6.1210 Intro to Algorithms
Fall 2023

LECTURE 1 9/7/23 11AM

Algorithms: sequence of instructions to solve a particular problem

(computational) problem: "relation between allowed inputs and output"

we want

correctness: for all inputs

efficiency: in time and memory

Addition

preschool addition:

for $i = 1, 2, \ldots, y$

$O(1) \rightarrow x \mapsto x+1$

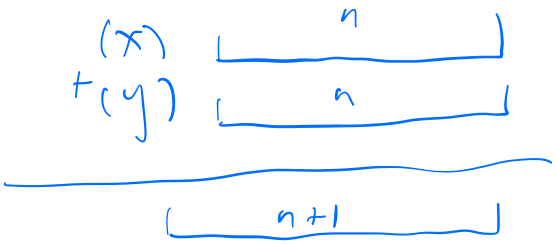
output x

$T(n)$ = runtime of preschool addition for inputs of length n

$T(n) = O(10^n - 1)$

$\rightarrow O(10^n)$ SLOW

grade school addition:



runtime:

n+1 columns (output digits)

each takes O(1) time

ξ

$O(n)$ fast (est)

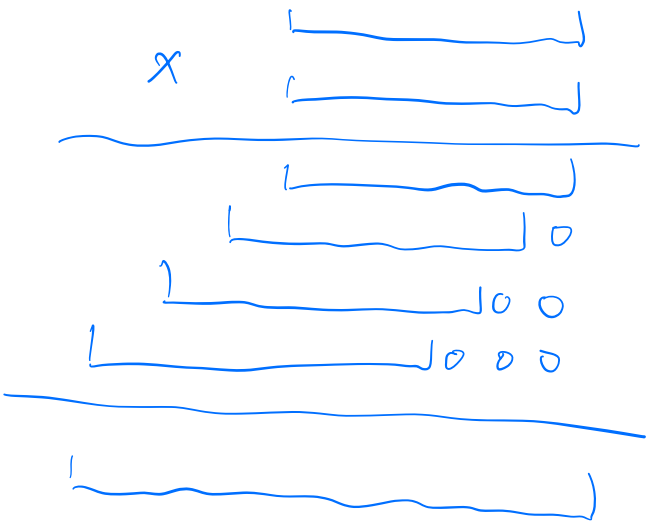
proof of optimality:

(1) reading input takes $O(n)$

(2) writing output takes $O(n)$

Multiplication

grade school multiplication:



runtime:

if each $(\times)$ takes $O(1)$,

n^2 total multiplication

$\rightarrow O(n^2)$

one idea: divide-and-conquer

$x = x_h 10^{\frac{n}{2}} + x_\ell$

$y = y_h 10^{\frac{n}{2}} + y_\ell$

$x \cdot y = x_h y_h 10^n + (x_\ell y_h + x_h y_\ell) 10^{\frac{n}{2}} + x_\ell y_\ell$

(1) divide into x_h, x_ℓ, y_h, y_ℓ

(2) find $A = x_h y_h$, $B = x_h y_\ell$, $C = x_\ell y_h$, $D = x_\ell y_\ell$.

(3) find $A 10^n + [B+C] 10^{\frac{n}{2}} + D$

$\rightsquigarrow T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow \Theta(n^2)$ w/ Master Theorem.

($\Theta(n)$: additions)

Karatsuba's Algorithm:

let $E = (x_h + x_\ell)(y_h + y_\ell)$

then $A 10^n + (B+C) 10^{\frac{n}{2}} + D = A 10^n + (E-A-D) 10^{n/2} + D$

$\rightsquigarrow T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow \Theta\left(n^{\log_2 3}\right)$

$\approx \Theta(n^{1.59})$ better than $\Theta(n^2)$.

$\downarrow$

2019 (Harvey, van der Hoeven)

$\Theta(n \log n)$

LECTURE 2 9/12/23 11AM

efficiency + divide/conquer

we are focused on minimizing time and space

(1) construct mathematical model → (2) prove thms about it

$T(n) = \max_{x, |x|=n} \text{runtime}(x)$ ↑ worst-case

$S(n) = \max_{x, |x|=n} \text{space}(x)$ ↑ worst-case

Rules of Analysis of Algorithms

- correctness
- measures worst-case
- measures asymptotically

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

$\forall n \geq n_0 \quad \forall x, |x| = n, \quad \text{runtime}(x) \leq c_0 f(n)$

$\Rightarrow T(n) = O(f(n))$

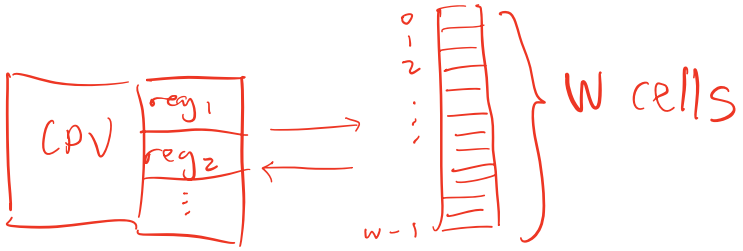
$\forall n \geq n_0 \quad \exists x, |x| = n, \quad \text{runtime}(x) \geq c_0 f(n)$

$\Rightarrow T(n) = \Omega(f(n))$

Measuring Time/Space / "n"

WORD-RAM model

random access machine/memory



↑ O(1) registers, 1 word each (i.e. 64 bits)

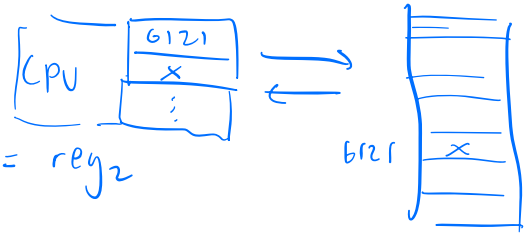
each cell is a word of 64 bits (small w)

unit-time operations

1) arithmetic on registers: + − × ÷

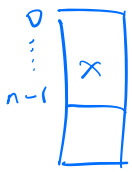
2) read/write from RAM e.g. read[reg1] = reg2 ✡

3) allocate $W \mapsto W+1$



Input/Output

Initial state of RAM:



n = # of words needed to store input

Memory Allocation

$W = n + O(1)$ initially

T = increment W by m costs m timesteps

→ but then $W \leq 2^{64}$ must be true

↑ bad since you need to be able to access every place in memory, contradiction if W keeps increasing technically

✡

fix: word size

to make $n \to \infty$ reasonable

increment w in Allocate whenever needed $w = \max(\lceil \log n \rceil, 64)$

peak-finding

input: array of integers, length n (n consecutive integers in RAM)

problem: find a peak in array A

Def A peak of A is an index i s.t. $A[i] \geq$ neighbors if they exist

Peaks always exist (take global max)

Alg 1 - loop thru A until peak found

obviously correct and $T(n) = O(n)$ ← length of array

$A = [1, 2, \ldots, n]$ yields $T(n) = \Omega(n)$

$\Downarrow$

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

Alg 2 - binary search (divide-and-conquer) - if middle index isn't a peak, there must be a peak in one of the halves (i.e. the side that prevented middle index from being the peak)

$\rightsquigarrow T(n) \leq T(\frac{n}{2}) + O(1) \Rightarrow T(n) = \Theta(\log n)$

LECTURE 3 9/14/23 11AM

Data Structures

- interfaces (ADT) are defined by
  - ~ what objects can be stored
  - ~ what objects are accessible

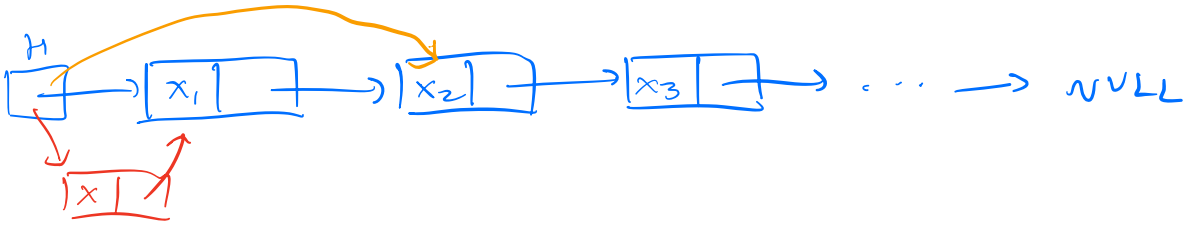
ex) Stack → LIFO, any kind of object

pop(): returns / deletes the top

push(x): stores x at top

implementations are how we store data according to our interface (many implementations can exist for one interface)

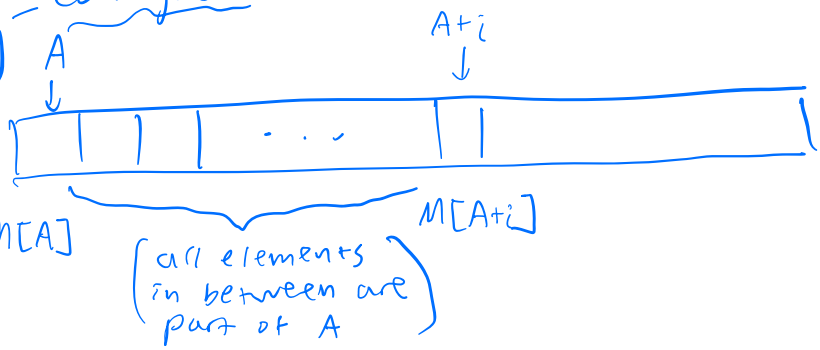
Linked List



push(x): relink head O(1)

pop(): remove x_1, return x_1 O(1)

Array - contiguous chunk of cells



each cell is a different word in WORD-RAM model

let top = first empty slot in A

⇒ accessing M[A+i] is O(1)

push(x): A[top] = x, top++ O(1)

pop(): top--, return A[top] O(1)

but if we are at the end of our memory, we need to reallocate A to A' somewhere else

① allocate A'
② copy A into A'
} $\Theta(A + A')$

Amortized Analysis

Def A collection of operations has amortized complexity T if any sequence of k operations takes time $\leq k \cdot T$

i.e) per operation $\leq T$ time

Array Doubling

when A is full, resize into A' twice the size of A

thus $T(\text{resize}) = O(3A) = O(A)$

for the next A pushes, $T(\text{push}) = O(A)$ total

WTS k pushes take $O(k)$ time, i.e.,

amortized cost $O(1)$.

proof we need to have array size $= 2^{\lceil \log k \rceil}$,

which requires allocations taking

$$O(1 + 2 + 2^2 + \cdots + 2^{\lceil \log k \rceil}) = O(k)$$

we have also pushed k times, thus we have

average time $\frac{O(k)}{k} = O(1)$.

