sean mintz
mintzsea@mit.edu

9/5/25 REC 1:

ASYMTOTICS: O, Ω, Θ

$f \in O(g) \rightarrow f(n) \leq c \cdot g(n) \quad \forall n > n_0$ g will be upper bound of f

$f \in \Omega(g) \rightarrow f(n) \geq c \cdot g(n) \quad \forall n > n_0$

RECURRENCES:

- master's thm
- tree method
- substitution

LEC REVIEW:

- $O(n^2)$ multiply every pair of digits

```
 342
 483
-----
1026
  60
```

KARATSUBA D&C:

$\underset{x_{hi}}{324} | \underset{x_{lo}}{876}$ $\quad$ $\underset{y_{hi}}{289} | \underset{y_{lo}}{735}$

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

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

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

$E = \text{mult}(x_{lo} + x_{hi}, y_{lo} + y_{hi})$

Recitation 1

Lecture Summary

- Algorithms, correctness, and efficiency
- Preschool versus grade school algorithms for addition
- Grade school versus Karatsuba's algorithms for multiplication

Exercises: Review of big-O, Karatsuba's Algorithm

1. Getting a feel for common recurrences:

   (a) Prove that $T(n) \leq T(n/2) + O(n)$ is solved by $T(n) = O(n)$, using the following steps. (i) Draw a recurrence tree representing this recurrence relation. Analyze the total amount contribution at each level and the number of levels of the tree (depth). (ii) Prove that the sum $\sum_{\ell \leq depth}$ (total from level ℓ) $= O(n)$, substituting the expression you got in part (i) for "total from level ℓ". (iii) Check the correctness of your solution using the Master Theorem.

   (b) The runtime of the optimal sorting algorithm mergesort (covered later in the course) satisfies the recurrence $T(n) \leq 2T(n/2) + O(n)$ on lists of size n. Perform the same analysis as in part (a) for this recurrence. Optionally, solve the recurrence using the substitution method.

   (c) Recall from lecture that the naive divide-and-conquer algorithm for multiplication satisfies the recurrence $T(n) \leq 4T(n/2) + O(n)$. Perform the same analysis as in part (a) for this recurrence.

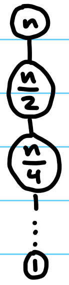
2. See the lecture 1 notes for the definition of Karatsuba's algorithm. Recall that the number of basic operations it uses to multiply two n-digit numbers, $T(n)$, follows the recurrence relation $T(n) = 3T(n/2) + Cn$, for some constant C. Use the recursion tree method to prove that $T(n) = O(n^{\log_2 3})$.

3. In this class we will frequently prove correctness of algorithms via induction. Early in the course, these proofs might seem unnecessary, because we will start by studying (relatively) simple algorithms whose correctness is easy to see. However, it is important to get practice with these inductive correctness proofs, because, as the course progresses, we will see more and more sophisticated algorithms, whose correctness is far from obvious. We will convince ourselves of their correctness via proofs.

1) a) $T(n) \le \underset{\text{bounded by}}{\underline{T(n/2)}} + O(n)$

$T(\frac{n}{4}) + O(\frac{n}{2})$

recurrence tree:



$T(\frac{n}{4}) + O(\frac{n}{2})$

$T(\frac{n}{8}) + O(\frac{n}{4})$

#levels = $\log_2 n$

$1 = \frac{n}{2^i}$

#nodes @ i = 1

(no branching)

work per node @ i = $O(\frac{n}{2^i})$

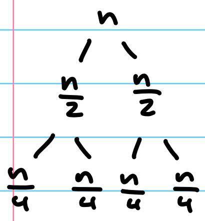
$$\sum_{i=0}^{\log_2 n} 1\cdot c\frac{n}{2^i} = c\cdot n\sum\frac{1}{2^i} = cn\frac{1-(\frac{1}{2})^{\log_2 n+1}}{1-\frac{1}{2}} = cn \quad O(n)$$

masters thm: CASE 3 $c > \log_b a$, $O(n^c)$

work @ top dominates recurrence

b) $T(n) \le \overset{a}{2}T(\overset{b}{n/2}) + O(n)$

$T(\frac{n}{2}) = 2(2T(n/4) + O(n/2))$



#levels: $\log_2 n$

nodes @ level i: 2^i

work per node @ level i: $O(\frac{n}{2^i})$

levels

$$\sum_{i=0}^{\log_2 n} \underset{\text{nodes @ level}}{2^i}\cdot \underset{\text{work @ level}}{c\frac{n}{2^i}} = cn\sum_{i=0}^{\log_2 n}1 = cn(\log_2 n+1)$$

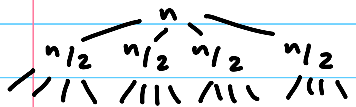
$O(n\log n)$

c) $T(n) \le \overset{a}{4}T(\overset{b}{n/2}) + O(n)$

$T(\frac{n}{2}) = 4(4T(n/4) + O(n/2)) + O(n)$

a= how much splitting

b= work per level



#levels: $\log_2 n$

★ work incr. per level ??

nodes @ level i: 4^i

work per node @ level i: $O(\frac{n}{\underset{b}{2^i}})$

not $O(\frac{n}{4^i})$

$$\sum_{i=0}^{\log_2 n} c\cdot\frac{n}{2^i}\,4^i = cn\sum_{i=0}^{\log_2 n}2^i$$

$$= cn2^{\log_2 n}\underbrace{\sum_{i=0}^{\log_2 n}2^{-i}}_{\le 2} = 2cn\cdot n \to O(n^2)$$

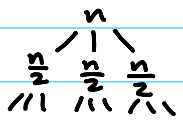
$1+\frac{1}{2}+\frac{1}{4}+\ldots \to n$

case 1 MT

$c < \log_b a$

2) $T(n) = \overset{a}{3}\, T(\overset{b}{\tfrac{n}{2}}) + Cn$ prove $T(n) = O(n^{\log_2 3})$

$T(\frac{n}{2}) = 3\left(3T\left(\frac{n}{4}\right) + c\left(\frac{n}{2}\right)\right) + Cn$



levels: $\log_2 n$

nodes @ level i: 3^i

work @ level i: $O\left(\frac{n}{2^i}\right)$

$$\sum_{i=0}^{\log_2 n} 3^i \cdot c\,\frac{n}{2^i} = cn \sum_{i=0}^{\log_2 n} \left(\frac{3}{2}\right)^i$$

← geom. series depends on ratio more than biggest term

$$= cn \cdot O\left(\left(\frac{3}{2}\right)^{\log_2 n}\right)$$

$$= cn \cdot O\left(\frac{3^{\log_2 n}}{n}\right)$$

$$= O\left(3^{\log_2 n}\right)$$

With this in mind, prove by induction on n that Karatsuba's algorithm run on two n-digit integers x, y always returns $x \cdot y$. The proof is relatively straightforward, but has a tricky case which needs to be handled carefully.

- We ignored the base case of the recursion when discussing Karatsuba's algorithm in class. Describe how the algorithm might handle this base case. For this analysis, feel free to modify the algorithm by choosing the size of the base case to be any constant.
- Your proof should have a base case and an inductive case, corresponding to the recursive and base cases of Karatsuba's algorithm.

4. **(optional)** Toom-Cook multiplication is a generalization of the Karatsuba algorithm for multiplication, which can achieve even better asymptotic results. The runtime for Toom-Cook multiplication satisfies $T(n) = (2k-1)T(n/k) + O(n)$, where k is a positive integer parameter of the algorithm (Karatsuba multiplication corresponds to $k = 2$). Use a recurrence tree to solve for the asymptotic runtime of Toom-Cook multiplication. Your answer should involve k as a parameter.

still a const. size (not based on n)

3) Pf by Strong Ind: P(n) = Karatsuba's works

table

need to pick large enough BC

BC: $n \leq 3$: keep a table for 3 digit products, $(10^3)^2 \in O(1)$

might have carryover

IND. STEP: assume correct for $\leq n-1$. show true for n.

makes recursive calls for # at most $\frac{n}{2}+1$ digits.

adding back together is correct ⌣̈

*need to show calling to smaller things

9/10:

ALGORITHM: procedure to solve computational problem

EFFICIENCY: analyze w/ worst case lower & upper bounds

Ω - pick one input, this gives lower bound

O - do some analysis ab. alg. for any input

WORD-RAM: can do arithmetic & follow pointers in $O(1)$

EX: peak finding w/ binary search

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

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

Introduction to Algorithms: 6.1210
Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson — Recitation 2

Recitation 2

Lecture Summary

- Algorithms
- Efficiency and Worst-Case Analysis
- Model of Computation
- Peak Finding and Binary Search

Exercises: Algorithms, efficiency and worst-case analysis

Problem: Given the students in your recitation, return either the names of two students who share the same birthday and year, or state that no such pair exists.

Algorithm: Choose one student at a time and ask for their birthday. Then ask everyone else in the room whether they have the same birthday, and if a match is found, return it. Otherwise, send the chosen student out of the room, and run the same algorithm on the remaining students. If there are no more students in the room, return None.

★ 1. Prove that the birthday algorithm is correct by induction.

2. Provide an upper bound on the worst-case runtime for the birthday algorithm. Use the implementation below as a guide:

```
def birthday_match(students):
    '''
    Find a pair of students with the same birthday
    Input:  list of student (name, bday) tuples such as
    (("a", "8-28-1999"), ("b", "10-25-2003"))
    Output: tuple of student names or None
    '''
    # Base case
    if len(students) == 0:   O(1)
        return None
    (name1, bday1) = students[0]
    for i in range(1,len(students)):  O(n)
        # Check if students are the same
        (name2, bday2) = students[i]
        if bday1 == bday2: O(1)
            return (name1, name2)
    return birthday_match(students[1:])  O(n)
```

① P(n) = the birthday alg. works

THM: P(n) for all n > 1

B.C.: P(0) = none

P(1) = none

INDUCTIVE STEP: PROOF BY <u>CASES</u>:

assume true for n-1.

CASE 1: person n in a pair, then alg. will be correct b/c alg. loops over everyone

CASE 2: person n not in a pair. then, person n will be excluded and alg. will recurse over n-1, which is assumed to be true.

