(u) = |V|$

cost of BFS = $O(|V|+|E|)$ ← runtime

proof of correctness: induction (layers)

*dive in

DFS: single_source_reachability(G,s):

```
def visit(P,v):
    for u ∈ Adj(v):
        if u ∉ P:
            P[u] = v
            visit(P,u)
```

P = hash table

A	/
B	A
C	D
D	B
E	D

RUNNING TIME: $O(|E|)$

* improved R.T. @ sacrafice of shortest path

* can see if vertex is in graph in $O(1)$.

$T(n) = O(n^2)$

$T'(n) = O(n^2)$

WRONG!

$T(n) - T'(n) = O(n^2) - O(n^2) = 0$ ✗

counter:

$T(n) = n^2$

$T'(n) = 7 \in O(n)$ → $T(n) - T'(n) = n^2 - 7 \neq 0$

don't subtract → usually add!

10/14 LAST TIME:

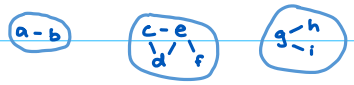
- BFS
- DFS

] time $O(|V|+|E|)$, linear in graph size

- Single-source shortest path → solve w/ BFS
- Single-source reachability → solve w/ DFS

(undirected graphs)

CONNECTED COMPONENTS: two vertices u, v are in the same CC. if there is u-v path in graph



← EX: 3

how to find connected components? input: $G=(V,E)$, output: CCs

$S \leftarrow \{\}$ set holding CCs

TOTAL RUNTIME: linear (DFS/BFS only touch each node only once)

for each $v \in V$ not already in a CC in S: ↖ $O(|V|)$

sloppy: $(O(|V|+|E|)) \cdot |V|$

run $C \leftarrow$ SSReachability(G, v) ← ex: a only returns b bic only reachable node

← DFS/BFS $O(|V|+|E|)$

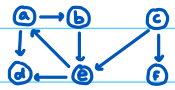
add C to set S

return S

time: $O(|V|) + \sum_{\text{comp } C} \left(\begin{matrix} O(\#\text{vertices in } C: \\ +\ \#\text{edges in } C) \end{matrix} \right)$

$= O(|V|) + O(|V|) + O(|E|) = O(|V|+|E|)$

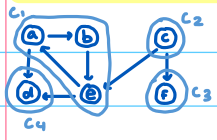
DIRECTED GRAPH:



(for directed graphs)

STRONGLY CONNECTED COMPONENTS (SCC): two vertices u,v are in the same SCC if

∃ u-v path AND v-u path in the (directed) graph

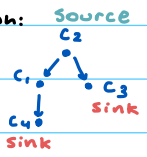


*as soon as edges have direction, be careful of single vs. bidirectional

CONDENSATION GRAPH of directed graph: ↳ G_{con}

V = SCCs of $G = \{C_1, C_2, C_3, C_4\}$

E = there is an edge $(C_i, C_j) \in G_{con}$ if ∃ an edge from vertex in C_i to vertex in G



*draw edges going b/t CCs → careful of direction!

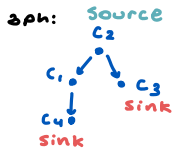
*nice way to summerize structure in directed graphs

(directed acyclic graph)

CLAIM: for every directed graph G, its condensation graph G_{con} is acyclic (no cycle) - DAG

PF BY CONTRADICTION: assume ∃ cycle in G_{con}: $C_{i_1} \to C_{i_2} \to \dots \to C_{im}$

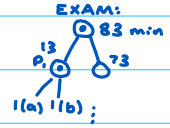
all vertices in $C_{i_1}, \dots, C_{im}$ are in the same SCC ⇒ C_{i_1}, C_{i_2} are in same SCC. contradiction.



IDEA:
- ∃ sink C_{sink} in G_{con}
- BFS/DFS from vertex in C_{sink} finds entire C_{sink}.
  - Q_1: how to find vertex in C_{sink}? found C_{source}, but want C_{sink}. reverse arrows and see which finishing time is longer.
  - Q_2: what's next?

* Finishing time of root always > subprob.

EXAM:



$O(|V|+|E|)$

FULL DFS: run DFS from every unexplored vertex in G until all are explored

FINISHING TIME: of a vertex v in Full DFS run is time at which run has explore v and all its neighbors

KOSARAJV-SHAVIR: finding SCCs

CLAIM: if C & C' are SCCs and ∃ edge $C \to C'$ then if run full DFS from any vertex, largest finishing time in C is bigger than largest finishing time in C'

PROOF: two cases

1) C explored first

2) C' explored first

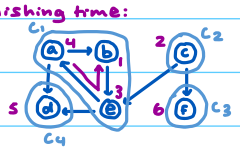
FINDING ALL VERTICES IN C_{SOURCE}:

FINDSOURCE(G):
- run full-DFS on G, recording finishing times
- return vertex w/ biggest finishing time.

FIND ALL OF C_{SINK}(G):
- reverse all edges in G to get G^{rev}
- v ← FINDSOURCE(G^{rev})
- return SSReachability(G, v)

FINDSCC(G): $O(|V|+|E|)$
- let G^{rev} be G w/ all edges reversed
- run full DFS on G^{rev} recording finishing times f
- run full DFS on G w/ vertices ordered by reverse finish time f (highest → lowest)
  - each time need a new tree, mark as new SCC

Finishing time:



10/16

TOPO SORT: (must be on acyclic, directed graphs)

TOPO ORDER on $G=(V,E)$ is order σ ∀ edge (u,v) $\sigma(u) < \sigma(v)$

DFS, reverse finish order is topo order

C

REDUCTION: algorithm for transforming one problem to another $A \leq B$

Ex: squaring mult. $a^2 = a \cdot a$ (Turing & Coke Reduction)

Ex: mult. to squaring $(a+b)^2 = a^2 + 2ab + b^2$

$$a \times b = \frac{(a+b)^2 - a^2 - b^2}{2}$$ (Turing & Coke)

Ex: reachability to SSSP: see if $\delta(s,v)$ is finite (Turing & Coke)

TURING REDUCTION: $A \leq_T B$ iff $\exists$ A that solves A while using B as atomic subroutine

- "oracle subrout" bc soln to B is sometimes called oracle

COOK REDUCTION: polynomial time Turing Reduction (polynomial upper bound)

THM: $A \leq_T B$ in time $f(n)$, B solveable in time $g(n)$

$f(n) \cdot g(n + f(n))$

MANY-ONE REDUCTION: $A \leq_m B$
fn f alg.