Memory Overhead

- memory used, in addition to memory needed to store objects, as a function of n.

memory overhead of linked list is $O(n)$ due to n pointers.

for arrays, we could get large overhead if we push many elements and then pop almost all.

fix: Array Halving

- reallocate to size $\frac{n}{2}$ array if # of elements $\leq \frac{n}{2}$ (just in case we need to push again, we don't want to have a tight bound)

Sequences - new interface

- build(A) } container
- len()
- iter_seq() → returns elements in order
- get_at(i)
- set_at(i,x)
- index_of(x) } static

- insert_at(i,x)
- delete_at(i)
- insert_first(x)
- delete_first()
- insert_last(x)
- delete_last()

} special cases of (but usually implemented faster)

(order)	container	static	insert_at delete_at	insert_first delete_first	insert_last delete_last
linked list	n	n	n	1	n
doubly-linked list (pointers to before, after, end)	n	n	n	1	1

dynamic array	n	1 index_of $\Rightarrow n$	n	n as is 1 (amortized) ↗ recitation 3	1 (amortized)

set - collection of objects each with a key

$\Rightarrow$ key-value map

LECTURE 4 9/19/23 11AM

Sort - to make things organized

input: list of n items with some sort of relationship between them

comparisons take $O(1)$

① Brute Force - check all $n!$ permutations

BAD! $O(n!) = O(n^n)$.

② Insertion Sort

UNSORTED

NUSORTED

NSUIORTED

⋮

ENORSTUID

DENORSTU

for $i = 1$ to $n-1$

insertion { insert $A[i]$ into correct spot in $A[0:i-1]$

Claim - insertion sort is correct

Proof induction via recursion.

obvious for $n = 1$,

assume up to $n-1$.

Then def ins_sort2([A]):

ins_sort2(A[:n-1]) ← inductive hyp

insert(last_element)

↑

correct since all other n-1 are sorted, n^{th} element must be correctly place at some point, since all placements are checked □

Complexity: $T(n) = T(n-1) + t_{insert}(n)$

↑

$O(n)$

$$T(n) \leq T(n-1) + cn$$

$$\leq c \cdot n + c(n-1) + T(n-2)$$

$$\vdots$$

$$\leq c(n + (n-1) + (n-2) \dots)$$

$$\leq c \cdot \frac{n(n+1)}{2} = O(n^2).$$

$T(n) = \Omega(n^2)$ by taking "$n, n-1, \dots, 1$"

$\therefore T(n) = \Theta(n^2)$

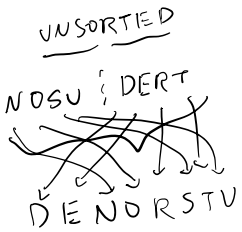
- stable: preserves order of equal elements
- in place: overhead just $O(1)$

now try **divide and conquer**

$$T(n) = \sum_k T\left(\frac{n}{k}\right) + t_{merge}(n)$$

```
def merge_sort(A):
    L = merge_sort(A[0 : n/2])
    R = merge_sort(A[n/2 : n])
    return merge(L, R)
```

$T(n) = 2T\left(\frac{n}{2}\right) + t_{merge}(n)$ ✓




```
def merge(L, R):
    out = []
    top_L = top_R = 0
    while:
        pick smaller of L[top_L], R[top_R]
        append to out
        increment top_i of side selected.
```

Claim: correct by induction

induct on length of out

hypothesis: out is sorted and ≤ everything "left" in L and R

inductive step:

assume out is sorted, ≤ L & R.

WLOG add $L[top_L]$ which means

$L[top_L] \le R[top_R]$ and $L[top_L] \le L[top_L + 1]$

⇒ out is still sorted, and ≤ L & R that's left.

Assume merge is correct; Claim: merge_sort is correct

proof: by induction, L and R are correct,
by correctness of merge, output is correct.

$t_{merge}(n) = O(n)$ the for n iterations of O(1) time

$\rightsquigarrow$ $T(n) = 2T(\frac{n}{2}) + O(n)$ $\rightsquigarrow$ $\boxed{T(n) = O(n \log n)}$
$T(n) \leq 2T(\frac{n}{2}) + cn$

not in-place, but stable

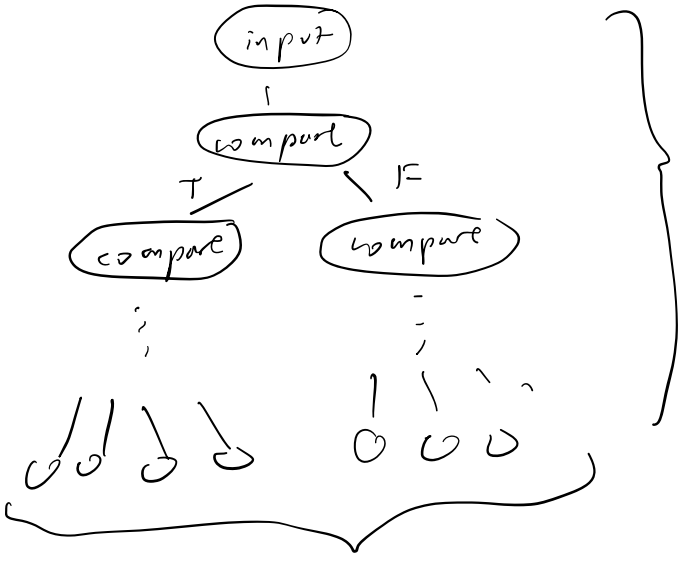
LECTURE 5 9/21/23 IAM

Data Structure	Operations O(·)				
	build	find	insert/delete	find_min/max	find_pre/next
Array	n	n	n	n	n
sorted Array	n log n	log n	n	1	log n

90% - 9% - 1% rule - 90% finding
9% adding
1% deleting

we have been using comparison model so far

Proof that comparison model will always use $\Omega(\log n)$ regardless of algorithm:



worst case number of comparisons is $\Omega(\log n)$.

$n+1$ at least,

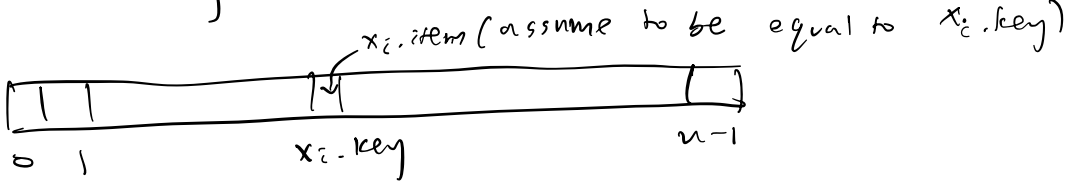
n for found elements

1 if not in list

Let's instead try a Direct Access Array

- if $x_1.key, \ldots, x_n.key \in \{0, 1, \ldots, u-1\}$.

then make array of size u



- but order operations (find_next) takes $O(u)$ if there are big gaps
- we also need memory $O(u)$ ⇒ BAD

Hashing

find(x) → O(1)

build( ) → O(n)

sacrifice performance of find_next, find_min, etc.

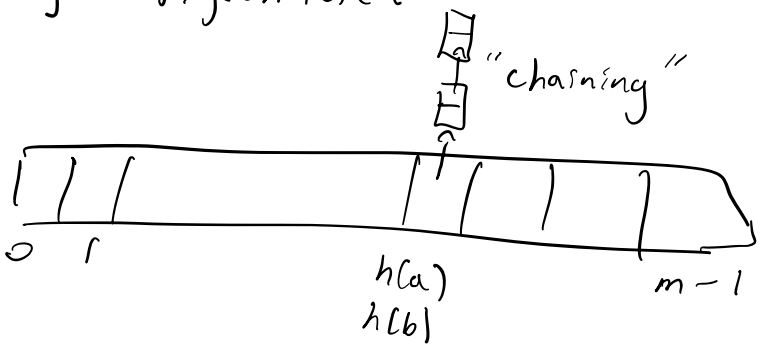
make hash function

$h(x) : \{0, 1, \ldots, n-1\} \to \{0, \ldots, m-1\}$

$m \ll n$

assume all keys are unique

but then $\exists\ h(a) = h(b)$ for $a, b \in \{0, 1, \ldots, n-1\}$ by pigeonhole.



$m = \Theta(n)$

if we can build array with each entry having chain of length at most $O(1)$, then we can find(x) in $O(1)$

- apply hash function
- look through chain

so, what hash functions will let chains be all $O(1)$?

Heuristic $H(k) = k \bmod m$ → okay w/ randomness

↳ large primes $m = \Theta(n)$ far from powers of 2, 10 are good.

↳ otherwise we could get lots of overlaps (e.g. all bits end in 0 but $m = 2^k$), wasted space

Simple Uniform Hashing Assumption

Each key x_i maps to a uniformly random location from hash function independent of other x_i.

$Pr[h.k = i] = \frac{1}{m} \quad \forall\, i, k.$ → uniform

$Pr[h.k = i \cap h.k' = j] = \frac{1}{m^2} \quad \forall\, i, j, k, k'.$ → independent

let load factor $= \frac{\#\text{ elements}}{\text{table size}} = \frac{n}{m} = \alpha$

running time : $O[\text{size of chain in } h(k)]$

$E(\text{size of chain at } h(k)) = \frac{1}{m} \sum_{i=1}^{m} (\text{size of chain at index } i)$

$= \alpha$

Then finding/returning is $O(1) \Rightarrow$ ans $= \boxed{O(1+\alpha)}$

we can also fix m as needed if load factor is too large (similar to array doubling/halving)

Data Structure	Operations $O(\cdot)$				
	build	find	insert/delete	find_min/max	find_prev/next
Array	n	n	n	n	n
sorted Array	$n \log n$	$\log n$	n	1	$\log n$
Direct Access Array	u	1	1	u	u
Hash Table	n_{avg}	1_{avg}	$1_{avg\ am}$	n	n

LECTURE 6 9/26/23 11 AM

Using comparisons, we would prove $\Omega(\log n)$ for finding.

Similarly, we have $n!$ possible sortings so using a tree of comparisons guarantees at least $\Omega(\log n!) = \underline{\Omega(n \log n)}$ time.

Linear-Time Sorting Algorithm - Radix Sort

Step ① Direct Access Array.

if all keys are in $\{0, 1, \ldots, u\}$

then simply insert each element of A into an array D of size u,

s.t. $D[x.key] = x$.