② *write as recurrence for recursion:

upper: $T(n) = T(n-1) + O(n)$ linear recurrence

master's thm is only for $\frac{n}{\#}$

work done:

$O(n)$ • n

$O(n-1)$ • $n-1$ $\quad O\left(\sum_{i=1}^{n} i\right) = O(n^2)$ → total work = $n + (n-1) + \ldots + 1 = \frac{n(n+1)}{2}$

$O(n-2)$ • $n-2$

⋮

could differ by a constant

③ lower: bad input: no pair → $\Omega(n^2)$ ← any input can be a lower bound, but we want the "tightest" lower bound, so input the worst case

3. Provide a lower bound on the worst-case runtime for the birthday algorithm.

Exercises: Binary search

4. Binary search is an efficient algorithm for finding an element in a sorted array. Consider an array with n unique numbers that has been sorted in ascending order and then rotated an unknown number of positions k such that the index of each number in terms of its original index i is $i + k \mod n$.

   E.g. below is an example rotated array of length 11 which is rotated $k = 6$ positions, compared to the original sorted array.

```
Index:              0    1    2    3    4    5    6    7    8    9   10
rotated array:  [   7,   8,  10,  12,  13,  15,   1,   2,   3,   4,   6 ]
```

$k = 6$

```
sorted array:   [   1,   2,   3,   4,   6,   7,   8,  10, ~~13,~~ 12  13,  15 ]
```

   (a) Design an $O(\log n)$ algorithm to find the value of k. Don't forget to prove correctness and argue runtime.

   (b) Design an $O(\log n)$ algorithm to determine if a number x is in the array. Again, remember to prove correctness and argue runtime.

5. Given a positive integer n, return $\lfloor \sqrt{n} \rfloor$. The algorithm should run in time $O(\log n)$.

6. Given a monotone function $f : \mathbb{Z} \rightarrow \{-1, 0, 1\}$ with exactly one root α, find α. The algorithm should run in time $O(\log |\alpha|)$. (Assume that f can be evaluated in constant time).

* recursion is typically
pf. by strong ind. by cases!

fails for: 4 5 6 7 8 1 2 3

<R, >L all good

X 1 2 3 4 5

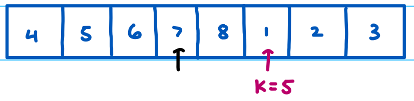
<L, <R → go R

4 5 1 2 3

← >L, >R, go L

3 4 5 1 2

4 a)



* looking for discontinuity

alg: compare A[i] to A[-1].

if A[i] > A[-1], recurse right

otherwise, recurse left

base case: size ≤ 2

Correctness: pf by STRONG IND

base case: ...

inductive step: pf by CASES.

1) if A[i] > A[-1], we know right side unsorted. problem of size n/2 solved by n/2

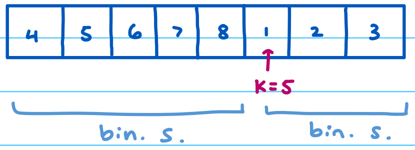
2) otherwise, left unsorted. problem of size n/2 solved by n/2

runtime: $T(n) = T(\frac{n}{2}) + O(1)$

$O(\log n)$ ü

b) run part a to find k → O(logn) ← already proved in part a "black box" (can just use)

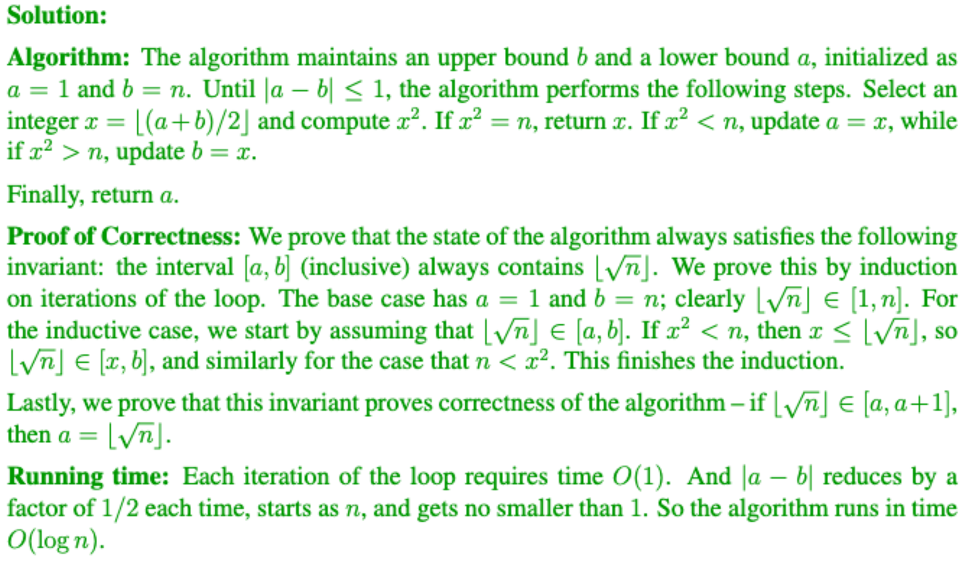
after finding k, do 2 seperate binary searches



@ most 2 binary searches on arrays of size ≤ n

runtime = 2·O(logn) → O(logn)

5



Solution:

Algorithm: The algorithm maintains an upper bound b and a lower bound a, initialized as $a = 1$ and $b = n$. Until $|a - b| \leq 1$, the algorithm performs the following steps. Select an integer $x = \lfloor (a+b)/2 \rfloor$ and compute x^2. If $x^2 = n$, return x. If $x^2 < n$, update $a = x$, while if $x^2 > n$, update $b = x$.

Finally, return a.

Proof of Correctness: We prove that the state of the algorithm always satisfies the following invariant: the interval $[a, b]$ (inclusive) always contains $\lfloor \sqrt{n} \rfloor$. We prove this by induction on iterations of the loop. The base case has $a = 1$ and $b = n$; clearly $\lfloor \sqrt{n} \rfloor \in [1, n]$. For the inductive case, we start by assuming that $\lfloor \sqrt{n} \rfloor \in [a, b]$. If $x^2 < n$, then $x \leq \lfloor \sqrt{n} \rfloor$, so $\lfloor \sqrt{n} \rfloor \in [x, b]$, and similarly for the case that $n < x^2$. This finishes the induction.

Lastly, we prove that this invariant proves correctness of the algorithm – if $\lfloor \sqrt{n} \rfloor \in [a, a+1]$, then $a = \lfloor \sqrt{n} \rfloor$.

Running time: Each iteration of the loop requires time $O(1)$. And $|a - b|$ reduces by a factor of $1/2$ each time, starts as n, and gets no smaller than 1. So the algorithm runs in time $O(\log n)$.

★ 6

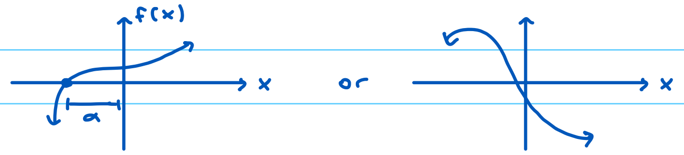
Solution:

Algorithm: This is a variant of Binary Search called *Exponential Search*. Initialize $B = 1$, then double B repeatedly until $f(-B) \leq 0 \leq f(B)$. Binary Search the range $[-B, B]$.

Proof of Correctness: For all $B \geq |\alpha|$, we have $-B \leq \alpha \leq B$, and by monotonicity $f(-B) \leq f(\alpha) = 0 \leq B$. B is a strictly increasing integer, and there are only finitely many B for which the loop guard fails, so the first step will eventually terminate. At this point we have $f(-B) \leq 0 \leq f(B)$. Again by monotonicity and uniqueness of α, this gives $-B \leq \alpha \leq B$. Correctness now follows from Binary Search.

Running time: Each iteration of the loop takes constant time. B increases by a factor of 2 each time, starts as 1, and ends between $|\alpha|$ and $2|\alpha|$. Therefore the first loop runs in time $\Theta(\log|\alpha|)$. Binary Search on a range of length $\leq 4|\alpha|$ takes $O(\log|\alpha|)$ time, for a total of $\Theta(\log|\alpha|)$.

$f: \mathbb{Z} \to \{-1, 0, 1\}$ ← one direction +



soln: log(value of root)

binary search to find where the "flip" if

go from small → big

O(log|α|) to find root,

O(log|α|) to find

9/12 DATA STRUCTURES: stores / manages data

INTERFACE: what operation it supports

IMPLEMENTATION: how its represented, what algorithms are used for, the operations

INTERFACES so far:

SEQUENCE: ordered collection of items w/ indices

- STACK: subset of sequence w/ LIFO operators
- QUEUE: subset of sequence w/ FIFO operators

HOW TO IMPLEMENT SEQUENCE:

- linked list
- array → dynamic array, expands / shrinks when needed (only happens when doubles)
  - average a series of expensive operations over a series of cheap ones

AMORTIZED ANALYSIS:

- operations have amortized cost T if any sequence of m operations (starting from empty data structure) takes at most mT
- to show amortized cost, we use potential functions. ϕ (data structure) → $\mathbb{R}$

  $\phi(A) \geq 0$

  for all operations from $A_i \rightarrow A_{i+1}$ that take time t, amortized cost is $t + \phi(A_{i+1}) - \phi(A_i)$

$\phi(A) = c|n - \frac{m}{2}|$ c = constant
n = # elements
m = capacity

- when doubling, ϕ goes from $c\frac{n}{2}$ to 0

  when full ↗ ↖ after adding capacity

Introduction to Algorithms: 6.1210
Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson Recitation 3

Recitation 3

Lecture Summary

- Data structures, interfaces and implementation
- Amortization
- Sequence interface
- Implementations of the Sequence interface: linked list, dynamic array

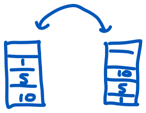
Exercises

1. Given a implementation of the Stack interface which takes $O(1)$ amortized time for each operation, show how to use it to implement the Queue interface, and analyze your implementation's runtime.