EX: square to math

$a \longmapsto (a, a)$ square(a) = mult(a, a)

take input x for A
transform to y=f(x)
B(y) should be same
answer as A(x)

if reduction is poly time, called **Karp Reduction**

REDUCTIONS & GRAPHS:

GRAPH DUPLICATION:

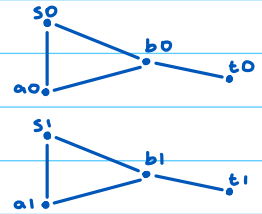
PROBLEM: given $G=(V, E)$, two vertices $s, t \in V$

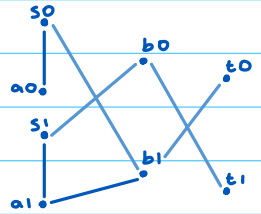
OUTPUT: shortest even-length s-t path

$V' = V \times \{0, 1\} = \{(v, b) : v \in V, b = 0 \text{ or } 1\}$

$E' = \{((u, b), (v, 1-b)) : (u, v) \in E, b \in \{0, 1\}\}$

$G' = (V', E')$





$(G, st) \mapsto (G', s_0, t_0)$ reduces SPSP to SPSP

runtime: $|V'| = 2|V|$ linear time to compute G'
$|E'| = 2|E|$ $O(2|V| + 2|E|) = O(|V| + |E|)$ linear

SELF-REDUCTION

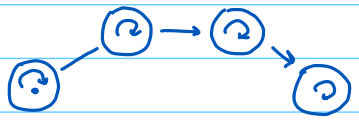
OUTPUT FORM:

v1: output $d(s, -)$

v2: output P shortest path tree

$v1 \leq v2$ compute depth with pre-order traversal

$v2 \leq v1$ for each vertex v, P(v) is in-neighbor w/ $d(s, v) = 1 + d(s, P(v))$



INPUT RESTRICTION: G is semi-connected iff $\forall u, v$ pairs of vertices, there is either u-v or v-u path

problem: given G, is G semiconnected?

problem 2: $E \subseteq <$ see if all edges $(v, v+1)$ exist ✓

problem 3: G acylic reduce to P2 with DFS

10/21

PRIORITY QUEUES

	implementation: AVL	HEAP:	sequence:
build(A)	O(nlogn)		O(n)
insert(x)	O(logn)	O(logn)	
delete_max()	O(logn)	O(logn)	
delete_min()	O(logn)		

EITHER OR! not both.

if multiple, deletes one of them

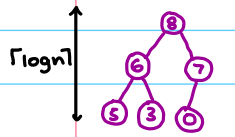
```
def pq_sort(A):                          O(nlogn) sorting. NOT in place (have to build tree)
    B = []                               O(1)
    q = build(A)                         O(nlogn)
    for i = 0... |A|:                                  } O(nlogn)
        B.prepend(q.delete_max())        O(logn)      }
    return B
```

AVL:

item	
height	
l	r

binary
COMPACT TREE: all nodes pushed as far left as can be

→ has 1:1 correspondence w/ array implementation



ARRAY REPRESENTATION:

∞	8	6	7	5	3	0
	1	2	3	4	5	6

(by level)

→ const. overhead for maintaining n nodes *useful for heaps

find children / parent:

LEFT(x) = 2x x = parent

RIGHT(x) = 2x+1 left child = 2x

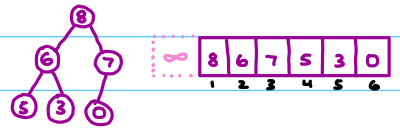
right child = 2x+1

PARENT(x) = $\lfloor \frac{x}{2} \rfloor$

MAX DEPTH: ⌈log n⌉

MAX HEAP PROPERTY: for x ∈ Tree, x ≥ left(x), x ≥ right(x)

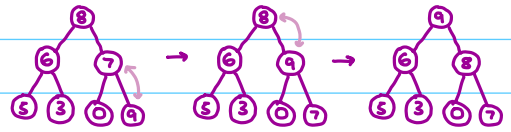
- don't know relationship b/t L/R children, but know that parent is biggest
- MIN HEAP PROP: opposite (parent = min elt.)



∞	8	6	7	5	3	0
	1	2	3	4	5	6

```
def heapify_up(x):
    p = x/2
    if A[x] > A[p]:                      } Θ(logn)
        swap A[x] with A[p]
        heapify_up(p)
```

insert(9):



∞	9	6	8	5	3	0	7
	1	2	3	4	5	6	7

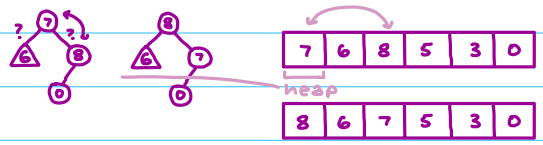
delete_max:

swap root w/ last node

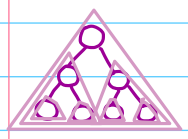
heapify_down(1) ← start from root

return removed value

heapify-down: compare w/ children



case 1: $A[x] \geq A[L(x)]$
$\geq A[R(x)]$
case 2: $A[x] < A[L(x)]$
$\geq A[R(x)]$
swap w/ $L(x)$
case 3: both children $> A[x]$
swap w/ larger child



go backwards through A
heapify-down along the way
$O(n \log n)$

at every level, # nodes = $\frac{n}{2^{h+1}}$

converges on a constant

$$\text{work} = \sum_{h=0}^{\log n} \frac{n}{2^{h+1}} ch = nc \sum_{h=0}^{\log n} \frac{h}{2^{h+1}} < nc \sum_{h=0}^{\log n} \frac{h}{2^h} < nc \sum_{h=0}^{\infty} \frac{h}{2^h} = nc \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} \to \text{work} \in \Theta(n)$$

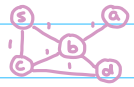
($\frac{1}{2} \in \Theta(1)$)

```
def heap_sort(A):
    build(A)                Θ(n)
    for i = 1 ... |A|:
        delete_max()        O(log n)      } Θ(n log n)
```

decr. values →

10/23 **SHORTEST PATHS: DIJKSTRA**

weighted graphs, dijkstra, analysis



SSSP: single source shortest paths

reduction/duplication

- on undirected graph w/ equal weights: BFS $O(V+E)$
- on undirected graph w/ non-equal weights (small positive ints): graph $O(W(V+E))$
  - W: max weight of graph
  - if w is ugly #, have to go through & compare. ex: 1/2, 2.12, π, e

for now, positive non-zero #s.

$w(u,v)$: weight of edge b/t u & v

$\delta(u,v)$: shortest path weight b/t u & v

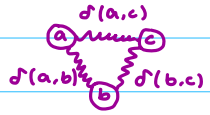
$\delta(s,v) = \delta(v)$

$d(v)$: distance estimate of shortest path b/t u & v in $\forall$

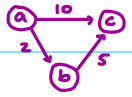
$d(s,v) \equiv d(v)$

$d(s) = 0$
$d(v) = \infty \leftarrow v \neq s$

TRIANGLE INEQUALITY: *uuu paths*



$\delta(a,c) \leq \delta(a,b) + \delta(b,c)$ for paths

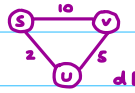


$w(a,c) > w(a,b) + w(b,c)$

RELAXATION:

$d[u] > d[s] + w(s,u)$ set $d[u] = 2$

a[s]: 0
relax an edge



$d[v] > d[s] + w(s,v)$ set $d[v] = 10$

$d[v] > d[u] + w(u,v)$ set $d[v] = 7$

☆ **relax(u, v):**

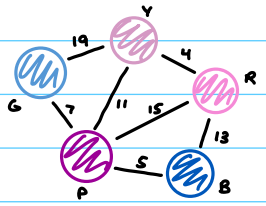
set $d[v] = \min(d(v), d(u) + w(u,v))$

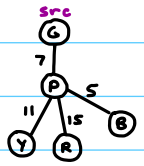
if $d[v]$ changed, set $parent[v] = u$

* relaxation is safe

· what order to relax in?

order in terms of δ: from G to others





G P Y B R

DIJKSTRA:

· relax edges from each vertex in incrementing order of distance from source

· will use priority queue Q on vertices (unique IDs) s, u, v, etc. and keys correspond to d[v]'s

ALGO:

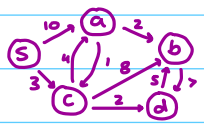
· set $d[s] = 0$, $d[v] = \infty$ for $v \in S$ in V

· build priority queue Q for item $(v, d(v))$, $v \in V$

while Q not empty

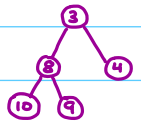
· delete $(v, d(v))$ from Q, with min $d[v]$

· for $v \in Adj^+(u)$: relax (u,v)



delete v from Q	s	a	b	c	d
s	0	∞	∞	∞	∞
c		10		3	
d		7	11		5
a		7	10		
b			9		

$\delta(c) = 3$
$\delta(d) = 5$



decrease_key(i, new_key):

- change val & heapify_up
- log(n) time heapify up

CORRECTNESS: @ end of Dijkstra, $d(v) = \delta(v)$ for all $v \in V$

· if relaxation sets $d(v) = \delta(v)$, then $d(v) = \delta(v)$ on termination

· relaxation is safe & only decreases $d(v)$

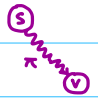
show $d(v) = \delta(v)$ when v is removed from Q

STRONG IND of number k vertices removed from Q.