Then obviously we are sorted $\rightarrow \Theta(u)$.

Step ② what if repeats??

→ send them to the same place in D,

but put them in a sequence (queue) FIFO

STILL a ↝ Stable Sort

"Counting Sort"

Step (3)

turn x.key into tuple in base n.

e.g. for $n \sim n^2$, let $y.key = \left(\left\lfloor \frac{y}{n} \right\rfloor, y - \left\lfloor \frac{y}{n} \right\rfloor \cdot n\right)$

$\uparrow a$ $\quad \uparrow b$

s.t. $y = an + b$

Now suppose each

$x.key = (x.k_1, x.k_2, \ldots, x.k_c)$

most significant ↖ ↗ least significant.

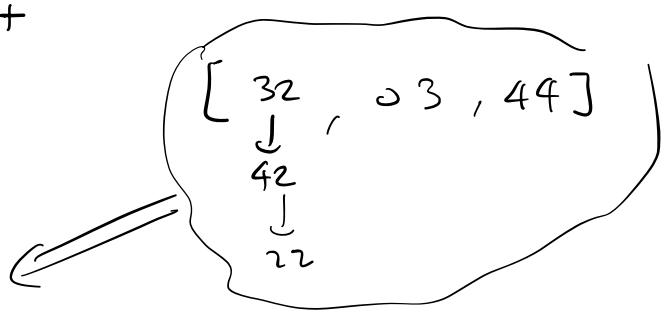
sort lexicographically c times, starting from

LEAST significant digit first

$[32, 03, 44, 42, 22]$

$\rightarrow [32, 42, 22, 03, 44]$

$\rightarrow [03, 22, 32, 42, 44]$.

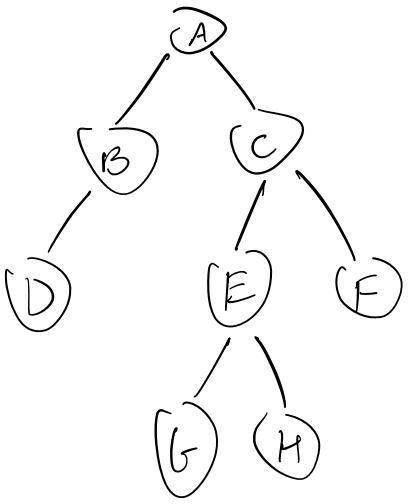


each sort takes $O(n)$ ⇒ total time $\boxed{O(cn)}$.

LECTURE 7 9/28/23 11AM

Binary Trees

parents, childs



ancestor = any node in path from x to root (including root)

descendants = anything in right/left subtree

depth = # of ancestors

height = max (depth over all leaves in subtree)

define height (tree) = depth (tree)

Traversal

- in-order: left, node, right ← we care about this one
- pre-order: node, left, right
- post-order: left, right, node

ex) {D, B, A, G, E, H, C, F}.

from above

```
find_min(x):
    if !x.left : return x
    else: return find_min(x.left).
```

⇐ O(h)

```
find_next (x):
    if  x.right:
        return  find_min (x.right)
    else:
        go up thru x's ancestors, until
        x.parent.right ≠ x.  return x.parent.
```

Binary Search Property - in a node, all of its left subtree nodes are less than node's key, and all its right subtree nodes are more than node's key.

```
insert (T, x):   ~ insert into tree a key x
    compare with T.left, T.right:
    if (x > T.key):
        if !T.right, T.right = x
        otherwise,
        insert (T.right, x)
    if (x < T.key):
        if !T.left, T.left = x
        otherwise,
        insert (T.left, x)
```

$\Leftarrow O(h)$

```
delete (T, x):
    while True:
        if  x.right:
            y = find_min (x.right)
            swap (x,y)
        elif  x.left:
            y = find_max (x.left)
            swap (x,y)
        else:
            delete x
```

$\Leftarrow O(h)$

if we can have $h = O(\log n)$, we win.

build(T): n inserts, each $O(h)$

$\Rightarrow O(n \log n)$ if $h = O(\log n)$.

LECTURE 8 10/3/23 11AM

We'd like to keep track of our nodes and their "indices" implicitly, so we use Tree Augmentation

→ special values about the tree, depends only on children, can be calculated in $O(1)$

— subtree size: node.size = node.left.size + node.right.size + 1

— height: node.height = max(left.height, right.height) + 1

⇒ can recursively build using post-order transversal
⇒ $O(n)$ to initialize.

```
get_at(X, i):
    j = size(X.left) + 1
    if i == j, return X.value
    elif i < j, return get_at(X.left, i)
    else, return get_at(X.right, i - j)
```

insert_at, set_at, delete_at works similarly, $O(h)$ time

→ find X_i, then if it has left node, add node as right child of max element in left node
else: add node as left child of X_i

both require modifying size augmentations, $O(h)$

to climb up tree

→ find X_i, delete if it has no children

else: swap X_i with either $\min(X_i.\text{left})$ or $\max(X_i.\text{right})$ and delete.

```
index_of(x):
    i = 1                          ← 1-indexing
    if (x.left):
        i += x.left.size
    while x has parent:
        if x.parent.right = x:
            i += x.parent.left.size + 1
        x = x.parent
    return i
```

build(x) takes $O(n)$ since we can recurse on our already-sorted $X_1, X_2, \ldots, X_n$.

Balanced Binary Trees

AVL Property - for any node x, let skew = height of x.right − height of x.left

⇓

if skew ≤ 1 $\forall$ nodes x.

Notice that $h = O(\log n)$ for an AVL tree of n nodes, height h.

PF Let n_h = min [nodes in AVL of height h].

$n_0 = 1$ •

$n_1 = 2$ /

notice $n_h \geq 1 + n_{h-1} + n_{h-2}$

$\geq 1 + n_{h-2} + n_{h-2}$

$\geq 2n_{h-2} \geq 4n_{h-4} \geq 8n_{h-6} \geq \cdots \geq 2^{h/2}$

$$\Rightarrow n_h \geq 2^{h/2}$$

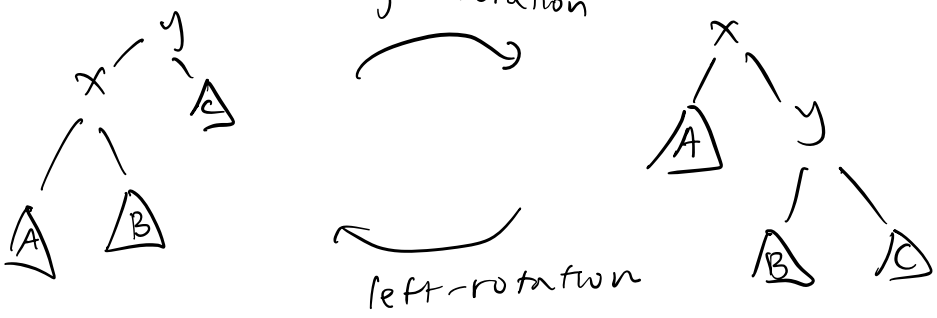
$$\log(n_h) \geq \frac{h}{2} \Rightarrow h \leq 2\log(n_h)$$

$$\leq 2\log(n) = O(\log n)$$

↑ n_h is minimum for height h.

Rotations

operation $O(1)$ time



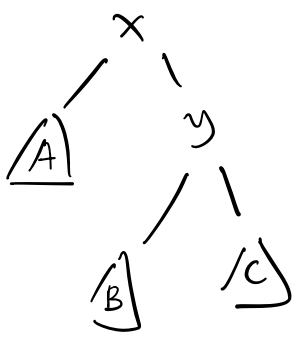
in-order transversal is same for both

after an insert/delete, skew ≤ 1 might no longer be true.

suppose x is the first node (when moving up from modified leaf) s.t. $|skew| > 1$.

we know $|skew| = 2$, WLOG skew $= 2$.

then

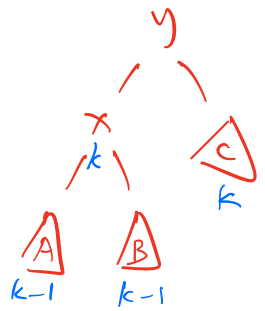


$h(x) = k+2$

$h(y) = k+1$

$h(A) = k-1$

$y.skew = 1$: rotate to get



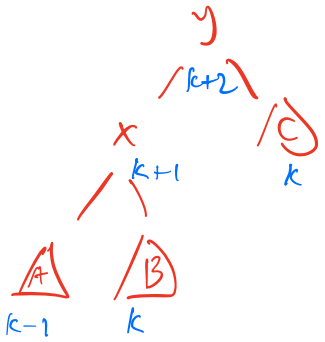
$h(B) = k-1$

$h(C) = k$

$\Rightarrow$ skew$(x) = 0$

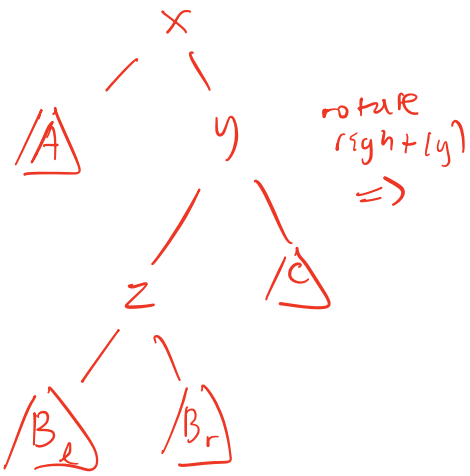
skew$(y) = 0$

y.skew = 0: rotate to get

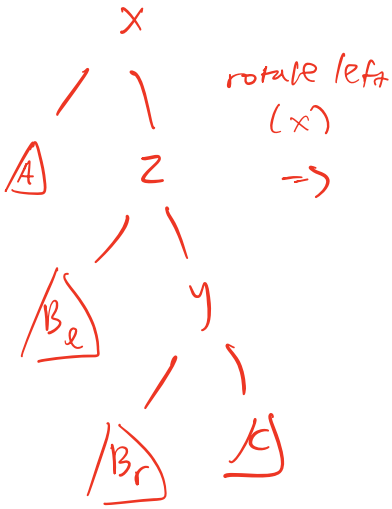


$h(B) = k$
$h(C) = k$
$\Rightarrow$ skew(x) = 1
skew(y) = -1.