2. Modify the Dynamic Array implementation of the Sequence interface such that the operations `insert_first` and `delete_first` take $O(1)$ amortized time alongside `insert_last` and `delete_last`.

3. Suppose the next pointer of the last node of a linked list points to an earlier node in the list, creating a cycle. Given a pointer to the head of the list (without knowing its size), describe a linear-time algorithm to find the number of nodes in the cycle. Can you do this while using only constant additional space outside of the original linked list?

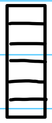
4. (Optional, hard) Suppose you have an implementation of the Queue interface which takes $O(1)$ amortized time for each operation. Show how to use it to implement the Stack interface using only black-box access to a single Queue and constant extra space. (Black-box means that you are only able to interact with the Queue using the Queue interface.) What are the best possible amortized runtimes you can achieve for the Stack operations?



* amortized b/c
stack operations
are amortized

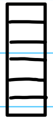
★ ①

STACK:



push(x)
pop()
peek()
isEmpty()

QUEUE:



enque(x) → push to B: O(1) am
deque() → flip if needed, pop from F: O(n) am
peek() → might also have to Flip, but $O(1)_{am}$
len() → keep a counter O(1)

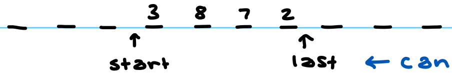
flip, O(k) am where
k = items in B, but
that means we did
k insertions, so k+1
operations w/ 2k → O(1)am

observation: 2 stacks can make a queue.

$\xrightarrow{push} B \xrightarrow{flip} F \xrightarrow{pop}$

how does it work?

② Dynamic Array: insert_last, delete_last, insert_first, delete_first



← can easily move two pointers

run out of space w/ n items in sequence

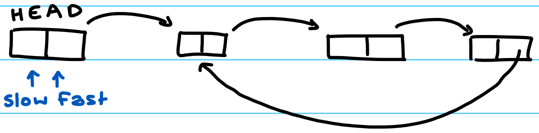
why not n/2 ?

allocate n space on both sides. Amortized

over n insertions you have to do for it

to double

③



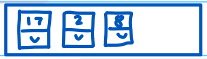
- two pointers, one moves 2 at a time, the other goes 1 at a time. (diff rates)
- wait until they meet then you know a cycle exists
- s will not complete full rot. before f catches up → can calc. runtime
- O(n)

9/17 2 IMPORTANT INTERFACES:

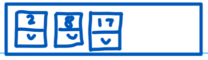
- sequence (a bunch of items in some order; a list)
- set (key-value pairs; similar to dictionary; order operations)

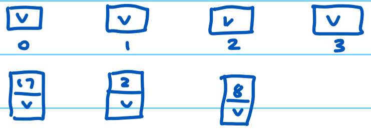
IMPLEMENTING SET:

- dynamic array



- sorted dynamic array





find(k) ∈ Θ(n)

  - build runtime = time to sort
  - find(k) takes O(logn)

SORTING:

- INSERTION SORT: O(n2)

S E A N "comparing"

E S A N

A E S N

A E N S ✱ STABLE

- SELECTION SORT O(n2) "handpicking"

S E A N ← looking for largest char each time then move on

E A N S

E A N S

A E N S ✱ NOT STABLE

A E N S

CHARACTERISTICS OF SORTING ALGS:

- in-place: O(1) auxilary memory
- stable: preserves order of elements w/ equal keys

DIVIDE & CONQUER:

1) divide into subproblems (going down)

2) solve subproblems recursively

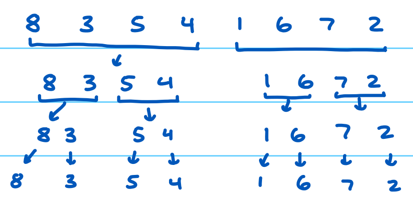
3) combine solutions into subproblems (building back up)

EX: MERGE SORT "two-finger algorithm"

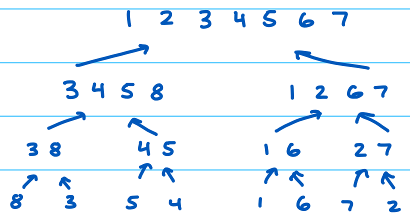
- not in-place (O(n) extra space)
- stable
- $T(n) = 2T(\frac{n}{2}) + \underset{\text{splitting initially}}{O(n)} + \underset{\text{look @ elements \& put back (two fing)}}{O(n)}$

$\Rightarrow O(n\log n)$

splitting:



combining:



Introduction to Algorithms: 6.1210
Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson — Recitation 4

Recitation 4

Lecture Summary

- Incremental algorithms, e.g. insertion sort, selection sort
- Sorting (stable, in-place)
- Divide and conquer, e.g. merge sort, 2D peak finding

Exercises

1. Describe how to use a sorted array to implement the Set interface. What is the runtime of each operation? (You can assume there is already plenty of space allocated for the array.)
2. Describe how to use a sorted doubly linked list to implement the Set interface. What is the runtime of each operation? (head & tail)
   (forward & back)
3. In a train station, trains come at certain arrival times and leave at certain departure times. Given the sorted list of all arrival times and the sorted list of all departure times, what's the minimum number of platforms the station needs to accommodate all the trains? Give an algorithm and analyze it. Assume there are 0 trains in the station to begin with.

Divide and Conquer

Inversion Counting

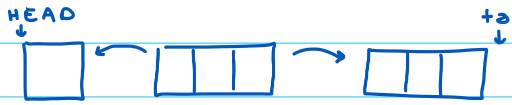
We will see how to solve this problem using the divide and conquer approach: Given an array, count the number of inversions in it. Two elements `a[i]` and `a[j]` form an inversion if `a[i]` $>$ `a[j]` but `i < j`. The inversion count of an array gives an idea of how sorted it is.

4. What is the brute force algorithm for solving this problem? What is its running time?
5. Find and analyze a more efficient algorithm for the inversion count problem.

① SET:

(x.key
x. v)

build	Θ(nlogn)	→ need to build & sort. use merge sort
find(k)	Θ(log n)	→ binary search (divide in half each time)
insert(x)	Θ (n)	→ search O(logn) where val goes then shift O(n) everything over
delete(k)	Θ (n)	→ search, delete, shift values over
iter_ord()	Θ (n)	→ sorted; walk from beginning to end
find_min()	Θ (1)	→ 1st
find_max()	Θ(1)	→ last
find_next(k)	Θ(logn)	→ search then +1
find_prev (k)	Θ (logn)	→ search then -1

② 

build	Θ (nlogn)	→ sort & build; merge sort then build linked list
find(k)	Θ (n)	→ have to walk through list
insert(x)	Θ (n)	→ walk to find position
delete(k)	Θ (n)	→ get to item is Θ(n); deleting is easy
iter_ord()	Θ (n)	→ print out in order
find_min()	Θ (1)	→ head pointer
find_max()	Θ(1)	→ tail pointer
find_next(k)	Θ(n)	→ get to value then +1
find_prev (k)	Θ (n)	→ get to value then -1

★ ③

application of merge sort & merge set

arrivals:	1	3	6	8	10
depart:	5	9	11	11	13

max count = 3

approach: keep a counter

if A[x] < D[y], increment counter & move x

if A[x] > D[y], decrement counter & move y

return highest value that counter reached.

④

6	5	2	4

5 inversions

BRUTE FORCE: look @ all inversions to the right of each element.

$\Theta(n^2)$

★ ⑤

$\underbrace{6 \quad 5}_{\text{intra invert}} \xleftrightarrow[\text{inter inversion}]{} \underbrace{2 \quad 4}_{\text{intra invert}}$

comparing i of left w/ j of right

2, 4 is inverted with everything on the left

left[i] > right[j] → inter-inversions += $\overbrace{\text{len(left) - i}}$

index

Θ(nlogn)

2 4 5 6

+1 → 5 6   +2 24 +2

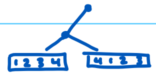
6 5 2 4

9/24 lec. 5: LINEAR SORT

· comparison models: can only differentiate models via comparisons

· sorting items n: n! possibilities

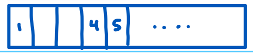
· any comparison model alg. $\Omega(\log(n!)) \in \Omega(n\log n)$



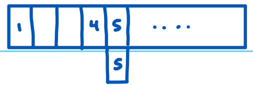
↖ lower bounded by nlogn always.

u = value of largest # of string

DAA SORT: $\Theta(u)$



COUNTING SORT: DAA sort w/ chaining

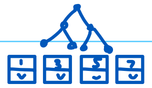


TUPLE SORT: split by place value, sort by least significant first ; stability matters

RADIX SORT: $u < n^c$, tuple sort w/ counting sort. cn time, $\Theta(n)$

implementing SETS w/ HASHING:

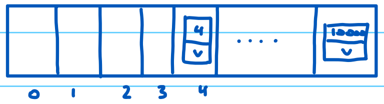
· comparison model: Find(k) lower bounded by $\Omega(\log n)$



← key-val. pair

· outside comparison model: DAA

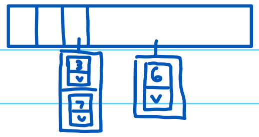
u >> n, lot of space, slow order ops



HASHING: #randomly map w/ hashing func (uniformly random)

· map to smaller space $h(k): \{0, \dots, u-1\} \rightarrow \{0, \dots, m-1\}$

· collisions → chaining



· expected analysis: SUHA (assume uniformly & random

& pairwise indep.

$Pr[h(k)=i] = \frac{1}{m}$ ← 1 key going into a certain slot $Pr = \frac{1}{m}$, m = total # spots

$Pr[h(k)=i \text{ AND } h(k')=j] = \frac{1}{m^2}$ ← 2 keys ending up in same space $\frac{1}{m} \cdot \frac{1}{m} = \frac{1}{m^2}$

· chain length $O(\alpha+1)$, $\alpha = \frac{n}{m}$ ↖ chain bounded by this, which is constant → $\Theta(1)$ ex to find → $\frac{\text{\# total items}}{\text{length of array}}$

· n gets too big, increase m → $\Theta(1)$ am. ex. have to make new table, rehash everything

→ similar to expanding dynamic array.