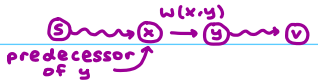
BASE CASE: s $d(s) = 0 = \delta(s)$

IND. STEP: we are removing v from Q and all vertices u removed from Q,

assume $d(u) = \delta(u)$



let π be a shortest path $w(\pi) = \delta(v)$



y is 1st vertex in π that is still in Q.

x has been removed

I.H. $d(x) = \delta(x)$

(x,y) was relaxed when x was removed $\Rightarrow d(y) \leq \delta(x) + w(x,y)$

subpaths of shortest path π are shortest paths $\Rightarrow d(y) = \delta(y)$

$\delta(y) \leq \delta(v)$ non-negative edge weights

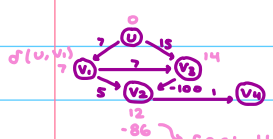
$\leq d(v)$ relaxation is safe

$\leq d(y)$ v is vertex with min $d(v)$ on Q

$d(v) = \delta(v) \; (= d(y))$

10128 **BELLMAN FORD**

Dijsktra fails for negative edge weights

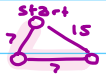


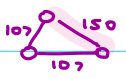
(u,v_1) (u,v_2)	(v_1,v_2) (v_1,v_3)	(v_2,v_4)	v_4 no out edge	(v_3,v_2) -100

$\delta(u,v_1)$

fact that it's < 12 is problematic. we thought we had the shortest path at that point.

→ if we run Dijsktra with +100 to every edge, discourages shortest path.



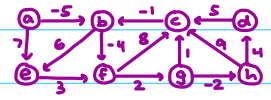


NEGATIVE WEIGHT EDGES:

$\delta(a,b) = -\infty$

$\delta(a,f) = -\infty$

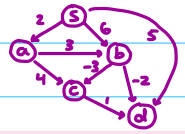
path is under



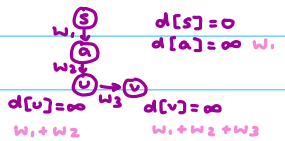
NEGATIVE WEIGHT CYCLE: b f g c b → -2 weight cycle

DAG SHORTEST PATHS: negative-weight edges

· no cycles, ∴ no negative cycles



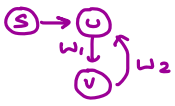
topological sort: s a b c d



if different order:
→ 1st: relax a→v
· order of relaxation matters.

CLAIM: DAG topological sort computes shortest path even w/ negative weights

$|v|$ vertices



$w_1 + w_2 > 0$

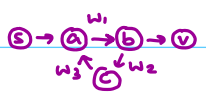
longest simple path len (# edges) $\leq |V| - 1$ = simple

SIMPLE SHORTEST PATHS: **SIMPLE**: no vertex is repeated

no neg. cycles

CLAIM: if $\delta(s,v)$ is finite, there exists a shortest path to v that is simple.

could : have path w/ positive weight cycle that is shortest path?



SP: a b c a b v $\quad w_1 + w_2 + w_3 > 0$

K-EDGE DISTANCE $\delta_k(s,v)$

is minimum weight path from s to v with $\leq k$ edges

compute $\delta_{|V-1|}(s,v)$, $\delta_{|V|}(s,v)$

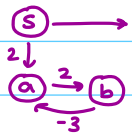
CLAIM 1: no negative cycles $\delta(s,v) = \delta_{|V|-1}(s,v)$

NEGATIVE CYCLE WITNESS:

if $\delta_{|V|}(s,v) < \delta_{|V-1|}(s,v)$ shortest non-simple path to v, so $\delta(s,v) = -\infty$

call v a negative cycle witness

CLAIM 2: every negative weight cycle C reachable from S contains a witness.



$|V| = 100$ $\quad \delta_{100}(s,a)$ $\quad \delta_{99}(s,a)$

$\delta_{100}(s,b)$ $\quad \delta_{99}(s,b)$

$\delta_1(s,a) = 2$

$\delta_2(s,a) = 2$ $\quad \delta_4(s,a) = 1$

$\delta_3(s,a) = 1$ $\quad \delta_5(s,a) = 0$

$\delta_1(s,b) = \infty$ $\quad \delta_2(s,b) = 4$ $\quad \delta_3(s,b) = 4$ $\quad \delta_4(s,b) = 3$

Proof of CLAIM 2: for $v \in C$, let v's predecessor in C

· negative-wt cycle C reachable from S

$$\sum_{v \in C} \delta_{|V|}(s,v) \leq \sum_{v \in C} \delta_{|V|-1}(s,v) + w'(v',v) \quad \leq |V| \text{ edges}$$

$$\leq \sum_{v \in C} \delta_{|V|-1}(s,v) + \sum_{v \in C} w(v',v) < 0$$

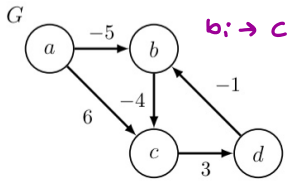
$$\Rightarrow \sum_{v \in C} \delta_{|V|}(s,v) < \sum_{v \in C} \delta_{|V|-1}(s,v)$$

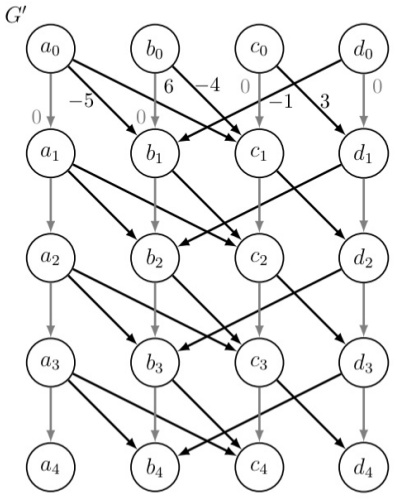
BELLMAN-FORD:

Example

$a_i \to b_{i+1}$

$b_i \to c_{i+1}$





$\delta(a_0, v_k)$ — δ_1 δ_2

$k \setminus v$	a	b	c	d
0	0	∞	∞	∞
1	0	**−5**	**6**	∞
2	0	−5	**−9**	**9**
3	0	−5	−9	**−6**
4	0	**−7**	−9	−6
$\delta(a, v)$	0	$-\infty$	$-\infty$	$-\infty$

initial relax ↓

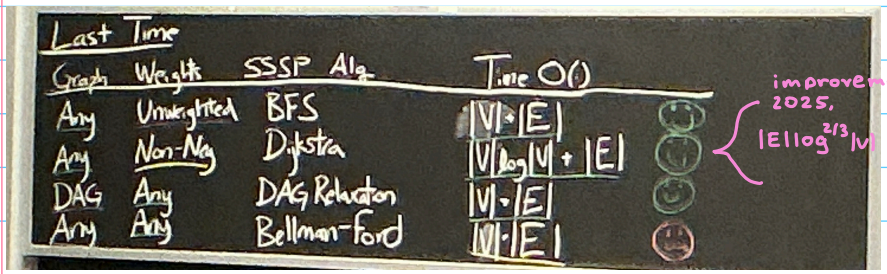
Note that b is a witness, and c and d are not. The negative-weight cycle is bcd. Claim 2 simply says that there always exists a witness in the negative-weight cycle. Consider the case where there is an edge from a to a different vertex e, meaning that $|V| = 5$. In this case, $\delta_5(a, b) = -7$, so b is not a witness in this new graph, but $\delta_5(a, c) = -11$, and c is a witness.