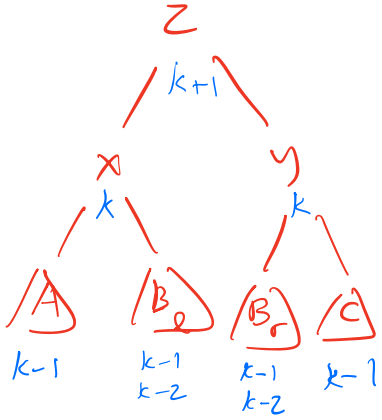
y.skew = -1: zoom in:



rotate right(y) $\Rightarrow$



rotate left(x) $\Rightarrow$



all |skew| ≥ 2 resolved.

$h(z) = k$
$h(C) = k-1$

$h(B_\ell)$, $h(B_r)$ } $k-1$ or $k-2$

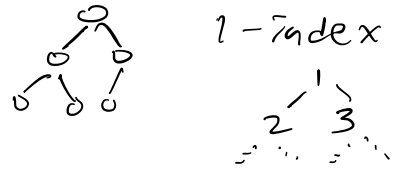
$\Rightarrow h = O(\log n)$ for AVL tree now.

LECTURE 9 10/5/23 11AM

Priority Queues

we want build()
insert(x)
delete_max(x) ← delete + return largest key

using	AVL	Heaps
build( )	$O(n \log n)$	$O(n)$
insert(x)	$O(\log n)$	$O(\log n)$
delete_max(x)	$O(\log n)$	$O(\log n)$



Heaps - are compact trees

(1) complete; every node has 0 or 2 children except at most 1

(2) left-justified

→ implicitly stored as an array with length n with pointers:

$left(i) = 2 \cdot i$

$right(i) = 2 \cdot i + 1$

$parent(i) = \lfloor i/2 \rfloor$

heap property $x.key \geq x.right.key$, $x.key \geq x.left.key$

→ by transitivity, $x.key \geq$ all keys of nodes in its subtree

- heapify_up( ): fixes heap property if $x.key > x.parent.key$
  ⇒ swap w/parent till done $O(\log n)$
- heapify_down( ): fixes HP if $x.key < x.left.key$ or $x.right.key$
  ⇒ swap w/ largest child till done $O(\log n) = O(height(i))$

to implement PQ

insert(x): put at end of array and heapify-up

$O(\log n)$ (am)

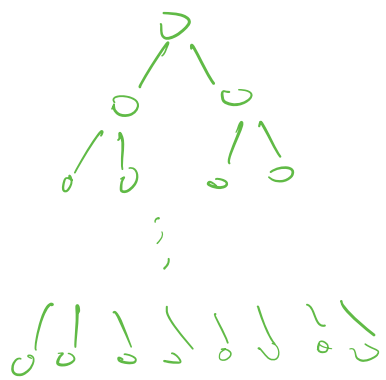
↑ if we need to increase size

delete_max(x): swap first with last, delete the last, then heapify-down()
to validate HP

$O(\log n)$

build(A):
first add all elements to array.
then heapify_down(i) for $i = n, n-1, \ldots, 1$.

runtime:



height	#
k	$m/2^k$
$k-1$	$m/2^{k-1}$
⋮	⋮
2	$m/4$
1	$m/2$
0	m

$n \leq 2m$

$$\Rightarrow O\left(\underbrace{m \cdot 0 + \frac{1m}{2} + \frac{2m}{4} + \frac{3m}{8} + \cdots + \frac{km}{2^k}} \right)$$

$$\frac{m}{2} + \frac{m}{4} + \frac{m}{8} + \frac{m}{16} + \cdots = m$$

$$\frac{m}{4} + \frac{m}{8} + \frac{m}{16} + \cdots = \frac{m}{2}$$

$$\frac{m}{8} + \frac{m}{16} + \cdots = \frac{m}{4}$$

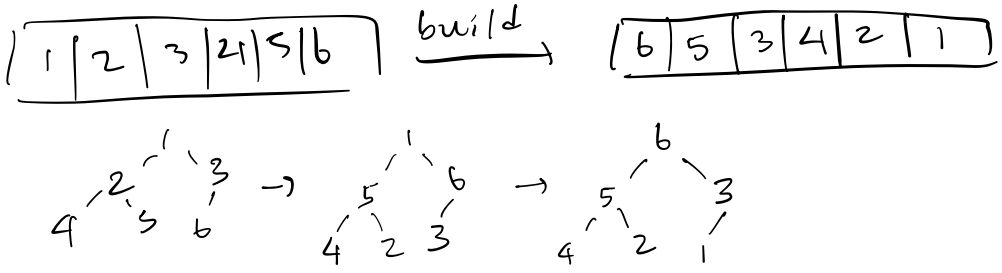
$= O(2n) = O(n)$ time !?!

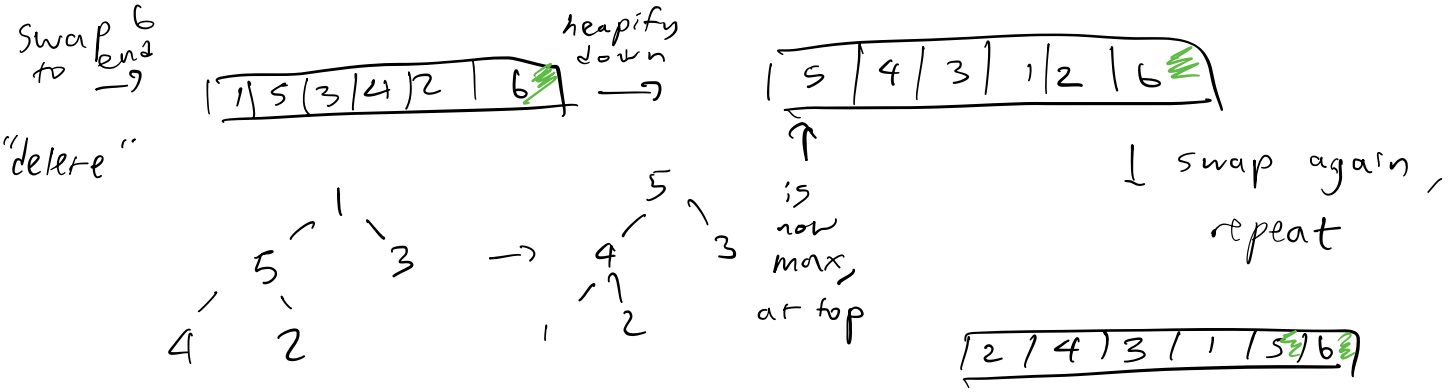
given our heap built in $O(n)$ time, we can actually make HEAP_SORT (PQ-SORT) in $O(n \log n)$.

1. build (A)
2. while !empty:
   delete_min

symmetric as for delete_max

HEAP_SORT is IN PLACE. (not STABLE)

e.g. 



LECTURE 10 10/12/23 11AM

Graphs.

V = set of vertices

E = edges between vertices, directed or undirected

$u \to v$ (u,v) — directed

$u - v$ $\{u,v\}$ — undirected

we use

Simple graphs

each edge is between two distinct nodes,

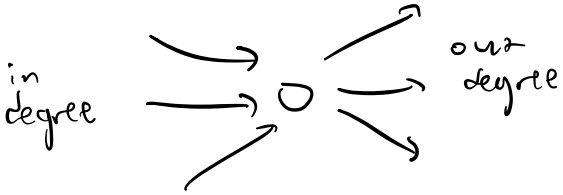
each pair of (undirected) edges appears at most once.

for $G = (V, E)$. $n = |V|$, $m = |E|$

then $m = |E| \leq \binom{n}{2}$ for undirected edges

$m = |E| \leq 2 \cdot \binom{n}{2}$ for directed edges

in Directed Graphs



$\overleftarrow{Adj}(u) = \{ v \in V \mid (v,u) \in E \}$

vertex

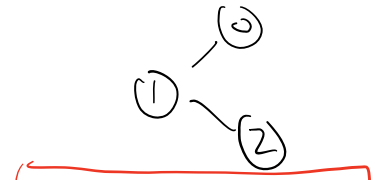
$\overrightarrow{Adj}(u) = \{ v \in V \mid (u,v) \in E \}$

$Adj(u) = \{ v \in V \mid \{u,v\} \in E \}$

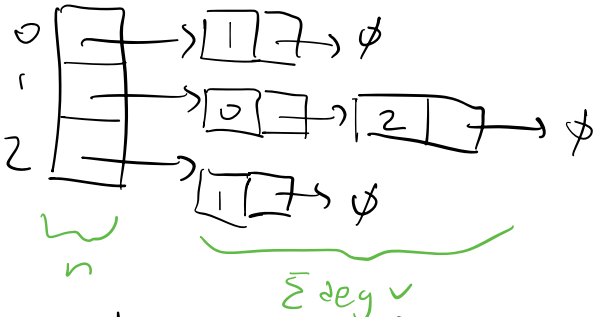
Handshake Lemma

$\sum deg(v) = 2m$

Graph Representations



$\Rightarrow$



Adjacency List

array of linked lists. $\rightarrow$ only good for sparse graphs

total size $= O(n + \sum \deg v) = O(n+m)$.

"linear time"

A path is a sequence $(v_1, v_2, \ldots v_k) \in V^k$

where $(v_i, v_{i+1}) \in E \quad \forall \; 1 \le i \le n-1$.

A simple path does not repeat vertices.

The length $\ell(p)$ of a path p is the number of edges: $k-1$.

The distance $d(s,v)$ is the length of the shortest path from s to v. $d(s,v) = \infty$ if no such path exists.

Problem: Single Source Shortest Paths:

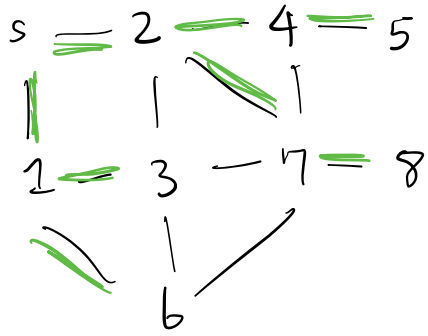
Compute $d[s,v) \quad \forall \; v \in V$ for given $s \in V$.

↳ inefficient to return every shortest path separately. use shortest path trees:

- for all $v \in V$, store parent $p(v)$ – second-to-last vertex on the shortest path from s.