→ once you have to expand hash, amortized $\Theta(1)$ over all the inserts that got us there

Introduction to Algorithms: 6.1210
Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson — Recitation 5-6

Recitation 5-6

Lecture Summary

L05

- Comparison Model and Lower Bound
- Direct Access Array Sort
- Counting Sort
- Tuple Sort
- Radix Sort [1]

L06

- Comparison Find Lower Bound
- DAA Sets
- Hashing + hash functions
- Collisions and chaining

DUPLICATES

Given an unsorted array $A = [a_0, \ldots, a_{n-1}]$ containing n positive integers, the DUPLICATES problem asks whether there is an integer that appears more than once in A (the answer is 'yes' or 'no'; you don't need to return the repeated integer). All of the questions in this section should have 1-3 sentence answers.

1. Describe a (very simple) **worst-case** $O(n^2)$-time algorithm to solve DUPLICATES.
2. Describe a **worst-case** $O(n \log n)$-time algorithm to solve DUPLICATES.
3. ★ Describe an **expected** $O(n)$-time algorithm to solve DUPLICATES.
   ↳ hashing!
4. ★ If $k < n$ and $a_i \leq k$ for all i, describe a **worst-case** $O(1)$-time algorithm to solve DUPLICATES.
   n=10 items, k<10 for all items ⇒ there will be a duplicate
5. If $n \leq k$ and $a_i \leq k$ for all i, describe a **worst-case** $O(k)$-time algorithm to solve DUPLICATES.

[1] CoffeeScript Counting/Radix sort visualizer: `https://codepen.io/mit6006/pen/LgZgrd`

① for each #, loop through entire array and see if there is >1 occurence. → $\Theta(n^2)$

② merge sort array then scan (see if any adjacent pairs are same). $O(n\log n) + O(n) = O(n\log n)$

★ ③ hash all items. check if inserting a duplicate as you insert into table.

hash
expected chain size is $O(\alpha+1)$, $O(1)_{exp}$ to check each chain → $O(n)_{ex}$.

④ YES; aren't enough distinct vals.

⑤ counting sort $n \leq k$; when seeing dupe, be done.

Linear Sort

1. **True or False:** There exists a comparison sorting algorithm that sorts 5 numbers and uses at most 6 comparisons in the worst case.

2. Sort the following integers using a base-10 radix sort.

$$(329, 457, 657, 839, 436, 720, 355) \longrightarrow (329, 355, 436, 457, 657, 720, 839)$$

3. Describe a linear time algorithm to sort a set n of strings, each having k English characters.

4. Describe a linear time algorithm to sort n integers from the range $[-n^{49}, \ldots, n^{50}]$.

5. (Optional) You are tasked with writing a pest buddy app, and you initial idea is to take a list of kerbs as input, and find all *similar* kerb pairs from the list and group students up that way. We define two kerbs to be *similar* if they differ by exactly one character. Assume each student has an 8 character long kerberos and no two kerbs are the identical. You want to test your idea and decide to first design an algorithm that counts the number of all *similar* kerb pairs. Design an algorithm that counts the *similar* kerb pairs in linear time.

Hashing tuples

The hash functions we've discussed only process a single machine word. What if we want to hash something much bigger, like an entire file?

Let's hash tuples of integers of the form $(a_1, \ldots, a_t)$. For this section, assume that, for any prime m, you have a hash function h which sends a tuple of t integers to a hash in $\{0, \ldots, m-1\}$, and can be computed in $O(t)$ time. Furthermore, assume that h comes from a distribution that satisfies the Simple Uniform Hashing Assumption.[2]

Given a list of *bad* tuples $[v_1, \ldots, v_n]$, we want to construct a data structure that supports the following operation: given a new tuple v, check whether v is identical to any bad v_i. The time of this operation and the space occupied by the data structure should be as small as possible.

It's okay if an innocuous tuple is ocassionally flagged as bad, as long as this happens with probability at most 1% for each tuple that isn't in the list.

Design:

1. an $O(tn)$-time **preprocessing algorithm** which takes $[v_1, \ldots, v_n]$ and builds a data structure that takes space $O(n)$.

2. a **lookup operation** which takes a tuple v and uses the preprocessed data structure to determine whether $v \in [v_1, \ldots, v_n]$ in time $O(t)$ and with false-positive probability at most 1% (and no false negatives).

[2] You don't need to figure out how to achieve this, but in case you're curious, the key idea is to compute a *rolling hash*: first compute $h_1 = h(a_1)$, then (abusing notation a bit) $h_2 = h(h_1, a_2)$, then $h_3 = h(h_2, a_3)$, and so on. This makes t calls to the basic hash function h.

LINEAR SORT:

★ ① FALSE → lower bounded by ~~nlogn~~ → 5! different combos

② (329, 457, 657, 839, 436, 720, 355)

LSB→MSB : (720, 355 436, 457, 657, 329, 839,)

Stable! (720, 329, 839, 436, 355 457, 657,)

(329, 355 436, 457, 657, 720 839,)

★ ③ tuple sort → repeatedly strings by each char from right → left

need to make 26 buckets, loop through n elements

k rounds of counting sort, each taking $\Theta(n+26) = \Theta(n)$ time

alg. runs in $\Theta(nk)$ time. run time = linear b/c input size is $\Theta(nk)$

④ add n^{49} to each #. all keys all $< 2n^{50}$, aka $< n^{51}$

can subtract n^{49} from each element of output to get correct #

⑤ ??

HASHING TUPLES:

Ⓐ ① make hash func. make all bad tuples go to one bucket; chances of tuple getting into bad bucket = $\frac{1}{m}$

d.s. storing all tuples will have size $\Omega(tn)$!

- hash table length = prime #
- magnitude $\Theta(n)$ where n = # of vectors
- $100n \leq m \leq 200n \rightarrow \frac{1}{m} \approx 1\%$

$$Pr[h(v) = h(v_1) \text{ or } h(v) = h(v_2) \text{ or } h(v) = h(v_3) \ldots h(v) = h(v_n)]$$

input for lookup

2) Union Bound

$\leq \frac{n}{m}$

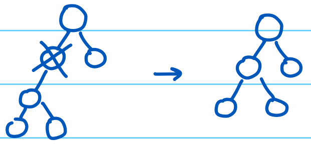
9/26 **BINARY TREES:**

- ancestor/descendant, depth/height
- traversals: in-order, pre-order, post-order → all O(n)
- left subtree < root; right subtree > root
- in-order traversal returns keys in sorted order

IMPLEMENTING SET w/ BST PROP:

- find(k) → Θ(h)
- order ops:
  - iter-ord()
  - find-min() / find-max() → Θ(h) h = height of tree
  - find-next(k) / find-prev(k) → Θ(h)

insert(k) / delete(k): Θ(h)



if leaf: delete

if 1 child: replace w/ child

if 2 child: swap w/ successor, preserves order & now have at most 1 child

Introduction to Algorithms: 6.1210
Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson — Recitation 7

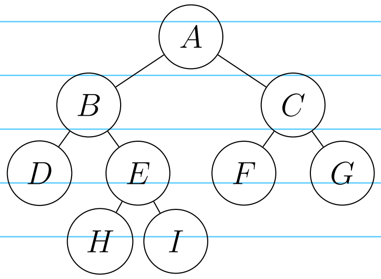
Recitation 7

Lecture Summary

- Binary Trees[1]
- Binary Search Trees Operations (find, insert, delete, successor)

Exercises

1. Give the in-order, pre-order, and post-order traversal orders for the following binary tree:



2. Practice with the BST implementation of the SET interface by inserting some items of your choice into a small BST, and then searching for and/or deleting some keys.

3. What order would we insert the elements 1, 2, 3, 4, 5, 6, 7 into a BST so that the height of the resulting binary tree is maximized? How would we insert them so that the height is minimized?

4. Prove that an in-order traversal of a BST yields a sorted array.

5. Given an array of items $A = (a_0, \ldots, a_{n-1})$, describe an $O(n \log n)$-time algorithm to construct a binary search tree T containing the items in A such that T has height $O(\log n)$. You may assume n is one less than a power of two.

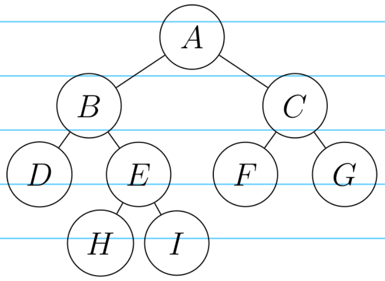
6. Let x and y be nodes in a BST, and suppose y is the successor of x and x has no right child. Prove that y is an ancestor of x, and in particular x is in the left subtree of y.

7. Argue that the following iterative procedure to return the nodes of a tree in traversal order takes $O(n)$ time.

[1] Visualizer found at https://github.com/edemaine/poketree

```python
def tree_iter(T):
    node = T.subtree_first()
    while node:
        yield node
        node = node.successor()
```

①

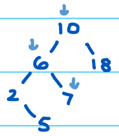


IN-ORDER: [D, B, H, E, I, A, F, C, G]

PRE-ORDER: [A, B, D, E, H, I, C, F, G]

★ POST-ORDER: [D, H, I, E, B, F, C, C, A]

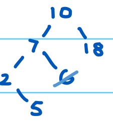
② insert: $\overset{\text{root}}{10}$, 6, 18, 2, 5, 7



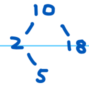
find(7) Θ(3)

inorder → find-next(5) → 6

delete(6) →



delete(7) →



FIND-NEXT(K):

if right subtree: find successor

elif left subtree: go to ancestors until reaching smthg in left

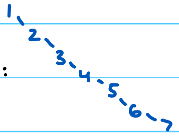
DELETE(K):

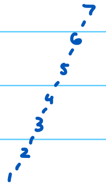
replace with right child.

if only left child: replace w/ left child.

③ maximized: in sorted order

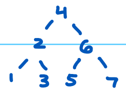
1, 2, 3, 4, 5, 6, 7 or 7, 6, 5, 4, 3, 2, 1:





minimized: start w/ middle elt.