- produce a duplicated DAG level by level
- Compute k edge dist. for each k

|V| passes of relaxation over all edges |E| — |V|−1 passes → S.P. wts.

$O(|V||E|)$ alg. — one more pass → witness

10/30



ALL-PAIRS SHORTEST PATHS:

- Definition
- Johnsons Alg
- All-pairs reliability

THE PROBLEM: APSP

INPUT: directed graph $G=(V,E)$ weights $w: E \to \mathbb{Z}$

OUTPUT: $\delta: |V|^2 \to \mathbb{Z}$ ... $\delta(u,v)$ length of shortest u-v path

OR "FAIL" if ∃ neg. weight cycle in G.

} output len $\Omega(|V|^2)$

GRAPH:	WEIGHTS:	ASAP ALG:	TIME O(·)
any	non-neg.	$|V|\cdot$ Dijkstra	$|V|^2\log|V| + |V||E|$
any	any	Johnson	$|V|^2\log|V| + |V||E|$
any	any	Floyd-Warshall	$|V|^3 / 2^{\Omega(\sqrt{\log|V|})} = |V|^{3-o(1)}$

OPEN QUESTION: ∃ alg for APSP that runs in time $O(|V|)^{2.99}$?

why study APSP?

- useful
- equivalent to many other problem
  - finding neg.-weight triangle
  - min-weight cycle
  - replacement paths problem

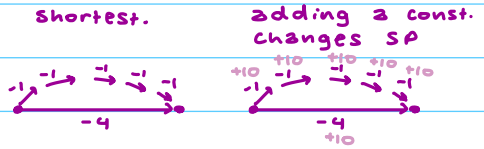
"MOST-BEAUTIFUL PATH PROBLEM"

$\{-10, \ldots, 10\}$ → use Johnsons.

JOHNSON'S ALG:

PLAN:

1) find negative cycle, if exists. FAILS if so.
   - Bellmon Ford, $|V| \cdot |E|$
2) make all edge weights non-negative ← ★
   - reweighting must preserve shortest paths
3) use Dijkstra from every vertex... clean up

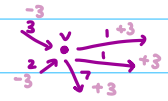


a REWEIGHTING that preserves shortest paths:

let $\phi: V \to \mathbb{Z}$

key idea: for every $v \in V$: add $\phi(v)$ to all outgoing edges from v

Subtract $\phi(v)$ from all incoming edges from v



CLAIM: this transformation changes weight of all uv paths by $\phi(u) - \phi(v)$

PF: weight of $\pi = (v_0, v_1, \ldots, v_k)$ is $\sum_{i=1}^{k} w(v_{i-1}, v_i) = \sum_{i=1}^{k} \left[w(v_{i-1}, v_i) + \phi(v_{i-1}) - \phi(v_i) \right]$

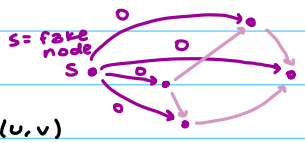
$$= \left(\sum w(v_{i-1}, v_i) \right) + \phi(v_0) - \phi(v_k)$$

COMPUTING THE REWEIGHTING:

$\exists \phi(\cdot)$ s.t. every weight ≥ 0 for all $(u,v) \in E$:

let $\delta_s(v)$ be weight of shortest s-v path $0 \leq \phi(u) - \phi(v) + w(u,v)$

define $\phi(v) = \delta_s(v)$

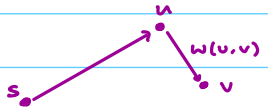


CLAIM: after reweighting by $\phi(v) = \delta_s(v)$, all edge weight ≥ 0

PF: by triangle inequality, $\delta_s(v) \leq \delta_s(u) + w(u,v)$

$\phi(v) \leq \phi(u) + w(u,v)$

$0 \leq \phi(u) - \phi(v) + w(\phi, v)$



JOHNSON'S ALG: $G=(V,E)$ $w: E \to \mathbb{Z}$ RT:

- add vertex s, connected to all $v \in V$ w/ weight zero edge $|V| \cdot |E|$
- use Bellmon Ford to look for negative cycles. FAIL if found.
- compute $\delta_s(v)$ from Bellmon Ford. reweight $(u,v) \mapsto w(u,v) + \phi(u) - \phi(v)$ $|E|$
- use Dijkstras ABSP to find $\delta'(u,v)$ $\forall u,v \in V$ $(|V|\log|V| + |E|)|V|$
- $\forall u,v \in V$ $\delta(u,v) = \delta'(u,v) - \phi(u) + \phi(v)$ ← search every pair $|V|^2$

ALL PAIRS REACHABILITY: → could solve w/ BFS from every vertex

INPUT: directed graph G unweighted $|V| \cdot (|V| + |E|) = |V|^2 + |V||E| \sim |V|^3$

OUTPUT: $\forall u,v \in V$ "reachable" or not will show alg $|V|^w = |V|^{2.372}$

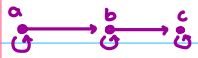
multiply n by n matrices:

naive: $O(n^3)$

clever: $O(n^w) = O(n^{2.372\ldots})$

ALG (Munro)

- let A be adjacency matrix of G w/ 1s on diagonal
- use repeated squaring to compute $A, A^2, A^4, A^8, \ldots, A^{|V|}$ $\log(|V|)$
- $\exists$ (u,v) path in G iff $A^{|V|}_{u,v} > 0$



$$A = \begin{array}{c} \\ a \\ b \\ c \end{array} \begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$ $O(|V|^2)$ space & time

$$A^2 = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$ → tells you how many paths there are from vertex to vertex

RUNTIME:

squaring matrix mult

$\log|V| + |V|^2 \longrightarrow O(|V|^w - \log|V|)$ w = how fast we can do matrix mult.

CLAIM: for all $\ell \geq 1$, $A^{\ell}_{u,v}$ counts # of paths from $u \to v$ length ℓ.

PF: by induction on ℓ:

BC $\ell = 1$: by construction

IND STEP: assume for $\ell-1$. the # of length ℓ $u \to v$ paths is:

$$\sum_{w \in |V|} (\#\text{ of length } (\ell-1) \text{ paths } u \to w) \begin{pmatrix} 1 \text{ if } (w,v) \in E \\ 0 \text{ otherwise} \end{pmatrix}$$

$$= \sum_{w \in |V|} A^{\ell-1}_{u,w} \cdot A_{w,v} = A^{\ell}_{u,v} \quad \blacksquare$$ matrix mult.