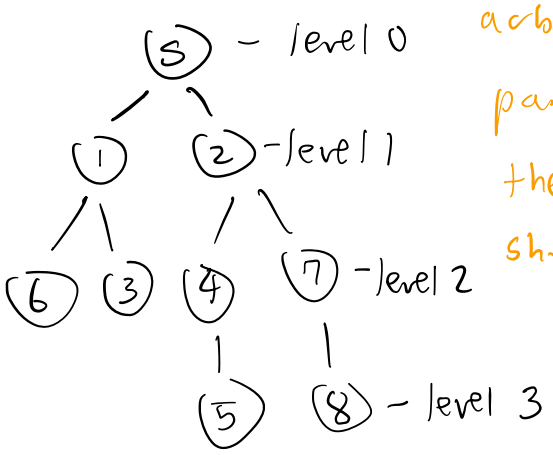
shortest path tree – set of all $p(v)$ given s. (size n)

ex)



9 – 10

↳ SPT =



pick arbitrary parent if there is a shortest path tie

BFS (s)

(1) initialize as $d(s,v) = \infty$ $\forall v$, $P(v) = \text{none}$ $O(n)$

(2) Base $d(s,s) = 0$, $P(s) = \text{None}$ ← L_0 $O(1)$

(3) Induction to compute level L_i, $O(n+m)$

$\forall u \in L_{i-1}$, check $\forall v \in Adj(u)$ with $d(s,v)=\infty$

↗ each edge checked at most twice → $\sum \deg v = O(m)$

if so, set $d(s,v)=i$, $p(v)=u$. add v to L_i

(4) Repeat until L_{i-1} is empty.

⇒ $O(n+m)$ → linear.

LECTURE 11 10/19/23 11AM

DFS - Depth First Search

- ○ - white — unvisited
- ⊘ - gray — reached
- ● - "black" — finished

1. initialize all nodes as white, $p(s)=$ none, $t=1$
2. visit(s)

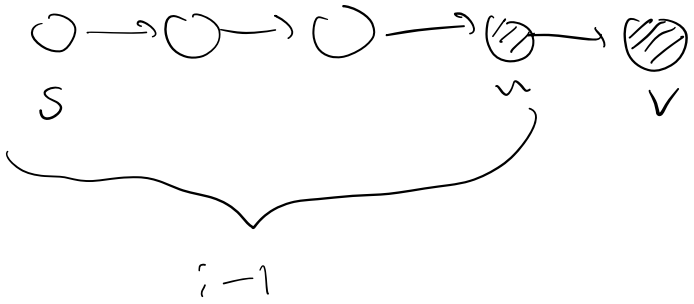
```
def visit(u):
    u.color = gray
    k[u] = t, t=t+1
    for every v ∈ Adj(u)        ← Adj+(u) for directed graph
        if v.color = white then p(v)=u and call
```

visit(v)

u.color = black

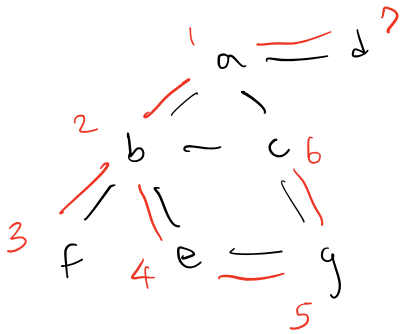
pf Sketch

induction on distance from s.



↳ all gray nodes form a simple path from s.

So v is reached.



⇒ a b f f e g e c g e b d d a .

time when g was gray

DFS Runtime - O(1) per edge, no vertex constraint

⇒ O(|E|).

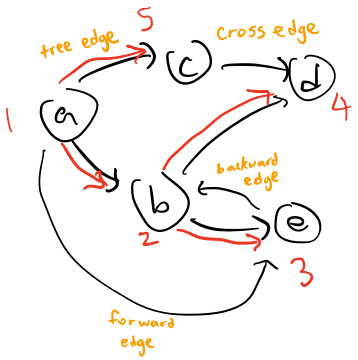
Edge Classification

for an edge $(u) \to (v)$,

when traversing

- tree edge $(u) \to (v)$ — v is white
- forward edge, if v is descendant of u in DFS tree — k(v) > k(u) and v is black
- backward edge, if u is a descendant of v in DFS tree — v is gray and k(v) < k(u)

- cross edge, if u, v are not descendants of each other (v is black and $f(v) < f(u)$)



$\exists$ back edge $\hookleftarrow$ $\exists$ cycle

Full DFS

$O(|V|+|E|)$, runs on all connected components of graph by re-running on each unvisited vertex

(keep list of all vertices, iterate on it ONCE)

a b e e d d b c c a

$\Rightarrow$ e d b c a

$\Rightarrow$ a c b d e $\Leftrightarrow$ topological order.

1 2 3 4 5

Directed Acyclic Graph (DAG) – directed graph, no directed cycles.

Topological Sort $O(|E|)$.

- the order in which each vertex finishes (not starts)

in a DFS $\Leftrightarrow$ reverse of topological order, where $(u,v) \in E \Rightarrow f(u) < f(v)$

($f(u)$, $f(v)$: finish times.)

PF Sketch

for  ,

if u is visited before v

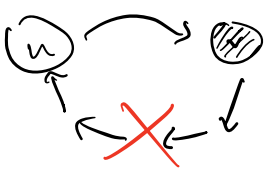
↳ $f(v) < f(u)$

$\Rightarrow u$ is before v in topological order.

if v is visited before u

↳ if $f(v) > f(u)$, then cycle exists, but DAG

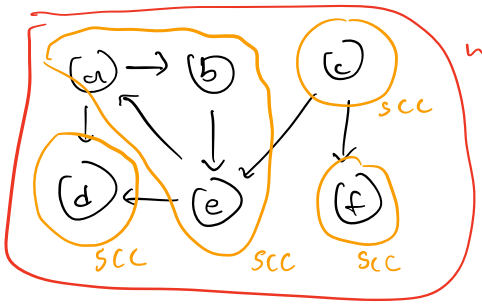
$\Rightarrow f(v) < f(u)$



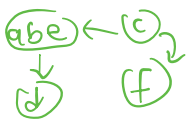
LECTURE 12 10/24/23 11AM

recall connected components from graph theory

Strongly connected component - u and v are in the same SCC if ∃ directed path from u to v and v to u.



WCC weakly connected components only need a path in the undirected version of the graph



Condensation graph

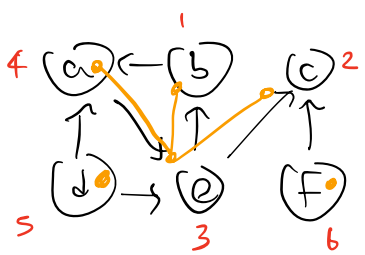
- Vertices are just the SCC's of G
- ∃ an edge from scc a to SCC b if ∃ a vertex in a and a vertex in b s.t. $(a,b) \in E$.

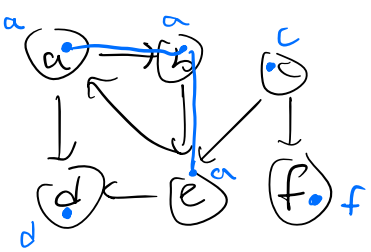
easy to see and prove that condensation graphs ARE DAGs

Finding SCCs Algorithm

G, G^{rev} = all edges reversed.

① run full DFS on G^{rev}, record finishing time of each vtx in f[v]

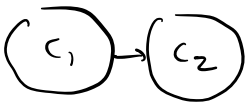




② for each u in reverse finishing time order,
set current = v.
if v unvisited, start DFS at v and for every u visited s.t. leader[u] = None, set leader[u] = current

Lemma for Correctness (proves no "spilling" into vertices not in SCC)

If C_1, C_2 are two SCC's of G s.t. there's an edge (C_1, C_2) in the condensation graph, then $\max_{v \in C_1} f(v) < \max_{v \in C_2} f(v)$



Proof

Case 1: suppose step ① visits node in C_1 before any node in C_2. But if we have an edge from $C_1 \to C_2$ in G, in G^{rev} we don't. So C_1 will be fully explored w/o ever going to C_2.

$\Rightarrow \max_{v \in C_1} f(v) <$ any $f(v)$ of $v \in C_2$

Case 2: suppose ① visits C_2 before anything in C_1.

then the first DFS will cross over using the edge $C_2 \to C_1$ in G^{rev} and fully explore C_1 before backtracking to C_2.

$\Rightarrow \max_{v \in C_1} f(v) < \max_{v \in C_2} f(v)$ □

Proof of Algorithm Correctness (2 DFS's $\Rightarrow O(|V|+|E|)$.

i.e. All v: leader[v] is a vertex in the same SCC as v, and all vertices in same SCC have the same leader.

Induction - assume all vertices visited so far are correctly labeled, and every vertex in the same SCC as a visited vtx has been labeled.

Claim DFS starting from unlabeled v labels exactly vertices in the SCC of v.

PF Starting at v, we either hit vertices in C_v or in C'

where

if we get to a $u \in C'$, by Lemma we get $\max_{w \in C'} f(w) > \max_{v \in C_v} f(v)$ since $(v,u) \in G$.

then done since correct.

So suppose w' achieves $\max_{w \in C'} f(w) = f(w')$. Then w' must have already been DFS'ed via inductive hypothesis, and $u \in C'$ so u must have been, too. So u does not get leader[u]$=v$, proving correctness.

LECTURE 13 10/26/23 11AM

So far

BFS (10)

DFS (11)

Topo. Sort (11)

SCCs (12)

Problem - given directed graph $G = (V, E)$

source $u \in V$

target $v \in V$

weights for edges w.

find shortest path from $u \to v$

Σ weights minimized.

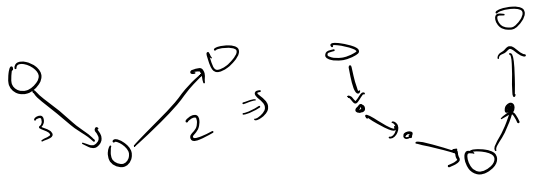
SSSP $\Rightarrow$ all targets.

If all $w(e) = 1$,

we use BFS.