*lots of ways to do this, as long as 2 is before 1 & 3, 6 before 5 & 7 (can insert in many diff orders)



④ PF by STRONG INDUCTION:

P(n):= in order traversal on BST w/ n nodes returns sorted order.

B.C.: n = 1 → returns itself ✓

ASSUME alg. holds for all n = 1, ..., n-1. WTS: n

in-order traversal will do left → mid → right.

- left & right are sorted by I.H.
- by BST property, all left nodes < mid < all right nodes

∴ will be returned in correct order.

★ we know alg. holds for n-1. adding 1 node will be a leaf.

Ⓡ

★ 5

- sort first → $O(n\log n)$
- make middle elt. root
- recurse left & right of root, build subtrees

time: $T(n) = 2T\left(\frac{n}{2}\right) + O(1) \rightarrow T(n) = O(n)$

height: $H(n) = H\left(\frac{n}{2}\right) + 1 \rightarrow H(n) = O(\log n)$

$2^k - 1$

$2^k - 2$

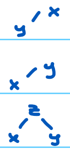
$2^{k-1} - 1$ ← won't run out of nodes to build perfect tree with

6

2 nodes x & y:

3 cases: z (ancestor):

- z = x
- z = y
- z = z



contradictions, lines of reasoning

★ 7

* only traverse every edge @ least 2x

whenever node has no R child, only traverses up.

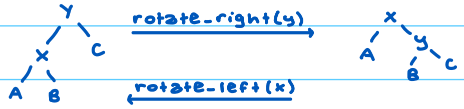
Appendix A: Binary Node and Binary Tree Implementation

```
class Binary_Node:
    def __init__(A, x):                     # O(1)
        A.item   = x
        A.left   = None
        A.right  = None
        A.parent = None
        # A.subtree_update()                # wait for R08!

    def subtree_iter(A):                    # O(n)
        if A.left:   yield from A.left.subtree_iter()
        yield A
        if A.right:  yield from A.right.subtree_iter()

    def subtree_first(A):                   # O(h)
        if A.left:  return A.left.subtree_first()
        else:       return A

    def subtree_last(A):                    # O(h)
        if A.right: return A.right.subtree_last()
        else:       return A

    def successor(A):                       # O(h)
        if A.right: return A.right.subtree_first()
        while A.parent and (A is A.parent.right):
            A = A.parent
        return A.parent

    def predecessor(A):                     # O(h)
        if A.left:  return A.left.subtree_last()
        while A.parent and (A is A.parent.left):
            A = A.parent
        return A.parent

    def subtree_insert_before(A, B):        # O(h)
        if A.left:
            A = A.left.subtree_last()
            A.right, B.parent = B, A
        else:
            A.left,  B.parent = B, A
        # A.maintain()                      # wait for R08!

    def subtree_insert_after(A, B):         # O(h)
        if A.right:
            A = A.right.subtree_first()
            A.left,  B.parent = B, A
        else:
            A.right, B.parent = B, A
        # A.maintain()                      # wait for R08!
```

```python

def subtree_delete(A):                  # O(h)
    if A.left or A.right:
        if A.left:  B = A.predecessor()
        else:       B = A.successor()
        A.item, B.item = B.item, A.item
        return B.subtree_delete()
    if A.parent:
        if A.parent.left is A:  A.parent.left  = None
        else:                   A.parent.right = None
         # A.parent.maintain()          # wait for R08!
    return A
```

10/1 **AVL ROTATIONS:**

- O(1)
- maintain traversal power



AVL PROPERTY: for every node m, $|skew(m)| \leq 1$, can prove $height \in O(\log n)$

CALCULATE SKEW of node in constant tree:

- need height of left & right
- maintain height of every node (stored @ every node)
- insert: new leaf gets height = 0; every node on path to roots gets updated
- m height = max(m.left height, m.right height) + 1
- delete: only need to update ancestors, do something

MAINTAIN AVL PROPERTY W/ ROTATIONS:

- insert/delete may cause a rotation on an ancestor
- at most 2 rotations to fix violation O(1)
- deletion: may propogate the violation up & may need to keep rotating O(logn)
- always consider lowest value w/ a validation

Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

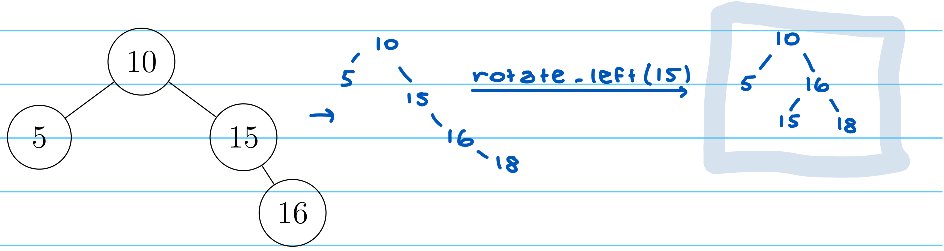
Recitation 8

Lecture Summary

- Balanced Binary Trees

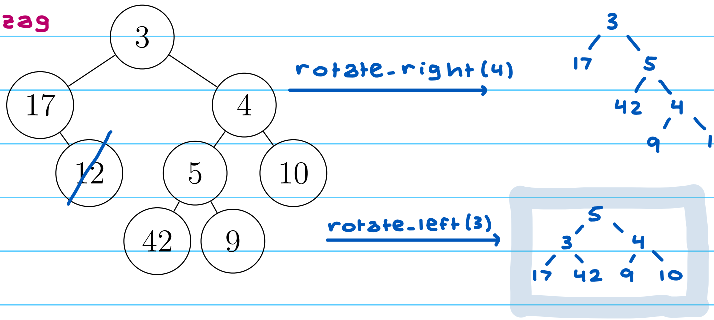
Exercises

1. What does this AVL binary search tree look like after inserting 18?



2. What does this AVL tree look like after deleting 12?

#rotate 2x b/c there's zigzag



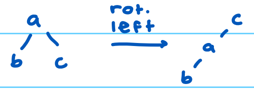
3. When we rotate a tree, right or left, the in-order traversal remains the same. Do the pre-order and post-order traversals remain the same?

4. We've defined the AVL Property in terms of skew. Specifically, a tree has the AVL Property if every node, n, in it has $|\text{skew}(n)| \leq 1$. An AVL Tree can reach any of the leaves from the root in $O(\log n)$ time. Imagine an AVL-2 Tree that allows a skew of up to two. I.e., it only guarantees $|\text{skew}(n)| \leq 2$. Show that the height of such a tree is still in $O(\log n)$.

5. We saw how to create an iterator for an AVL tree in which the Next operation takes amortized constant time but worst-case logarithmic time. Describe how to change the AVL tree such that this Next operation takes worst-case constant time. The functionality of the AVL tree (including asymptotic runtimes) should otherwise be unchanged.

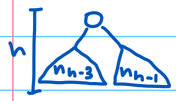
6. Suppose you have some items x, each of which has one comparable key $k_1(x)$ and one hashable key $k_2(x)$. Assume that both keys are unique. Describe an unordered SET data structure that can INSERT efficiently and can FIND or DELETE efficiently using either key. What is the runtime of each operation, and is it worst-case, amortized, and/or expected?

③ not necessarily - such as in question 2, some left/right children get swapped, which would change the ordering of pre/post order traversal.



pre: abc cab

④ want to consider smallest (least nodes) tree of height h.



call this number of nodes n_h

$h < \log n_h < \log n$

$n_h = n_{h-1} + n_{h-3} + 1$

$n_h > 2 \cdot n_{h-3}$ ← lower bounded

$n_h > 2 \cdot (2 n_{h-6})$

$n_h > 2^{h/3}$

$h < 3 \log n_h < 3 \log n \in O(\log n)$

⑤ we know traversing costs linear $O(n)$ time, through every node.

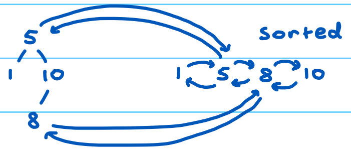
TLDR: want to go through predecessors in $O(1)$ worst case time

→ LINKED LIST

CROSS-LINKING:

- delete: normal AVL deletion & linked list deletion
- insert: normal AVL insertion

  insert into linked list from parent b/t elts.



*could've also put pointers b/t elts in tree → each node points to its predecessor/successor

⑥ AVL ↔ hash table

→ cross link AVL tree (keyed by k_1) w/ hash table (keyed by k_2).

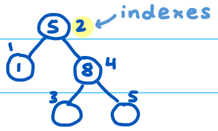
- each chain in hash will be a linked list of tree nodes
- find by k_1 uses 1 AVL find: $O(\log n)$ worst
- find by k_2 uses 1 hash find: $O(1)$ expected
- insert/del dynamic ops → am/ex $O(\log n)$

10/3 AUGMENTATIONS

ORDER-STATISTIC TREE: AVL tree w/ size augmentation (stores size of subtree)

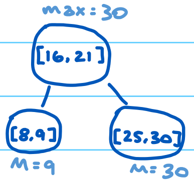
```
os_select(T, index):
    curr_index = T.left.size + 1
    if curr_index == index:
        return T
    if          < index:
        os_select(T.left, index)
    if          > index:
        os_select(T.right, index)
```

```
os_rank(T, x):
    count = x.left.size +1
    walk up tree; if reach node from right,
    add  node.left.size + 1
```



AUGMENTATIONS: can augment on AVL tree w/ field F w/o changing runtimes as long as F can be computed for any node from just itself & data from left & right in O(1). ← NOT ancestors