TEST 2

11/4 **GREEDY ALGORITHMS**

ACTIVITY SCHEDULING: $A = \{[s, f)\}$

goal: create set S containing activities from A s.t. no overlap

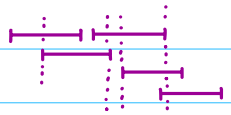
and S is maximal size. (most # of activities)

input: $\{[1,8), [2,4), [3,6), [4,7), [7,8)\}$

output: $\{[2,4), [4,7), [7,8)\}$

let A' = A without S, or any other activities that overlap with it.

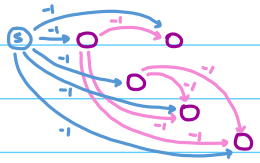
$S \times \{S_1\}$ is optimal in A'



- turn every activity into vertex:
- add an edge from every activity a to every activity that starts after a finishes
- add supernode connecting to all tasks.

cycles impossible!

- DAG SP on weights -1. (finding max path)
- find SP ("longest", most negative path)



- covers most edges

$\Omega(n^2)$ → we can do better!

GREEDY CHOICE PROPERTY approach:

let $a \in A$ be the activity with earliest finish time.

$\exists S$, optimal solution that contains a.

PROOF: let S be optimal $a \notin S$.

$S = \{s_1, s_2, \ldots, s_k\}$

Swap a for s_1 to get S'

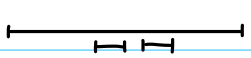
$S' = \{a, s_2, s_3, \ldots, s_k\}$

$|S'| = |S|$ ∴ must be an optimal solution. a = greedy choice

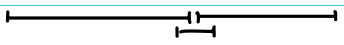
GREEDY CHOICE PROPERTY: there's at least one optimal solution that contains greedy choice

counterex:
interval w/ earliest start time:

[1,5), [2,3), [3,4)



shortest interval counterex:



another greedy choice:

latest start time

GREEDY ALGORITHM:

1) sort by finish time
2) make greedy choice
3) walk the list until non-overlapping activity and add it
4) go to 2

PF: claim: greedy alg produces optimal solution.

let S be an optimal solution. S starts with greedy choice g.

let GREEDY be soln produced by Greedy algorithm.

let A' be activities in A that don't overlap w/ g.

GREEDY $\setminus \{g\}$ is optimal for A'

$|GREEDY| - 1 \geq |S| - 1$

$\therefore |GREEDY| \geq |S|$

* use induction
* reduce greedy to a subproblem of itself
* can show that its a subproblem

ACTIVITY SCHEDULING

INPUT: $A = \{(t, d)\}$

OUTPUT: schedule, S, that minimizes the max late time.

$\ell(a)$ = how late, after due time, a finishes. min = 0

$\ell(A) = \max(\ell(a_i) \; \forall a_i \in A)$

$A = \{(1,1)_{a_1}, (2,2)_{a_2}, (3,3)_{a_3}\}$

S_1: t=0, a_1 (1), a_2 (3), a_3 (6)

a_1 ends @ 1. supposed to end there. a_3 supposed to end @ 3, but ends @ 6.

$\ell(S_1) = 3$ ← latest (6-3)

S_2: t=0, a_3 (3), a_2 (5), a_1 (6)

$\ell(S_2) = 5$

observation 1: there exists an optimal schedule with no idle time.

observation 2: let S be optimal for A. let a_1 be scheduled first in S.

if remove a_1, then $S \setminus \{a_1\}$ is optimal for $A \setminus \{a_1\}$

observation 3 GCP: ∃ optimal schedule where the 1st activity due is scheduled first.

PF 3: let S be optimal schedule where a, activity with earliest due date, is not first

let b directly precede a in S.

let S' = S but with a, b swapped

S: [| b | a |]

S': [| a | b |]

for a, $\ell(a)$ in S' improved vs $\ell(a)$ in S.

$\ell(b)$ might have gotten worse.

$\ell(b)$ in S' $\leq \ell(a)$ in S.

GREEDY ALG:

1) sort by due date $O(n \log n)$

2) make greedy choice

3) go to 2

CLAIM: greedy alg produces optimal strategy, GREEDY.

PF: let S be optimal in A. S begins with g_1

let t_1 be duration of g_1.

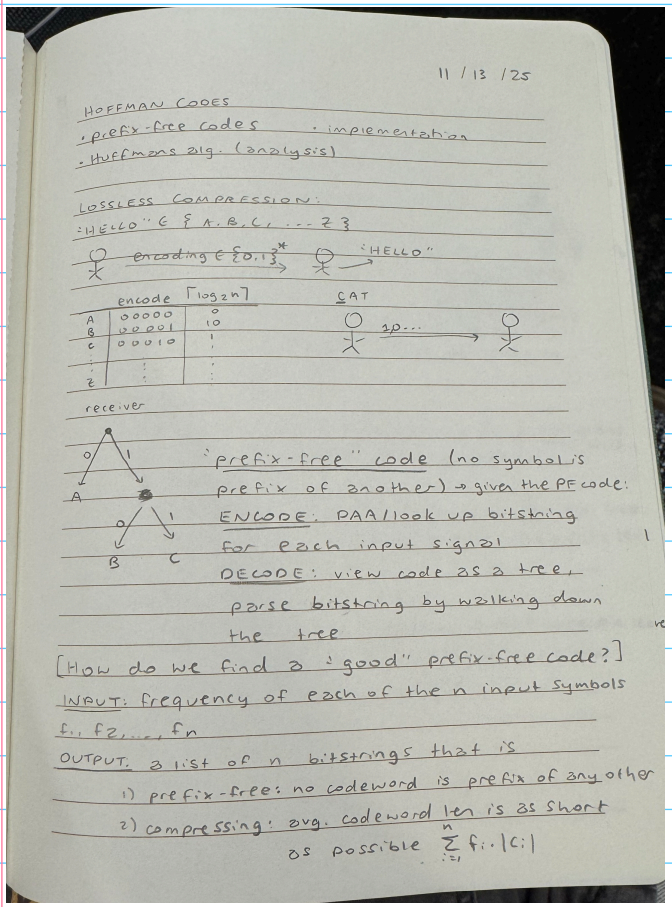
GREEDY $\setminus g_1$ is optimal in $A \setminus g_1$.

$\ell(\text{GREEDY} \setminus g_1) \leq \ell(S \setminus g_1)$ in $A \setminus g_1$

adding g_1 adds lateness of t_1 to all activities evenly

$\therefore \ell(\text{GREEDY}) \leq \ell(S)$

11/13



Ex: A 00 ✓
B 010
C 10
D 1011] ✗

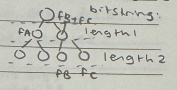
[TREE VIEW OF PREFIX-FREE CODE:]

$f_A = 5$, $f_B = 1$, $f_C = 2$

- weights of nodes = sum of children

define cost(T) := $\sum_{\text{node } x} f_x$

claim: cost(T) = $\sum_{i=1}^{n} f_i \cdot |c_i|$



- each value f_i in the tree is counted $|c_i|$ times - once per level in tree until hit leaf i.

goal: alg for

INPUT: frequencies $f_1, f_2, f_3, \ldots f_n$] Huffmans fits this

OUT: prefix-code tree of min cost]

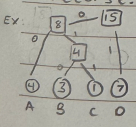
[Observation 1: any optimal tree is a full bin. tree

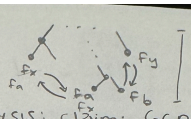
observation 2: in optimal tree, $f_i \le f_j$, then $|c_i| \le |c_j|$

ALGORITHM: input: $f_1, f_2, \ldots, f_n$

- start by giving each frequency a leaf node → n leaves
- find "smallest" nodes f_a, f_b & replace them w/ new internal node w/ weight $f_a + f_b$
- recurse.