Even if all $w(e)$ are positive integers from $1, 2, \ldots, N$

we can replace



but can blow up runtime by $O(N)$.

w can also be

non-integral (e.g. travel times)

non-positive (planning for future)

Dijkstra's does only for non-negative real numbers (fitting in a word)

Defn a path $(u_1, u_2), \ldots, (u_{n-1}, u_n)$ has cost $w(u_1, u_2) + \cdots + w(u_{n-1}, u_n)$

Defn $\delta[v]$ = distance (shortest path) from $u \to v$

u known

$\delta(a, b)$ = shortest distance from $a \to v$

$d[v]$ = "best so far" path from u to v's distance

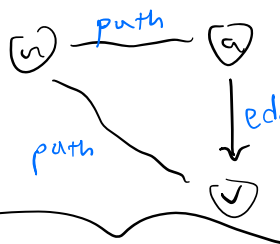
parent$[v]$ = previous node on best path $u \to v$ so far

Optimal subproblem property

Fact: if p is a shortest path from $u \to v$ and a is on this path, p is a shortest path from $u \to a$ (pf. by contradiction)

Relaxation

Triangle Inequality:



$\delta[v] \le \delta[a] + w(a,v)$

(otherwise just take path thru a instead)

```
def relax(a,v):
    if d[v] > d[a] + w(a,v):
        d[v] = d[a] + w(a,v)
        parent[v] = a
```

Safety Lemma:

relax preserves $d[v] \ge \delta[v] \quad \forall v \in V$.

pf by induction, (just follow path)

```
def Dijkstra(G, w):
    ∀ v, d[v] = ∞, parent[v] = ∅
    d[u] = 0.
    S = {u}
    pq = PQ.build(d)
         ↑ min-heap.
    while pq:  ← O(|V|)
        v1 = pq.delete_min()  ← O(|V|) times
        S.add(v1)
        for (v1,v2) ∈ E  ← O(|E|)
            relax(v1,v2)
            pq.update(v2) ~ pointers.
    return d, parent
```

starts with u

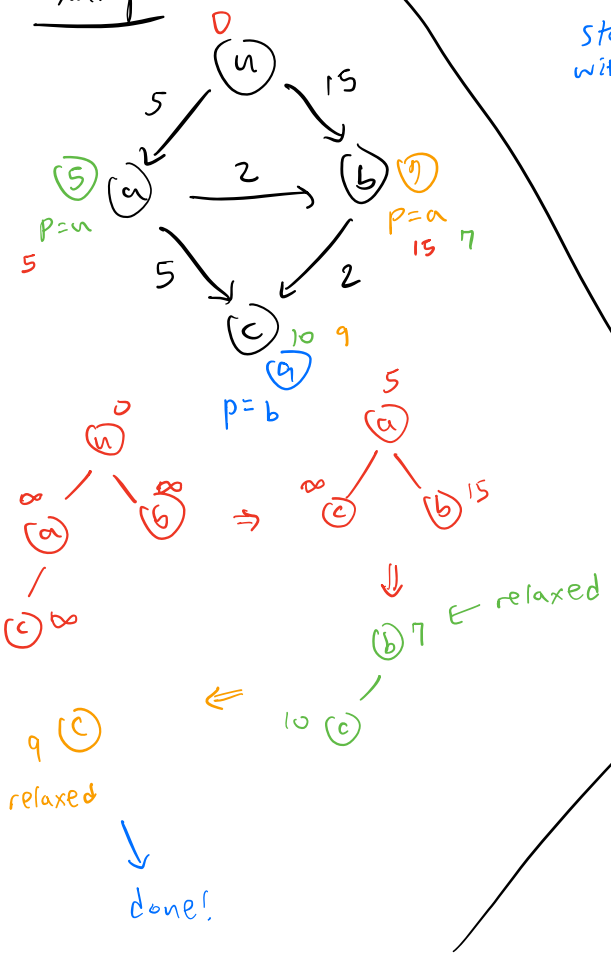
$O(|V| + |V| \cdot pq.\text{delete\_min} + |E| \cdot pq.\text{update})$

$= O(|V| \log|V| + |E| \log|V|)$ for binary heap.

$O(|V|^2)$ for DAA

$O(|V| \log|V| + |E|)$ for Fibonacci Heaps.

Example



Correctness

Claim: when we add v to S, $d[v] = \delta[v]$.

pf Induct. on # of vertices added to S.

Suppose (base case easy) $\forall z \in S, d[z] = \delta[z]$.

Induction Step – add new v to S.

Lemma: if $\exists$ s.p. $p: u \to y$ and y is the first vertex in p outside S, then $d[y] = \delta[y]$.

pf Suppose $x \in S$ is right before y. Then when x was added, $d[y] = \delta[x] + w(x,y) = \delta[y]$ was set correctly.

Now if we add v, note that $d[v] \le d[y] \quad \forall y \notin S$.

but by Lemma $\min(d[y]) = \min(\delta[y]) \le \delta[v]$. So $d[v] \le \delta[v]$

(y is along s.p. from $u \to v$)

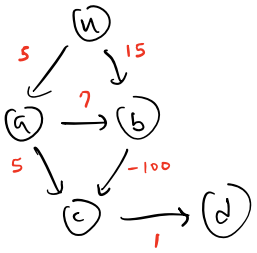
$\delta[v] \le d[v]$ by Safety Lemma

$\Rightarrow d[v] = \delta[v]$. □

LECTURE 14 10/31/23 11 AM

Dijkstra's can fail for negative-weight edges

ex)



Path Relaxation Lemma

Given shortest path $p = u \to u_1 \to u_2 \to \dots \to u_k$.

Suppose we relax them in order (with other relaxations possibly interspersed).

Then $\delta[u_i] = d[u_i] \quad \forall u_i$.

Pf by Induction

Hypothesis At the i^{th} step, $d[u_j] = \delta[u_j]$ for $j = 0, 1, \dots, i-1$

Induction $d[u_i] = d[u_{i-1}] + w(u_{i-1}, u_i)$

inductive hypothesis → $= \delta[u_{i-1}] + w(u_{i-1}, u_i)$

shortest path → $= \delta[u_i]$

Upshot

if we can find some ordering of edges (possibly repeated) s.t. every shortest path is a subsequence, relaxing in this order makes d correct for all v.

claim our order satisfies this !!

then relax in order

$e_1, e_2, \dots, e_{|E|}$

$e_1, e_2, \dots, e_{|E|}$

$e_1, e_2, \dots, e_{|E|}$

$\vdots$

$|V|-1$ times

Dijkstra

cleverly find ordering using PQ.

Bellman-Ford

number edges in E arbitrarily

$e_1, e_2, \dots, e_{|E|}$

works because

every shortest path has $\leq |V|-1$ edges

(otherwise cycle, but then we can repeat this cycle arbitrarily to get more (possibly shorter) paths)

unless we have negative-weight cycles

```
def BF(G, w):
    init d, parent   ← lists
    E = enum_edges(G)
    for i = 1,2,..., |V|-1:
        for e in E:
            relax(e)
    return d, parent
```

```
for e in E:
    if e can be relaxed:
        ABORT
```

∃ negative-weight cycle.

(proof below)

Proof

($\Rightarrow$) if no negative weight cycle, $|V|-1$ times relaxes all edges to get correct δ's $\Rightarrow$ no relaxable edges left

($\Leftarrow$) suppose ∃ negative weight cycle. $v_1, v_2, \ldots, v_k$

$$\Rightarrow \sum_{i=1}^{k} w(v_i, v_{i+1}) < 0.$$

$$\sum_{i=1}^{k} \left(d[v_i] + w(v_i, v_{i+1})\right) < \sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i+1}]$$

$$\Rightarrow \sum_{i=1}^{k} \left(d[v_{i+1}] - d[v_i] - w(v_i, v_{i+1})\right) > 0$$

but then $\exists$ $d[v_{j+1}] - d[v_j] - w(v_j, v_{j+1}) > 0$

$\Rightarrow$ (v_j, v_{j+1}) is relaxable on the last pass

finding negative-weight cycles can be done by keeping track of parent pointers.

to find all v s.t. $\delta(v) = -\infty$, just find all vertices reachable from marked vertices

DAG Relaxation

1) topological ordering.

2) relax edges in the order of vertices (increasing) in topological ordering

$\Rightarrow O(|V|+|E|)$ since no cycles.

LECTURE 15 11/2/23 11AM

All Pairs Shortest Paths - input G

size
$\Theta(V^2)$
↝ best case would make runtime $O(V^2)$

output
- $d[u,v]$: sp. distance
- parent $[u,v]$ = parent of v along s.p. $u \to v$.

naive

Dijkstra - $O(V^2 \log V + EV)$ run V times / 1 per vertex

BF - $O(V^2 E)$ run V times / 1 per vertex

bounding

$V-1 \le E \le \binom{V}{2}$

sparse ~ $O(V)$
dense ~ $O(V^2)$

⇒

	Sparse	Dense
$w \ge 0$	$O(V^2 \log V)$	$O(V^3)$
$\exists w < 0$	$O(V^3)$	$O(V^4)$

← Goal of today: Match Dijkstra runtimes even for negative weights

Adjustments to Graph

- adding weight w to every edge ⇒ BAD
- add potential function $\phi(v)$ to each outgoing edge
  subtract same amount from each ingoing edge.
  ↳ new cost of path telescopes to
  for $p(s,t)$ $\quad cost(p') = cost(p) + \phi(s) - \phi(t)$.
  ↑ if this was the minimum before reweighting, it should still be minimal
- so we want a choice of h s.t.

$$w'(u,v) = w(u,v) + h(u) - h(v) \ge 0$$

$\longrightarrow$ $w(u,v) + h(u) \geq h(v)$.

but this looks like triangle inequality?

so if we can get a vertex a connected to all vertices v, just set $h(u) = \delta(a,u)$

↳ but we need a to be able to reach every other vertex

↳ add new node a with $w(a,v) = 0 \quad \forall v \in V$.
(can compute h now with BF)

- dealing with negative-weight cycles:
  1. Find all negative-weight cycles
  2. delete them and proceed as usual.

Johnson

1) Compute SCCs of G with Kosaraju-Sharir $O(V+E)$
   - for each SCC, check using BF from a for one vertex in the SCC if $\exists$ negative-weight cycle $O(EV)$
   - if yes, add SCC_i to the list of negative cycles

2) now consider the DAG condensation graph H
   - for an edge between SCC A and SCC B, $w(A,B) = -1$ if either is in list of negative cycles. else $w(A,B) = 0$. $O(E)$
   - compute DAG all-pairs shortest paths on H, $O(V^2+VE)$
   - delete all SCCs after setting $O(V+E)$

$$\delta(u,v) = \begin{cases} -\infty & \text{if } \delta(A,B) < 0 \\ +\infty & \text{if } \delta(A,B) = +\infty \\ \text{otherwise continue} \end{cases}$$

u in SCC a
v in SCC b

3) use $\delta(a, v)$ as $h(v)$, reweight what remains of G into G' $O(V+E)$

4) run Dijkstra's from each $v \in V$, compute total distances $O(V^2 \log V + VE)$

$O(V^2)$

$\Rightarrow$ total $\boxed{O(V^2 \log V + VE)}$

no $O(V^{3-o(1)})$ solution known.