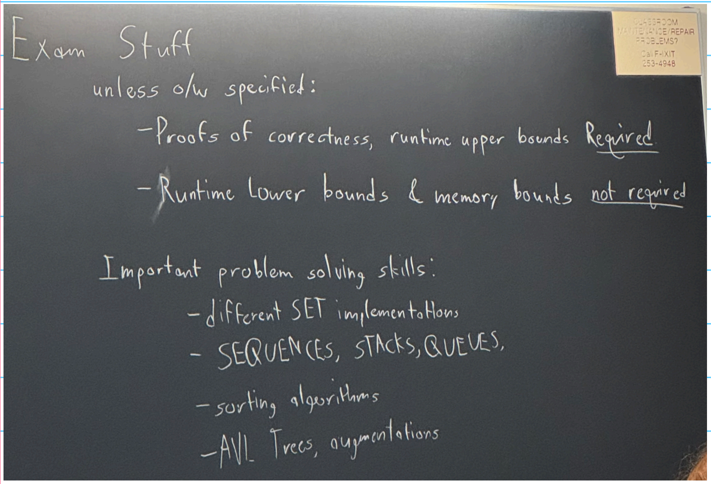
INTERVAL TREE: AVL BSTs w/ x.low & x.high, keyed by x.low



augmented by M max of any x.high in subtree.

interval_search(T[a,b]): finds node that intersects w/ [a,b]

while node T doesn't overlap, traverse down the tree, going left if x.left M ≥ a



Exam Stuff

unless o/w specified:

- Proofs of correctness, runtime upper bounds Required
- Runtime lower bounds & memory bounds not required

Important problem solving skills:

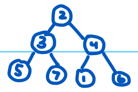
- different SET implementations
- SEQUENCES, STACKS, QUEUES,
- sorting algorithms
- AVL Trees, augmentations

CLASSROOM MAINTENANCE/REPAIR PROBLEMS? Call F-IXIT 253-4948

AVL SET: BST property

AVL SEQUENCE: maintains order

5	3	7	2	1	4	6



sequence: does not have keys;
only have indices;
care about maintaining order

set: care ab. being able to find things;
sorted by smthg. other than index

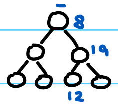
SET:	AVL tree	Dynamic Array
insert/del:	O(n)	O(n)
find:	O(logn)	O(logn)

am. cost O(T) if for any m ops, O(mT)

SEQUENCE:	order statistic tree	dynamic array:
	get_at(i) → O(logn)	O(1) ↖ good for grabbing things
	insert.at(i) → O(logn)	O(n)

RANGE QUERIES:

- max rating subject to cost ≤ m



key by cost (inequal.),

augment by max rating

≤ 14: 8 < 14, so everything on left can be considered

*review pset ر

Introduction to Algorithms: 6.1210 Recitation 9
Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

Recitation 9

Lecture Summary

- Order Statistics
- Augmentations
- Interval Trees

Exercises

1. Suppose you want a data structure that maintains a SET of integers, with the additional operation CLOSEST(), which should in constant time return the minimum positive difference between two elements of your SET.

   (a) Clearly CLOSEST() cannot be computed from scratch in constant time. A natural strategy to attempt in such a situation is to make it an augmentation. Explain why CLOSEST() cannot be an augmentation.

   (b) Describe (with proof) an augmentation from which CLOSEST() can be computed in constant time.
   Hint: Use an ordered triple that captures the difference between the examples you described above.

2. Augmenting a binary tree can be useful if the items being stored need to support queries based on two different data. The choice of key and augmentation need not be unique. Often there are dual representations in which the rôles of key and augmentation are swapped. Describe (with proof) an interval tree in which the interval $[a, b]$ is keyed by b instead of a.

3. Which of the following proposed augmentations for a binary search tree would work? For each one, either (briefly) describe how to efficiently compute the augmentation at v from the augmentation at its children, or explain why this is impossible. Assume the BST has no other augmentations.

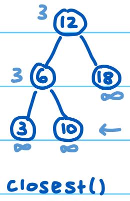
   - $f_1(v)$ = the **median of the keys** in the subtree rooted at v
   - $f_2(v)$ = the **three biggest values** in the subtree rooted at v
   - $f_3(v)$ = the **number of odd values** in the subtree rooted at v
   - $f_4(v)$ = the **height of the left subtree** under v (i.e. the subtree rooted at the left child of v)

4. Describe a data structure that maintains a sequence of n bits and supports two operations, each in $O(\log n)$ time:

- `flip(i)` : flip the bit at index i
- `ones_upto(i)` : return the number of bits in the prefix up to index i that are one

① a) augmentations should apply to all nodes in tree, which can be ≥ 2. ∴ this aug. won't work.

closest() : even w/ augmentation, have to compute difference w/ everything in subtree

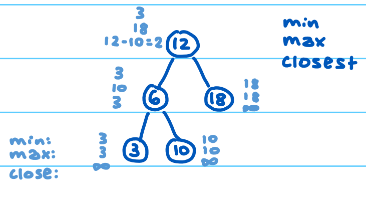


← 10 & 12 closer, but can't compute aug. in constant time b/c might have to go all the way down tree

b) 3 augmentations: $T.min = min(T.left.min, T.item)$

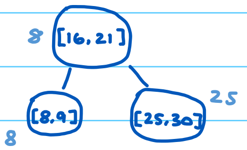
$T.max = max(T.right.max, T.item)$

$$T.closest = min \begin{cases} T.item - T.right.min \\ T.item - T.left.max \\ T.left.closest \\ T.right.closest \end{cases}$$



② all intervals [a,b] keyed by b & augmented by min: minimum a in subtree

```
interval_search(T, [a,b]):
    if [a,b] intersects T's interval, output T.
    else if T.right.min ≤ b, recurse right
    else recurse left
```



a b
[17,22]

17<21, 22>16

PF OF CORRECTNESS: always recurse on side w/ answer

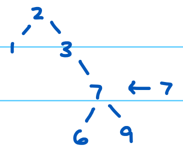
say [a,b] intersects w/ some [c,d], so c ≤ b & a ≤ d

let [c,d] be interval w/ max d.

if [c,d] in right subtree, T.right.min ≤ c ≤ b, so recurse on right

if [c,d] in left subtree, for any [c', d'] in right subtree, it cannot intersect [a,b] (c' > b). but T.right.min ≤ c', so recurse on left.

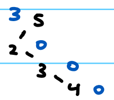
③ f_1 not possible b/c each subtree's median is diff.



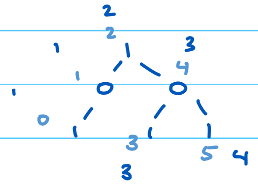
★ f_2 possible: merge sort ℓ & r values & keep first 3 elts.

f_3 possible: $f_3(\ell) + f_3(r) + 1$ (if v.value is odd)

f_4 not possible: think ab. left vs. right → ex:



4 **Solution:**



Use a binary tree with n nodes, height $O(\log n)$, and size augmentations (this could be an AVL tree, but we'll never need the rebalancing features). The ith node in traversal order stores the value of the ith bit, and we augment each node `A` with `A.subtree_ones`, the number of 1 bits in its subtree. We can compute this in $O(1)$ time from the augmentations stored at its children: add its childrens' `subtree_ones`, and add one if `A` has a 1.

To implement `flip(i)`, find the i^{th} node `A` using `get_at(i)` and flip the bit stored at `A.item`. Then update the augmentation at `A` and every ancestor of `A` by walking up the tree in $O(\log n)$ time.

To implement `ones_upto(i)`, we use `get_at(i)` and track the 1s we've passed so far. Specifically, whenever we go right from a node x, we add `x.left.subtree_ones` to our current count, and add one more if the bit at x is 1. This takes the same amount of time as `get_at`, which is $O(\log n)$.

Here's a recursive implementation of `ones_upto(T, i)`:

- Base case: if `T` is empty, return 0
- If $i <$ `size(T.left)`: return `ones_upto(T.left, i)`
- If $i =$ `size(T.left)`: return `subtree_ones(T.left)`
- If $i >$ `size(T.left)`: return `subtree_ones(T.left)` $\overset{0/1}{\underline{+\,\texttt{T.item}}}$ `+ones_upto(T.right,` $\underbrace{\texttt{i - size(T.left) - 1}}_{\text{index}}$`)`

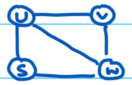
↑ EXAM 1 CONTENT ↑

10/8 GRAPHS:

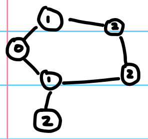
- directed/undirected

REPRESENTATION:

- adjacency matrix
- adjacency list {u: [v, w, s]
  s: [u, w]
  v: [u, w]
  w: [u, v, s]}



BFS: shortest path from a node (SSSP)



O(|V|+|E|)

DFS: not shortest path O(|E|)

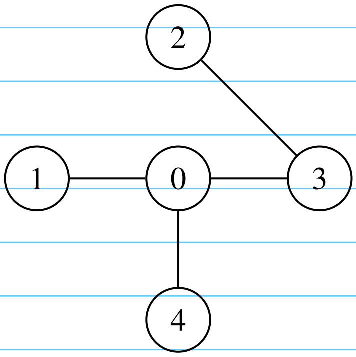
Recitation 10

Lecture Summary

- Graphs and representations
- Graph Problems
- Breadth First Search
- Depth First Search

Exercises

1. Write the adjacency list and adjacency matrix representations for the graph below.

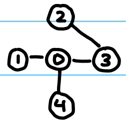


2. Given a directed graph G, construct and analyze the runtime of an algorithm to create graph G^R which is identical to G but with all the edges reversed. Solve for both an adjacency list and adjacency matrix representation.

3. Describe how to implement BFS using a QUEUE as the main data structure. Remember that a QUEUE supports the `enqueue` and `dequeue` operations.

4. Describe how to implement DFS using a STACK as the main data structure. Remember that a STACK supports the `push` and `pop` operations.

5. Given an unweighted graph $G = (V, E)$ in which some edges are red and some are blue, find a path from s to t with the minimal number of red edges.