Ex:





ANALYSIS: claim: GCP: there is an optimal tree that has 2 smallest freqs. as siblings

PF: start w/ optimal tree, transform it to one w/ GCP. let f_x, f_y be two smallest freqs $f_x \le f_y$ let f_x be freq. deepest in tree, f_b be sibling by O2, can swap f_x, f_a

know $f_y \le f_b$, f_y is 2nd smallest ⇒ By O2, swap $f_y \leftrightarrow f_b$

CLAIM: Huffman's alg. outputs min cost tree

PF: by induction on n, # of frequencies

BC: n=2 →  ✓

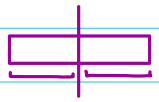
IND. STEP: let H = tree that HA (huffmans alg) outputs

- assume that Huffmans is optimal for n-1 freq/symbols.
- let f_x, f_y by smallest freqs
  - in H, f_x & f_y are siblings
  - by GCP, ∃ an optimal tree T w/ f_x, f_y siblings] GOAL: cost(H) ≤ cost(T)
- let T' & H' are T, H but w/ leaves f_x, f_y removed
- imagine running HA on freq table of n-1 freqs w/ new symbol z replacing x, y s.t. $f_z = f_x + f_y$
  - outputs exactly H'. let T' be any other tree on n-1 symbols. by ind. hyp. cost(H') ≤ cost(T')
  - by defn of cost of tree: cost(H) = cost(H') + f_x + f_y

cost(T) = cost(T') + f_x + f_y

cost(H') ≤ cost(T'), cost(H) ≤ cost(T) □

11/18 **DYNAMIC PROGRAMMING**



- generalizes recursion
- divide & conquer, except overlapping problems

COIN ROW PROBLEM:

ARRAY A of size n: represents a row of coins w/ positive values

GOAL: pick max amount subject to: no two adjacent coins can be picked.

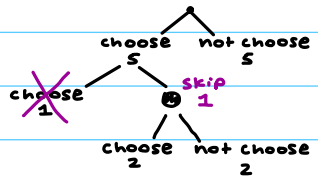
$A = [C_0, C_1, \ldots, C_{n-1}]$

A: 5 1 2 10 6 2 → max = 17

RECURSIVE / EXHAUSTIVE SEARCH:

1) decide to not pick C_0.
   use recursion to find optimal solution $[C_1, \ldots, C_{n-1}]$

2) decide to pick C_0. cannot choose C_1
   use recursion on $[C_2, \ldots, C_{n-1}]$



IMPLEMENTATION

Coins(A):

Base Case: if |A| < 2: output Σ A

Else output max(Coins(A[1:]), A[0] + Coins(A[2:]))
start @ 1 (Coins(A[1:])) — start @ 0 (A[0] + Coins(A[2:]))
overlap

#cells is exponential in n

$T(n) = T(n-1) + T(n-2) + \Theta(1)$

$T(n) = \Omega(2^{n/2})$ ∴ ← slow — reduce the work! (there's many of the same branch that we are redoing)

MEMOIZATION

```
def Coins(A):  linear time!
    memo = [None] * len(A)          *only going to solve each suffix (end part) problem ONCE
    def Coin_Helper(i):
        if i ≥ len(A): return 0
        if memo[i] == None:
            without_i = coins_helper(i+1)
            with_i = A[i] + coins_helper(i+2)
            memo[i] = max(with_i, without_i)
        return memo[i]
    return coin_helper[0]
```

PURE DP SOLUTION O(n) runtime

memo[i] depends on memo[i+1] & memo[i+2]

```
def Coins(A):
    memo = [0] * (len(A)+2)
    for i in range(len(A)-1, -1, -1):
        memo[i] = max(memo[i+1], A[i] + memo[i+2])
    return memo[0]
```

SRT BOT:

S: subproblems

R: relationship b/t subproblems

T: topsort (must be acyclic) - dependence b/t subproblems = DAG

B: identify base case

O: show how output depends on entries of M.

T: analyze runtime.

Coin Row - SRT BOT:

S: M(i) = max value of coins we can pick from A[i:] i: 0 to n-1

R: M[i] = max(...) i = 0, n-1

T: each M[i] depends on M[i'], i' > i

B: M(i) = 0, i ≥ n

O: output is M(0)

T: n values M(i), each takes O(1), ∴ O(n) runtime

memo[5] = 2
memo[4] = 6
memo[3] = 12
⋮
memo[0] = 17

Sol = [] report coin choices for M[0]

```
while i < n:
    if memo[i] == memo[i+1]:
        i += 1   ← i skipped
    else:
        Sol.append(A[i])
        i += 2
return Sol   ← i chosen
```

exhaustive search:

* DP does exhaustive search, but doesn't have to be exponential RT

n distinct subproblems

DAG:

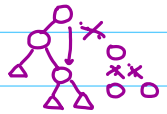
BOWLING PINS B: [1, -2, -4, 12, -3, 10] (23)

drop a pin, earn point value

drop 2 consecutive pins, product of values maximize score

1st step: choices (what they are

11/20



LONGEST COMMON SUBSEQUENCE:

input: A, B strings — A: GODZILLA

out: m, len of LCS — B: GHIDORAH

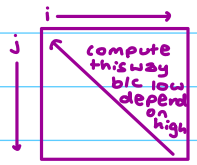
GDA
GIA
] out m=3

if 1st char same → if A[0] == B[0]: 1+LCS(A[1:], B[1:])

always LCS(A[1:], B)

always LCS(A, B[1:])

] return max of these 3 → this is exponential. we want to come up w/ memo so we don't recompute



if M(i,j) is None:
M(i,j) = max of
{ help(i+1, j+1)+1 if A[i] == B[j]
{ help(i+1, j)
{ help(i, j+1)
return M(i,j) RUNTIME: O(|A|·|B|) quadratic!

S: M(i,j) is length of LCS of A[i:], B[j:] where $0 \le i \le |A|$, $0 \le j \le |B|$

R: $M(i,j) = \max \begin{cases} 1+M(i+1,j+1) \text{ if } A[i]==B[j] \\ M(i+1,j) \\ M(i,j+1) \end{cases}$

T: increasing i+j (never read uninitialized indices)

B: 0, where j > |B| or i > |A|

O: M(0,0)

* can reconstruct LCS by walking diagonally down the table

T: O(|A|·|B|) ← size of table

OSSP: correctness:

1: A[1:], B[1:] if we have optimal LCS, then 1+LCS(A[1:], B[1:]) is optimal w/ A[0]==B[0]

2: A[1:], B: if we have optimal LCS, then optimal among solns that didn't pick A[0]

3: A, B[1:]: if we have optimal LCS, then optimal among solns that didn't pick B[0]

LONGEST INCREASING SUBSEQUENCE:

input: array A of integers — EX: A=[8,6,7,5,3,0,9] — * for each val, choose or not

out: length LIS — out: 6,7,9 → 3 (len) — PROPOGATE w!

LIS(A, w(=-∞)); (w: default)

if A is empty: return 0

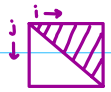
return max { 1+LIS(A[1:], A[0]) if A[0] > w
{ LIS(A[i:], w)

indexify A → i

indexify w → j (define A[-1] = -∞)

S: $M(i,j)$: length of LIS of $A[i:]$ w/ all numbers $> A[j]$

R: $M(i,j) = \max \begin{cases} 1 + M(i+1, i) \\ M(i+1, j) \end{cases}$



* don't care about w itself.