LECTURE 16 11/7/23 11AM

Row Coin Problem

coins in a row with values $C_0, C_1, \ldots, C_{n-1}$.

maximize value w/o picking consecutive coins

- brute force: all 2^n choices $\rightarrow O(2^n)$ bad
- recursion: $answer(A) = \max(answer(A[1:]), A[0] + answer(A[2:]))$
  - ↳ $T(n) = T(n-1) + T(n-2) + \Theta(1) \geq 2T(n-2)$
    $\Rightarrow T(n) = \Omega(2^{n/2})$ bad
- pure DP: $O(n)$

```
def coins(A):
    n = len(A)
    T = [0] * n
    T[n-1] = A[n-1]
    T[n-2] = max(A[n-2], T[n-1])
    for i in n-3, n-2, ... 0:
        T[i] = max(T[i+1], A[i] + T[i+2])
    return T[0]
```

SRBTOT argument

S (subproblems) - define memoization table T.

T[i] = maximum value we can pick from A[i:n]

R (recursion) - relationship between entries of T

$T[i] = \max\{T(i+1), A[i] + T[i+2]\}$ for $i = \{0, \ldots, n-3\}$

B (base case) - define base cases

$T[n-1] = A[n-1]$, $T[n-2] = \max\{A[n-2], A[n-1]\}$

T (topological sort) - show we have a DAG structure so no cycles.

T[i] only depends on T[i'] for $i' \geq i$ ⇒ fill T from right to left

O (output) - what we want

T[0]

T (runTime) - usually size of table × work done per entry

n entries, O(1) each ⇒ O(n).

Solution Recovery

for A = [5, 1, 2, 10, 6, 2]

T = | 17 | 13 | 12 | 12 | 6 | 2 |

booleans M = | T | T | F | T | T | T |

solution S = | ✓ | X | X | ✓ | X | ✓ |

True False True True

if $T[i+1] > T[i+2] + A[i]$ pick False otherwise pick True.

skip one if True, go to next if False

OSSP- Optimal Substructure Property

- optimal solutions contain optimal solutions to sub-problems

↳ guarantees that we can use recursion ⇝ DP.

Row Coin - ① any optimal solution w/ C_0 includes optimal solution for $[C_2, C_3, \ldots C_{n-1}]$

② any optimal solution w/o C_0 includes optimal solution for $[C_1, \ldots, C_{n-1}]$

LECTURE 17 11/9/23 11AM

Longest Common Subsequence

given two strings A and B length n and m, find longest substring's length (not necessarily contiguous) that appears in both

- 3 cases: ① if A[0] = B[0], include and solve for A[1:], B[1:]
  ② solve A[1:], B[0:]
  ③ solve A[0:], B[1:].

S table T[i][j] = LCS of A[i:], B[j:]

R $T[i][j] = \max \begin{cases} 1 + T[i+1][j+1] & \text{if } A[i]=B[j] \\ T[i+1][j] \\ T[i][j+1] \end{cases}$

B T[n][j]=0, T[i][m]=0 ∀ i,j

T only rely on entries where i+j ≥ i+j+1.

O T[0][0]

T O(1) to compute each entry, O(nm) entries ⇒ O(mn)

Longest Increasing Subsequence

input: A[0:n] of positive integers ⇝ can use piles from Midterm 1 to get in O(n log n).

GOAL: length of LIS.

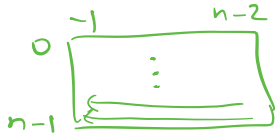
use LIS(A, last) = longest increasing subsequence of A s.t. each entry is > last → A[last]

let's use indices of A instead to keep table bounded!

LIS(A, -∞) is our solution

S table $A[i][j] \to LIS[i:]$ with entries $> A[j]$ $\quad 0 \le i \le n-1$, $-1 \le j \le n-2$

R $A[i][j] = \max \begin{cases} A[i+1][i] + 1 & \text{if } A[i] > A[j] \\ A[i+1][j] \end{cases}$



B $A[n-1][j] = \begin{cases} 1 & \text{if } A[n-1] > A[j] \\ 0 & \text{otherwise} \end{cases}$

T $A[i][j]$ depends only on $A[i'][j']$ for $i' > i \to$ go in rows upward

O $A[0][-1]$

T $O(1)$ time per $O(n^2)$ entries $\Rightarrow O(n^2)$

LECTURE 18 11/16/23 11AM

LIFE is a DAG

Single Source Shortest Path ~ optimal subproblem Property.

so DAG relaxation is

S $T(u)$ = shortest path length from s to u.

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

B $T(s) = 0$, $T(v) = \infty$ for all other $v \in V$

T topological ordering of DAG

O $T(t)$. $\forall t \in V$.

T $O(E)$ time to process all edges, $O(V)$ for nodes $\Rightarrow$ $O(V+E)$.

Bellman-Ford

S $T(v,k)$ = shortest path (length) from s to v using at most k edges.

R $T(v,k) = \min(T(u,k-1) + w(u,v))$ for $v \in Adj^-(v)$ or $T(v,k-1)$

B $T(s,0) = 0$, $T(v,0) = \infty$ for all other $v \in V$.

T topological since k is increasing

O $T(v,|V|)$

T k from 0 to $|V|$, for each $v \in V$ we process $\sum = |E|$ $Adj^-(v)$ once per $k \Rightarrow O(|V||E|)$.

Floyd-Warshall

S $T(u,v,k)$ = shortest path from u to v with only vertices in $\{1,2,\dots k\}$.

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

B $T(u,v,0) = \begin{cases} 0 & \text{if } u=v \\ w(u,v) & \text{if } (u,v)\in E \\ \infty & \text{otherwise} \end{cases}$

T k from 0 to $|V|=n$.

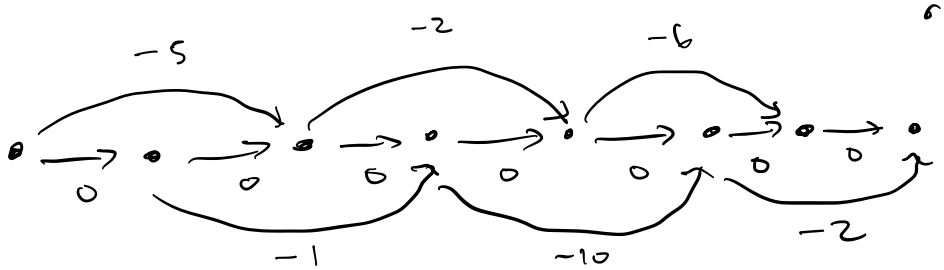
O $T(u,v,n)$ $\forall$ pairs $(u,v) \in V \times V$

T $O(1)$ work per entry $\rightarrow O(|V|^3)$ $\rightarrow$ $T(u,u,n)<0$ $\Rightarrow$ negative weight cycle.

Row Coin © as a Graph problem (shortest path)

5, 1, 2, 10, 6, 2

↳ -5 -1 -2 -10 -6 -2

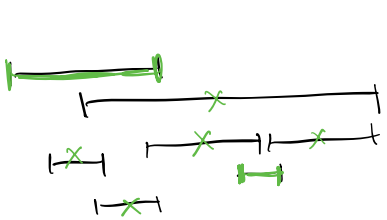


run DAG relaxation $\Rightarrow O(|V|+|E|) = O(n)$

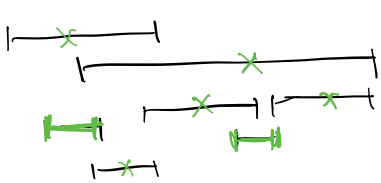
LECTURE 19 11/21/23 11AM

Activity Selection Problem

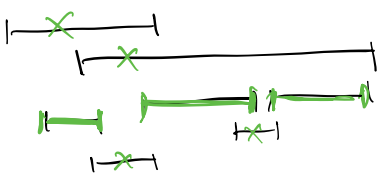
given list of activities with (s_i, f_i) start and finish times, pick subset of maximum cardinality w/o overlaps



earliest start ✗



shortest duration ✗



earliest end ✓

DP Solution

- table based on start times, recursion $T(i) = \max \begin{cases} T(i+1) \\ T(j)+1 \end{cases}$ for next $j >$ finish time of i

$\Rightarrow$ $O(n)$ work $\cdot$ $O(n)$ entries $\Rightarrow O(n^2)$ ← $O(n \log n)$ if binary search

Greedy Choice Property

- there is an optimal solution containing our greedy choice.

Proof of earliest choice satisfies GCP:

- suppose we have arbitrary optimal S with first ending activity $a' \neq a$ ← activity with earliest start time overall.
- swapping a' for a is still valid as a doesn't overlap any other activities in S. □

Correctness of (pick by earliest end time)

for any optimal solution S, WTS $|G| \geq |S|$.
(↑ greedy solution)

Suppose $G = \{a, g_2, g_3, \dots, g_k\}$, and pick S to contain a

by GCP $S = \{a, s_2, s_3, \dots, s_\ell\}$. $\Rightarrow$ $k \leq \ell$

by strong induction $G \setminus \{a\}$ is optimal for subproblem A' without activity a.

so must $S \setminus \{a\}$ as it is optimal.

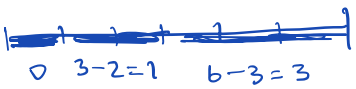
so $k-1 \geq \ell-1 \Rightarrow k \geq \ell \Rightarrow k = \ell$. □

Runtime: $O(n \log n)$ to sort, $O(n)$ to pick $\Rightarrow \boxed{O(n \log n)}$

Activity Scheduling Problem

- list of (t_i, d_i) (↑ duration, ↑ due time), one activity at a time
- schedule in a way as to minimize $\max(\text{lateness}) = \max(0, t + t_i - d_i)$ (↑ start time of i)

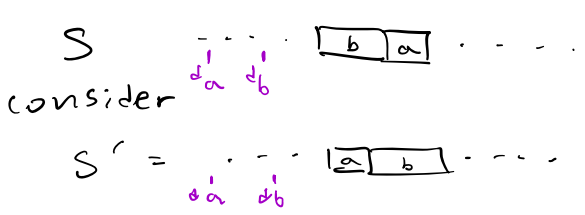
ex) (1, 1) (2, 2) (3, 3)



sort by duration is bad: ex) (1, 10000), (2, 2).