1) 

ADJ. LIST:

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

ADJ. MATRIX:

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

2) ADJ LIST:

- create new empty adj. list
- for each node v, go through its list
- for each node u in list, add v to u's list

ADJ. MATRIX:

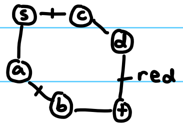
- transpose $O(|V|^2)$

3)
```
start w/ source on queue, mark source as visited
while not empty:
    · dequeue
    · for each unvisited neighbor:
            · enqueue
            · mark as visited
              d[v] = d[u]+1
```

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

4)
```
start w/ source on stack, mark as visited
while nonempty:
    · pop
    · for each unvisited neighbor:
        - push
        - mark as visited
```

$O(|E|)$

★ 5) 

IDEA: do BFS w/ a bunch of mini DFSs to ignore blue edges

DEQUEUE, can add/remove from both sides

```
initialize Q=[s], d(S)=0. mark s as visited
while Q non-empty:
    · remove 1st elt. u from Q. for each unvisited neighbor v of u:
        - set p(v)=u
        · (u,v) red: d(v)=d(u)+1 & append v. mark v as visited
        · (u,v) blue: d(v)=d(u) & prepend v. mark v as visited
```

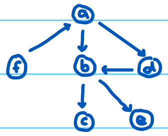
10/15 REC 11

FULL DFS: run DFS from a node $O(|V|+|E|)$

when it terminates, start again from an unvisited node.

repeat until all nodes are visited

DFS FINISHING ORDER:

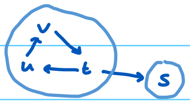


mark in order they finish (starting nodes should be last)

c, e, b, etc.

STRONGLY CONNECTED COMPONENTS:

for any nodes u & v in some SCC if path from u to v & v to u

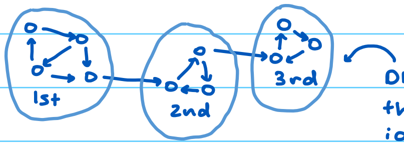


KOSARAJU-SHARIR: run full DFS on G^{rev} & record finishing times f[v] for each v

for each v in reverse f[v] order: Ex: f = a, b, c ↙ start here

- set current = v
- if v unvisited, run DFS starting from v. for any vertex u that doesn't have a leader, set leader[u] = v
  grouping SCCs

intuition: hierarchy of SCCs



DFS starting from 3rd group, then 2nd, then 1st. identify SCCs.

* if start from 1st, then everything would be classified as SCC together.

CONDENSATION GRAPH: node for each SCC, edge for any edges b/t nodes in SCCs

→ must be DAGs!

Introduction to Algorithms: 6.1210 Recitation 11
Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

Recitation 11

Lecture Summary

- Connectivity
- Strong connectivity
- Kosaraju-Sharir (KS)

Exercises

1. Design an algorithm to build the condensation graph $G_C = (V_C, E_C)$ of $G = (V, E)$.

2. **Cycle Detection**

   graph congruence (same nodes/edges)

   (a) Prove that $G_C \cong G$ iff G is acyclic. → must prove both directions!

   (b) Explain how to detect a cycle in a directed graph in linear time.

   (c) Explain how to detect a cycle in a directed graph using only a single Full-DFS.

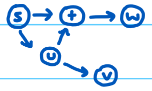
3. There are n lock boxes and m keys. Each box has a distinct lock, so each key can open exactly one box. There's at least one copy of each box's key, but for some boxes there may be multiple copies of the key. Someone put all keys in the boxes and locked them up, but luckily they made a note of which keys are stored in each box. Keys and boxes are numbered so that we know which box is opened by each key. Some boxes contain no keys while others contain multiple keys. Boxes can also be forced open with a rusty crowbar. Design an algorithm to find the smallest set S of boxes that you need to force open in order to open all the other boxes.

① APPROACH:

- run Kosaraju-Sharir → now each node marked w/ leader & we know the SCCs
- iterate through nodes & check leader, create node for each unique leader. O(|V|)
- iterate through edges u→v, add to G_{con} if leader of u ≠ leader of v O(|E|)

overall: O(|V|+|E|)

② a)



$G_c \cong G$ iff G is acyclic

⟶ if $G_c \cong G$, then G is acyclic:

prove both dirs. for iff

they have same # nodes, so all SCCs are single nodes. ∴ no cycles.

⟵ if G is acyclic, then $G_c \cong G$

L(u) = u → each node labels itself as leader for all u ("each node is SCC")

since all edges u→v go b/t diff. SCCs in original, edges are preserved.

acyclic = no cycles, so no SCCs → condensation graph reduces SCCs.

b) run KS. check if L(v) = v for all v.

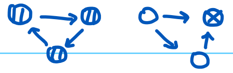
if not, there is a cycle b/c not every node is its own leader } less efficient b/c KS takes 2 cycles of DFS.

* can also check if # nodes in G & G_{con} is equal.

★ c) single full DFS: check active vs. finished for nodes.

if an edge goes back to a visited but unfinished node,

that node is an ancestor in DFS tree & forms cycle.



only becomes finished when all paths have been gone to.

★ ③ create graph G where:

nodes: boxes

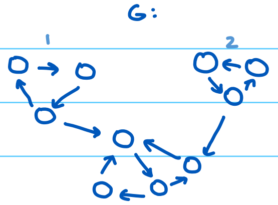
edges: connect u→v if u contains a key that opens v.

observe within SCCs, once one is unlocked, you have them all.

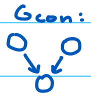
run KS, make G_{con}.

count nodes w/ indegree = 0 (source nodes)

return count



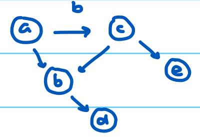
for this, breaking open 1 & 2 → 3 can be opened.



10/17 **TOPOLOGICAL ORDER:**

- ordering of nodes s.t. if $(u,v) \in E$, u comes before v in the order
- G has topo. order IFF G is DAG
- to get topo. order: full DFS, reverse finishing time is topo order

Ex: a, c, $\overset{e}{b}$ etc.



REDUCTION: alg. for transforming any problem of type A to type B ($A \leq B$)

TURING REDUCTION: $A \leq_T B$ if exists some alg. that solves A while using B as atomic subprob. (call B in constant time aka "oracle")

- if this alg is poly time: Cook Reduction (reduction process, not B's R.T) *focus on the process of reducing

MANY-ONE REDUCTION: $A \leq_m B$ if exists some function f that converts input of A to input of B. (feed into B – last thing we do)

(more specific)
(restrictive)

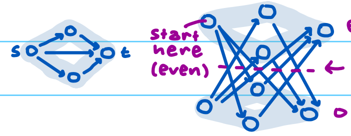
- if f takes poly time: Karp Reduction

Ex: can be n^{100}, not e^n

REDUCTION W/ GRAPHS:

- transform the obvious graph into another that you can just run alg. on

GRAPH DUPLICATION: even SPSP reduces to SPSP



start here (even)

even

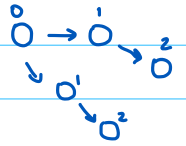
← edges flip b/t odd/even

odd ← if you end up here, length of path = odd

Ex: only want to return even # of paths

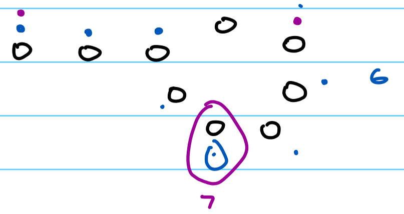
*have BFS keep track of even/odd

note about SSSP: can convert b/t $\delta(s, -)$ & shortest path tree in linear time



*for unweighted

DFS:



Introduction to Algorithms: 6.1210 Recitation 12
Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

Recitation 12

Lecture Summary

- Reduction
- Turing / many-one reductions
- Graph duplication
- Self-reduction

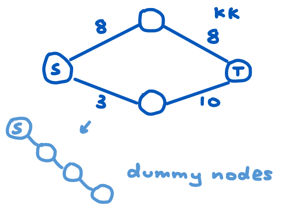
Exercises: Ash's Archipelago Adventure

Ash and his Pokémon are on vacation in the Orange Islands, represented as an undirected graph $G = (V, E)$. Each vertex is an island, and each edge is a route that can be traversed in either direction. Each route $\{u, v\}$ costs a given positive integer number $c_{u,v}$ of Power Points (PP) to navigate. Everyone is currently on Shamouti Island, and some of Ash's Pokémon want to go to different islands. Help Ash's Pokémon reach their desired destinations.

1. Totodile has k PP to spend and wants to go to Tangelo Island. Design an $O(k|E|)$ time algorithm to decide whether Totodile can reach Tangelo Island.

2. Ducklett has k PP to spend and wants to go to Tarroco Island. Ducklett can either swim across a route $\{u, v\}$ by spending $c_{u,v}$ PP, or fly over a route by spending 1 PP, regardless of $c_{u,v}$. However, after flying, Ducklett must rest and cannot fly again until after swimming across another route. Design an $O(k|E|)$ time algorithm to decide whether Ducklett can reach Tarroco Island.

3. Squirtle and Fletchling want to go to Trovita Island and can *each* spend at most k PP. Squirtle can swim (and carry Fletchling) across a route $\{u, v\}$ by spending $c_{u,v}$ PP. Fletchling can fly (and carry Squirtle) over a route $\{u, v\}$ by spending 5 PP, regardless of $c_{u,v}$. However, after flying, Fletchling must rest and cannot fly again until after being carried over a route. Design an $O(k^2|E|)$ time algorithm to decide whether Squirtle and Fletchling can reach Trovita Island.