T: increasing i (start from right end)

B: $M(|A|, j)$ $\forall j$
↳ $=0$

O: $M(0, -1)$

T: $O(|A|^2)$ * have space & time optimizations

S: $M(i)$, len of LIS of $A[i:]$ that begins w/ $A[i]$, $0 \le i < |A|$

R: $M(i) = 1 + \max \{ M(j)$ where $j > i$ and $A[j] > A[i] \}$

T: increasing i (only looking @ suffixes)

B: no base case! (there is no $j > i$)

O: max in M

set $A[k] \to M[k]$
What if we used a range query (implement using AVL BST)
augment w/ max

T: $O(|A|^2)$

T: implement M using AVL BST augmented w/ max.

S maps $A[k] \to M[k]$ in $O(\log|A|)$ time

walk back along input: $|A|$ steps

$O(|A| \log |A|)$

11/25 **DYNAMIC PROGRAMMING III:**

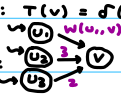
Shortest paths using DP, DP using SP algs

Subpaths of SPs are SPs

"cut and paste" arg 

DAG Single-Source SP:

S: $T(v) = \delta(s,v)$ for all $v \in V$

R: 

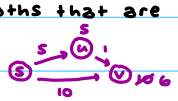
$T(v) = \min\{T(u) + w(u,v) \text{ for } u \in Adj^{-}(v)\}$

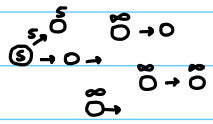
T: topological sort/order

SSSP for General Graphs: Bellmon Ford

→ need to handle cycles & negative cycles

compute shortest paths that are restricted

to a single edge 



≤ k-edge shortest paths

|V|-1 edges → longest simple path

K = |V|-1

K+1 iterations

compare Vth iteration (Find witnesses)

*remember how many cycles used so far as we build paths 

$u_0 \to v, \to u_2$

S: $T(v,k)$ = weight of SP from S to v using at most k edges for $v \in V$,

weight @ most k edges for $v \in V$, $k \in \{0, \ldots, \min(|V|, |E|+1)\}$ ↳ implies cycle

R: for every v & k, guess the last edge used in path of length k, or that it's best to use ≤ k-1 edges

$$T(v,k) = \min\begin{cases} \min\{T(u,k-1) + w(u,v)^2\} & u \in Adj^{-1}(v) \\ T(v,k-1) \end{cases}$$

T: depend on entries w/ smaller k

B: $T(s,0) = 0$, $T(v,0) = \infty$ for $v \neq s$

O: simple path can have @ most |V|-1 edges

if for some u we have $T(u,|V|) < T(u,|V|-1)$, we have witness for a neg. cycle. otherwise, $\delta(s,u) = T(u,|V|-1) = T(u,|V|)$

T: $O(|V| \cdot |E|)$

All Pairs Shortest Paths

dense graphs: $E = O(|V|^2)$

Bellmon Ford DP: $O(|V|^2|E|)$ BF: $O(|V|^4)$

Johnsons: $O(|V|^2 \log|V| + |V||E|)$ GOAL: make dense graph $O(|V|^3)$

FLOYD-WARSHALL DP

→ restrict paths to be within a subset of V. ignore negative cycles

vertices are all numbered 0 through |V|-1. 0 intermediate

$w(v,v) = 0$ for all v

$w(u,v) = \infty$ if $(u,v) \notin E$

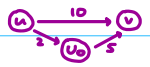
Ex: $\{0,1\} \leftarrow v_0, v_1$

Ex: u → v no intermediate vertices $\{\}$

S: $T(k,u,v)$ = min weight path from u to v through vertices in $\{0,1,\dots,k\}$

if k=0 → no intermediate vertices allowed. single edge if it exists

B: $T(0,u,v) = w(u,v)$ # or ∞ for all u & v



$T(1,u,v) = \min\{T(0\dots)$

$T(k,u,v) = \min\{T(k-1,u,k) + T(k-1,k,v),\ T(k-1,u,v)\}$

$(|V|-1)|V|\cdot|V| = O(|V|^3)$ for no neg. cycles

12/2 DP IV, Subset sum

SANDWICH CUTTING:

- V(ℓ) maps to value $\{1,2,\dots,L\}$
- sandwich of len L
- task: divide up so as to maximize total value

Ex: 1, 10, 13, 18, 20, 31, 32

ℓ 1 2 3 4 5 6 7

+2

10+10+13 = 33 31+1 = 32

what is 1st choice?

- pick where to make 1st cut * length is changing → ∴ parameterize by length
- if we cut k inches, $V(k) + SC(V, \ell-k)$ input: (L, v) size L+1 ↑represented as v(1), v(2), … , v(L)

S: $SC(\ell)$ = max value from cutting length ℓ $0 \le \ell \le L$

R: $SC(\ell) = \max_{1 \le k \le \ell} V(k) + SC(\ell-k)$

T: decreasing ℓ (depends on $\ell' < \ell$)

B: $SC(0) = 0$

O: $SC(L)$

T: $O(L^2)$ L SPs, O(L) each (quadratic in input size)

POLY TIME: bounded by polynomial in input size

PSEUDOPOLY: bounded by polynomial in input values

SUBSET SUM:

- input: $A = [a_1, a_2, \ldots, a_n]$ (positive), len L
- output: YES iff is a subset of A with sum L

EX: $A = [2, 5, 7, 8, 9]$, $L=21$? YES $(5, 7, 9)$

$L=25$? NO

DECISION PROBLEM: output is binary (T/F, Y/N, etc)

- do we include a_n? $A' = [a_1, \ldots, a_{n-1}]$ $L' = L - a_n$ o/w $L' = L$

S: $SS(i, L')$ = YES iff there is a subset of $A[1, \ldots, i]$ with sum L' $0 \le i \le n$, $0 \le L' \le L$

R: $SS(i, L') = \text{OR} \begin{cases} SS(i-1, L') \\ SS(i-1, L'-a_i) \end{cases}$ ← include only if valid

T: decreasing i

B: $SS(i, 0) =$ yes

$SS(0, L') =$ no if $L' \neq 0$

$SS(i, L') =$ no if $L' < 0$

O: $SS(n, L)$

T: $O(nL)$ SPs, $O(1)$ each, $O(nL)$ total (2^n reachable SPs)

CAN WE DO BETTER?? nobody knows!!

if so, we'd get efficient algos for all kinds of problems, e.g.

PARTITION:

- input: $[a_1, a_2, \ldots, a_n]$
- output: YES if exists a bipartition of A with equal sums

REDUCE TO SS: $A \mapsto (A, \frac{1}{2}\Sigma A)$ pseudopoly w/ SS algo.

$(A, L) \mapsto A + [L+1, \Sigma A - L + 1]$ goes into diff halves of partition

NEGATIVE SUBSET SUM: allowed neg. vals in A as well

EX: $A = [-1, 4, 7, 0, 0, 0]$ $L=3$ ← fill in remainder w/ 0's if needed

$[1, 6, 9, 2, 2, 2]$ $L'=9$

$[1M-1, 1M+4, \ldots, 1M]$ $L = 3M+3$

pseudopoly! M = poly input.