Sort by due time (also note it's optimal to have all activities consecutive)

Suppose we have optimal S with earliest due date activity a. If it's not at the front, suppose b is before it.



WTS $\max_S \ell(x) \geq \max_{S'} \ell(x)$

- b cannot maximize $\ell(x)$ in S because otherwise $d_b < d_a$.
- Swapping a and b cannot raise lateness by more than current max since $[\ell(a) \text{ in } S] \geq [\ell(b), \ell(a) \text{ in } S']$ □

Strong induct for proof of correctness, runtime $O(n \log n)$ again.

LECTURE 20 11/28/23 11AM

Huffman Codes

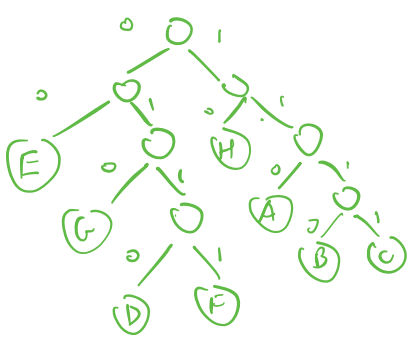
alphabet $\Sigma = \{A, B, \ldots\}$ $|\Sigma| = 64$

goal: represent each letter as a bit string

prefix-free — no string is a prefix of another

↳ else, we get ambiguity e.g. A = 000, B = 0001, C = 1000, D = 11 → 000 | 1000 = ADA or BC.

Tree



E = 00
G = 010
D = 0110
F = 0111
H = 10
A = 110
B = 1100
C = 1111

no leaf (letter) is under another letter, so there are no prefixes that are codes.

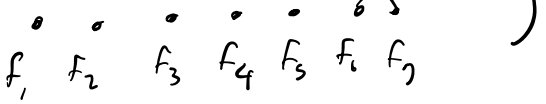
Problem Formulation:

Given T for an alphabet Σ, given $\sigma \in \Sigma$,

define f_σ = frequency of σ in string, d_σ = depth of σ in T.

Minimize $\sum_{\sigma \in \Sigma} d_\sigma f_\sigma = L(T)$ = length of encoded message.

↳ build a tree to make this happen.



since each is counted as much as their depths

$$L(T) = \sum_{\substack{\text{nodes} \\ (\text{not root})}} f_{\text{node}}$$

Plan

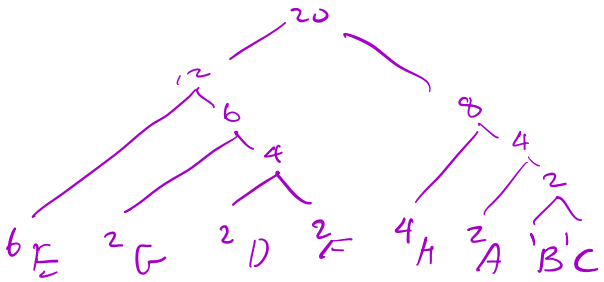
combine into a tree.

① combine two least frequent first ✓

② combine two most frequent first ✗

Algorithm (Greedy)

merge two least-frequent nodes at each step.



runtime (min heap to sort)

$O(n \log n)$ for n searches.

Greedy Choice Property

∃ optimal tree T

s.t. the two least-frequent nodes are siblings.

two observations

01 Any optimal tree is complete (otherwise just collapse: )

02 $f_g \leq f_h \Rightarrow d_g \geq d_h$ (rearrangement inequality)

Proof of GCP

exist by 01

let f_x, f_y be least frequent, let f_a, f_b be siblings of greatest depth.

we can use 02 to swap $f_a \to f_x$, $f_b \to f_y$ to make f_x, f_y siblings of greatest depth.

Proof of Greedy Algorithm

Greedy Choice Property + Induction

Let $f_x \le f_y \le \cdots$

$f_z = f_x + f_y$

Let H be tree provided by GREEDY.
Let T be an optimal tree with f_x, f_y as siblings (GCP).

H' = H minus f_x, f_y.

T' = T minus f_x, f_y.

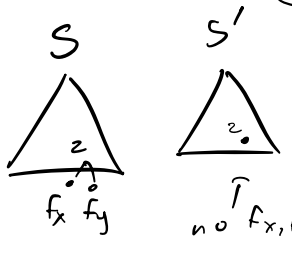
by induction $L(H') \le L(T')$

$+f_x+f_y$ $+f_x+f_y$

$\Downarrow$

$L(H) \le L(T)$ □

for



$$L(S) = \sum_{\text{not } x,y} f_c d_c + f_x d_x + f_y d_y$$

$$L(S') = \sum_{\text{not } z} f_c d_c + f_z d_z$$

$\downarrow$

$d_z + 1 = d_x = d_y$

$f_x d_x + f_y d_y = (f_x + f_y)(d_z + 1)$

↳ $L(S) = L(S') + f_x + f_y$

LECTURE 21 11/30/23 11AM

Question rod of length L, $v[\ell]$ for $1 \le \ell \le L$ has values.

cut rod up so $\ell_1 + \ell_2 + \cdots + \ell_k = L$, maximize $\sum v[\ell_i]$.

greedy - take maximum $\frac{v[\ell]}{\ell}$ first, then continue.

↳ no. take $[1, 10, 13, 18, 20, 31, 32]$ for $L = 7$

↳ max $\frac{v[\ell]}{\ell} = \frac{31}{6}$, so do $L = 1 + 6 \Rightarrow v[1] + v[6] = 32$.

but $v[2] + v[2] + v[3] = 33 > 32 \rightarrow$ greedy FAILS.

Dynamic Programming

S $T[\ell]$ = best case for len ℓ rod.

R $T[\ell] = \max_{i \in [\ell]} (v[i] + T[\ell - i])$

B $T[0] = 0$

T increasing ℓ

O $T[L]$

T $O(n)$ per problem, $O(n)$ problems $\Rightarrow O(n^2)$.

is this efficient?

i.e.) polynomial in # of words in input? → one $L \Rightarrow 1$, L indices $\Rightarrow L$ } $O(L)$ so yes, polynomial.

Subset-Sum

given an array $A = [a_1, a_2, \ldots, a_k]$, does there exist a subset summing to given L?

Dynamic Programming

S $T[i, \ell]$ = ∃ subset in $A[i:]$ with sum ℓ? (i: $1 \sim n$, ℓ: $0 \sim L$)

R $T[i, \ell] = T[i+1, \ell]$ OR $T[i+1, \ell - A[i]]$ and $\ell \geq A[i]$

B $T[i, 0] =$ True, $T[n, k] =$ True iff $a_n = k$

T decreasing indices i

O $T[1, L]$

T $O(nL)$ entries, $O(1)/\text{entry} \Rightarrow O(nL)$

But $O(nL)$ is not polynomial.

↳ L can be at most 2^w in w-bit wordRAM, which isn't polynomial!

⇒ pseudo polynomial: ONLY polynomial in input size / parameters

ex) Counting Sort $O(n+u)$ $n \leq 2^w$ in general

ex) DAA Set building $O(n+u)$ can contain elements in $\{0, 1, \ldots, u\}$

LECTURE 22 12/5/23 (Did not attend) + LECTURE 23 12/7/23 11AM

Complexity Theory

big search spaces (exponentially large)

↳ Greedy / DP lets us find in poly time

decision problems: YES/NO

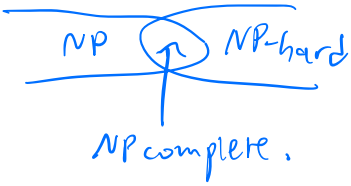
optimization, search, functional computation can all be reduced to this

$P \subseteq NP \rightsquigarrow$ certificate verification in poly time

$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$

ex) 3SAT

3-colorability



$A \leq_p B$ if ∃ polytime algorithm that, given an instance x of A, can output an instance y of B with same answer.

LECTURE 24 12/12/23 11 AM

decision problem - problem where for every instance x the answer is YES(1) or NO(0)

$F: \{0,1\}^* \to \{0,1\}$

$\{0,1\}^*$: all binary strings

19th century

formalize all math
- list of axioms
- Hope: given a proposition, mechanical way to determine if it's true or not with axioms

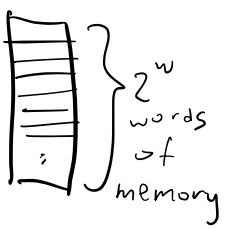
Entscheidungsproblem (decision problem)

Church 1935-36, Turing 1936-37 → No algorithm to solve this problem. ↔ related: Godel's incompleteness theorems

today - rough definition of algorithm
- proof that there is no alg. to solve Halting problem

Algorithm - A piece of code that runs on an idealized word-RAM machine

whenever we run out of memory, increase word size by 1



running program M (on input x)
- returns YES
- returns NO
- runs forever

M solves problem F if $\forall$ inputs $x \in \{0,1\}^*$, M halts and $M(x) = F(x)$ (no time bound as long as it's finite)

↳ F is "decidable" if $\exists$ M that solves F

↳ F is "undecidable" otherwise

recall $|\mathbb{R}| > |\mathbb{N}|$ by Cantor Diagonalization.

Claim $\exists$ F undecidable

note
algorithms are strings $\Leftrightarrow \mathbb{N}$

for Algorithm M

$$G(M,x) = \begin{cases} 1 & \text{if } M(x)=1 \\ 0 & \text{if } M(x)=0 \text{ or doesn't stop} \end{cases}$$

M	$G(M,x)$ on all inputs x_i
	x_1 x_2 x_3 ...
M_1	0 1 0 1 1 0 0
M_2	1 0 1 1 1 1 1 0 ...
M_3	0 1 1 0 0 1 1 0
⋮	⋮ ⋱

take y s.t. $\Rightarrow$ y this isn't in the table

y_i = opposite of i^{th} bit of i^{th} row

$\Downarrow$

$\nexists$ algorithm M that solves F

$F(x_i) := y_i$

Thm Halting problem is undecidable

Pf Suppose $\exists$ algorithm A that decides it.

```
def confuse(y):
    val = A(y,y)
    if val = True:
        infinite loop.
    else:
        Halt.
```

Claim A is incorrect on input (confuse, confuse)

1) Suppose A(confuse, confuse) = True.
   ↳ confuse(confuse) Halts
   ↳ A(confuse, confuse) = True → infinite loop → Bad.

2) Suppose A(confuse, confuse) = False
   ↳ confuse(confuse) must not halt
   ↳ but A(confuse, confuse) = False so halt → Bad.