12/4 last lecture:

- pseudopolynomial time
- equiv. subset sum, neg. subset sum, partition

today:

- decision problems, complexity, classes, P vs. NP

NP-hardness Better than $O(nL)$

what about longest simple path?

computational vs. relative complexity

compare APSP vs. longest path vs. subset sum

Sorting	$O(n!)$	$O(n \log n)$
SCC	$O(n!)$	$O(n+m)$
Subset Sum	$O(2^n)$	$O(nL)$

all of these can be reduced to this

DECISION: given some input, does it have some property

SEARCH: given input, which defines a search space, output a valid element

OPTIMIZATION: given input, constraint, cost function, output a soln which minimizes/maximizes cost

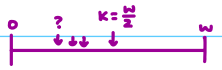
FUNCTION: given input, compute some output (math, counting)

MIN WEIGHT S-T PATH:

INPUT: graph G, nodes s-t

OUTPUT: weight of minimum s-t path

DECISION: Graph G, s-t, int k → does there exist an s-t path w/ weight ≤ k



exponential time $O(2^{poly(n)})$

COMPLEXITY CLASSES: P, NP, EXPTIME

P: easy & efficient: solveable in $O(poly(n))$

$O(n)$ ✓ $O(n^2)$ ✓ $O(n^{61210})$ ✓ $O(n^{\log\log n})$ ✗

A: most natural have small exponents. $O(n^3)$ APSP

B: Strong Church Theory Thesis: 2nd SP

SUBSET + LONGEST PATH

- easy to verify

a **VERIFIER** $V(x, c)$, x is instance, c is a witness

- if x is YES, there exists c s.t. $V(x,c) = T$
- if x is NO, for all c, $V(x,c) = F$

V must run in poly time

$|c| = poly(n)$

Ex: SUBSET SUM

C = list of numbers

verify V = sum up the list. check == L

LONGEST PATH

C: a path

V: is C a valid path? is |C| ≥ k?

SEARCH: does there exist a C s.t. V(x, c) = True

usually binary strings

P ⊆ NP

C = C

V = algo.

NP ⊆ EXPTIME

number of c's = $O(2^{poly(n)})$

loop over all C if V(x,c) = True → accept/return yes

$\underbrace{O(2^{poly(n)})}_{\text{much bigger}} \cdot \underbrace{O(poly(n))}_{V}$

P ⊆ NP ⊆ EXP. TIME

at least one is strict

we believe P ≠ NP

REDUCTIONS:

A ≤ B means A reduces to B,

- B is as hard as A,
- algorithm for B solves A

$A \leq_p B$

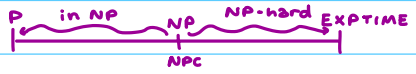
- poly time algo. for B solves A in poly time
- if $B \leq P$, then $A \leq P$

NP-HARD: for all $A \in NP$, $A \leq_p B$, B is NP-hard

key: B is as hard as everything in NP

NP-COMPLETE: if...

- in NP
- NP-Hard



CANONICAL PROBLEM: SAT

input: boolean func Φ

output: does there exist Assignment X s.t. $\Phi(x) = T$

- if there exists a poly time alg. for any NP-C problem ⇒ P = NP
- P ≠ NP ⇒ there does not exist poly time algo for any NP-C problem

2 WORLDS:

1: automated theorem proving
- P = NP ⇒ 2: public key encrytion FAILS
- P ≠ NP ⇒ best $2^{O(n)}$

- NP hard is worst case
- approx.
- SAT solvers

12/9 **COMPUTABILITY:** what is uncomputable?

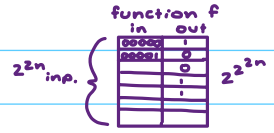
TURING (1936)

1. what is an algorithm? Turing machine
2. what is computable? what problems have algs? → linear, poly, expon.
3. what is uncomputable?

SET UP:

- $\{0,1\}^*$ = set of all bitstrings
- program = bitstring (ex: Python in ASCII)
- input as bitstring in $\{0,1\}^N$
- decision problem is just a function $\{0,1\}^N \longrightarrow \{0,1\}$ (input → YES/NO)
- def: **PROGRAM** P computes f if ∀ inputs $x \in \{0,1\}^N$, $P(x) = f(x)$
  & P always returns output ("HALTS")

Life is unfair: there are more problems than solutions.

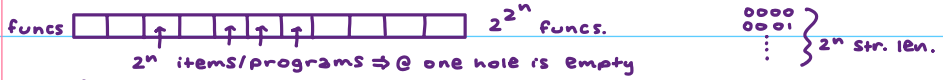


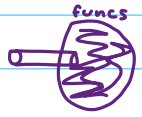
Fact: there exist functions $f: \{0,1\}^{2^n} \to \{0,1\}$ that can't be computed by any length n program

Pf: there are $2^{2^{2n}}$ such functions f. ↔ if a program computes f, it doesn't compute any other fn f.

2^n bitstrings of length n.

⇒ by pigeonhole → ∃ some fn s.t. ∄ program length n that computes it.





there are "natural" problems w/ no alg soln/no program that computes them.

HALTING PROBLEM:

input: binary string interpreted as a pair (P, x)
→ P as a program (Python prog. in ASCII)
→ x as input

output: YES if P on input x HALTS. no otherwise.

program takes a program as input

THM (TURING): no alg computes HALT ("Halt is undecideable")

PF: by contradiction. Suppose you have alg. that computes HALT.

define program P(x): if H(x,x) = yes → loop forever
otherwise → output yes

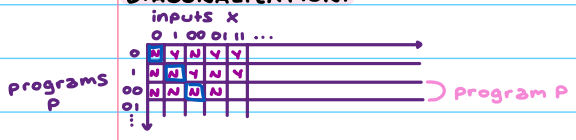
if we run P(P):

if H(P,P) = yes, loops forever // if P(P) halts, P(P) doesn't halt.

otherwise, output yes // if P(P) doesn't halt, P(P) halts.

CONTRADICT!

DIAGONALIZATION:



#diagonals are different

- there can exist programs that solve HALT on some/many inputs
  - #run for 1 day, output "unsure" if still running
  - #if python prog. consist of only 1-9, +, -, *.

TOTALITY: takes program Q as input

output: yes if Q halts on all inputs
no otherwise

THM: totality is undecidable

PF: towards contradiction, assume alg T compiles totality

then here is Halting alg: H(P,x): define program $Q(y)$ Q := "ignore input, runs P(x), outputs yes"
return T(Q) ← totality alg.

Claim: H computes Halt. ⇒⇐

1) P(x) halts → Q halts on all inputs → T(Q) = YES ✓
2) P(x) doesn't halt → Q doesn't halt → T(Q) = No ✓

Equiv ("Program equivalence")

- takes programs P, Q as input
- outputs YES if $\forall x$ $P(x) = Q(x)$

  NO otherwise

THM: Equiv. is undecidable

PF: by contradiction, assume E computes Equiv

consider H(P, x): defines Q = "ignore input, run P(x), output YES"

defines R = "ignore input, output YES"

return E(Q, R)

Claim: H' computes/solves HALT

P(x) halts ⇒ Q outputs "YES" ⇒ Equiv(Q, R) = "YES"

P(x) doesn't halt ⇒ Q loops forever, R doesn't ⇒ Equiv(Q, R) = "No"

UNDECIDABILITY IN UNEXP. PLACES:

1) testing whether multivar. poly equation w/ int coeff has an int. soln. $x^3 + 12y^3 + z^2 = 0$

2) whether a string is compressible given string x as input, is there length-ℓ program that outputs x?