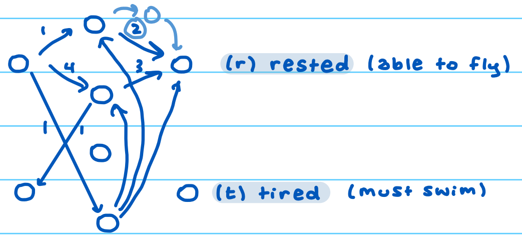
① · construct G_0 by removing unreachable vertices from s via DFS ← make sure r.t. based on |E|, not |V|

· construct G by dividing every edge (u,v) by $c_{u,v}$-edge chain if $c_{u,v} \leq k$ (o.w. remove edge)

* in general, DONT keep track of states w/ DFS! will get exponentially large

· BFS in G, from S to get dist. d to T → output $d \leq k$

· RT: @ most k|E| edges, @ most |V| + |E|(k-1) nodes. can bound w/ E: $O(k|E|)$

② 

*STATE: whether we can fly

directed

construct G w/ duplicate nodes u_r & u_t

· for all (u, v), add (u_r, v_t), add (v_t, u_r)

· if $c_{u,v} \leq k$: make $c_{u,v}$ directed chain: (choosing not to fly)

1: from u_r to v_r

2: from v_r to u_r $O(k|E|)$

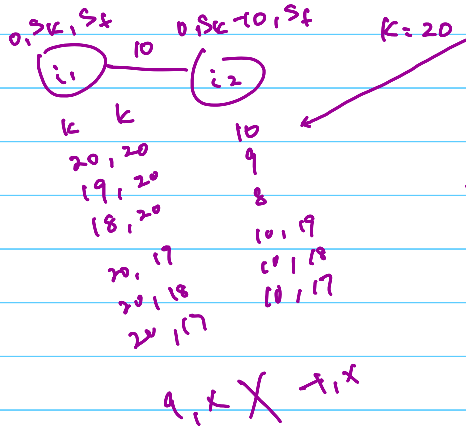
3: from u_t to v_r

4: from v_t to u_r

③ need to duplicate $\underline{k^2}$ times

· keep track of Squirtle & Fletching *STATE: whether we can fly + pp pts for each

· keep track of PP pts for both



* need to create a node for all possible values for both Squirtle & Fletching for each edge weight → $\Theta(|E|k^2)$

· can use BFS or DFS

· reachable to node

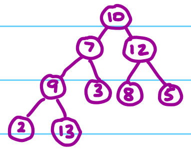
10/22

Arrays as Compact Binary Trees:

10	7	12	9	3	8	5	2	13	

children of i: 2i, 2i+1

parent of i: $\lfloor \frac{i}{2} \rfloor$



not a heap
can heapify-down(10)
& other violations

Priority Queue Interface:

	AVL:		CBT:
· build	nlogn	??	n
· insert(x)	logn		
· delete_max()	logn		

has to sort

d.s.

Heaps (compact binary tree implementation of PQ):

· Heap Property: key of every node > key of its children

· OPERATIONS:

- heapify-up(x): $\Theta(\log n)$ swap x up the tree while its bigger than parent
- heapify-down(x): $\Theta(\log n)$ swap x down the tree w/ bigger child while its smaller than a child

Priority Queue Operations:

- insert(x) $\Theta(\log n)$ put x at end, heapify-up(x)
- delete-max() $\Theta(\log n)$ swap first & last element in array, delete, heapify root down
- build $\Theta(n)$ heapify every item down, starting from leaves

Heap Sort: $\Theta(n\log n)$ in-place sorting!

1) build a heap

2) repeatedly remove max element

1	7	9	4	2	3	8	10	15

* in-place $\Theta(n\log n)$

Max Heap or Min Heap:

· can use either!

Introduction to Algorithms: 6.1210 Recitation 13
Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

Recitation 13

Lecture Summary

- Priority Queues
- Heaps
- Heap sort

Exercises

1. Practice with heaps:

   (a) Draw the compact binary tree associated with the array

   ```
   A = [7, 3, 5, 6, 2, 0, 3, 1]
   ```

   (b) Turn it into a max heap via linear time bottom-up heap-ification.

   (c) Run `insert(9)`.

   (d) Run `insert(4)`.

   (e) Run `delete_max`.

   (f) Run `delete_max` again.

2. How would you find the **minimum** element contained in a **max** heap?

3. How long would it take to convert a **max** heap to a **min** heap?

4. **Top-k Leaderboard**: In order to maintain a Tetris leaderboard, Alice would like to keep track of the k highest-scoring players, but does not care about the ranking of those players as long as they are the highest-scoring set of k players. Scores update whenever a new game is finished. At any given point in time, the highest-scoring player may be disqualified and have their score removed from the leaderboard.

   Alice is requesting a data structure that would allow her to efficiently maintain her leaderboard. Assuming a total of n players, she would like to support the following operations:

   - `insert`: insert players and their scores to the data structure in $O(\log n)$
   - `remove`: remove the highest-scoring player and their score from the data structure in $O(\log n)$ time

- `build`: given an initial array of n players and their scores, construct the data structure in $O(n \log k)$ time
- `highest-ranking-k`: return the highest-scoring set of k players, as a pointer to an array containing these players, in $O(1)$ time

Note that there is not enough time to construct the desired array of k players (which would take $O(k)$ time), but it is sufficient to return a pointer to an array that has already been constructed as a part of the data structure, which would only take $O(1)$ time. Also, recall that you must support insertion of new players in $O(\log n)$ time, and deletion of the highest-scoring player in $O(\log n)$ time.

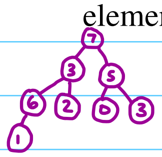
5. **Proximate Sorting**: An array of **distinct** integers is ***k-proximate*** if every integer of the array is at most k places away from its place in the array after being sorted, i.e., if the ith integer of the unsorted input array is the jth largest integer contained in the array, then $|i - j| \leq k$. In this problem, we will show how to sort a k-proximate array faster than $\Theta(n \log n)$.

   (a) Prove that insertion sort (as presented in this class, without any changes) will sort a k-proximate array in $O(nk)$ time.

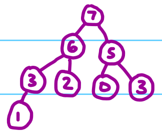
   (b) $\Theta(nk)$ is asymptotically faster than $\Theta(n^2)$ when $k = O(\log n)$, but is not asymptotically faster than $\Theta(n \log n)$ when $k = \Omega(\log n)$. Describe an algorithm to sort a k-proximate array in $O(n \log k)$ time, which can be faster (but no slower) than $\Theta(n \log n)$.

   Hint: Can you identify a subsequence of the array which must contain the minimum element?

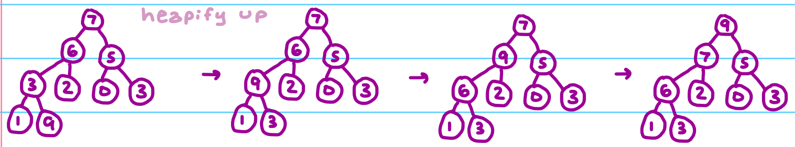
1) a)



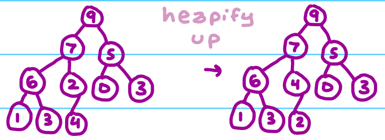
b)



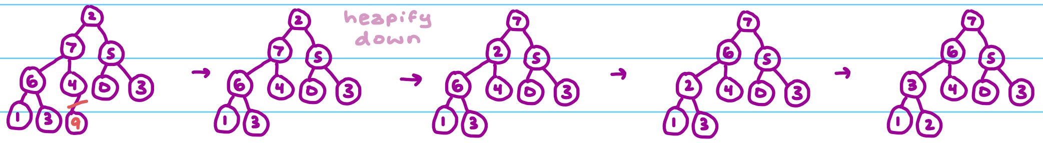
c)



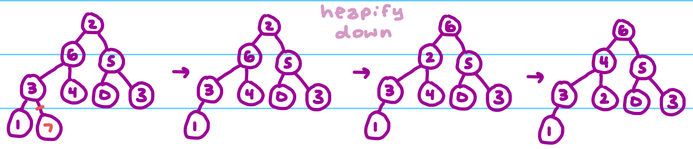
d)



e)



f)



② how to find min of max heap?

look @ leaves → min could be any of the leaves

search all $O(\frac{n}{2})$ leaves

↳ tldr max heaps don't rly tell us anything ab. min

③ convert max heap → min heap   * heap build doesn't sort, which is how we achieve Θ(n) linear

Θ(n) → heap build from start as if you didn't know anything about it.

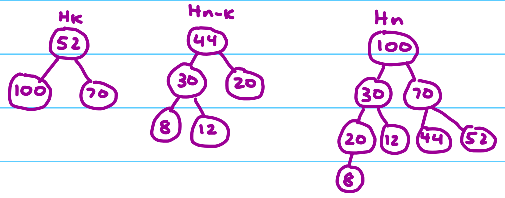
④ maintain highest-ranking-k: $\overset{H_n}{\text{min heap}}$ containing only the top k scores of n players

- top player has worst score (1st on the chopping block)
- $\overset{H_{n-k}}{\text{max heap of rest}}$ (n-k) players so we know who to promote (if needed)

remove top player: $\overset{H_n}{\text{max heap}}$ of all players (best @ top)

cross-link: maintain pointers between corresponding items

Ex: k=3



__BUILD:__

- make Hk with just first k players] O(k)
- for each remaining n-k items, compare to Hk's min. Hk.delete-min(), add new item] O((n-k)logk)
- construct Hn-k w/ heap build] O(n-k)
- construct Hn w/ heap build] O(n)

__HIGHEST-RANKING-K:__

- return Hk in O(1) time
- min heap can be compactly maintained as array

__INSERT:__

- compare player's score to root of min heap Hk
  - if greater: pop root, insert new player's score, insert removed player into Hn-k
  - if less: insert into Hn-k

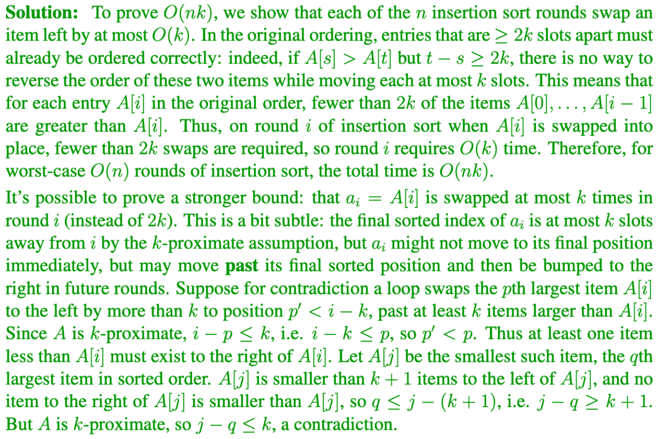
O(logk) + O(log(n-k)) + O(logn) = O(logn)

__REMOVE:__

player must be in Hk. pop root of Hn, obtain & delete in Hk O(logn), pop max player from Hn-k O(log(n-k))

- insert popped players into Hk to maintain its k players.

⑤ a) must prove each n insertion sort rounds swap an item left by @ most k.

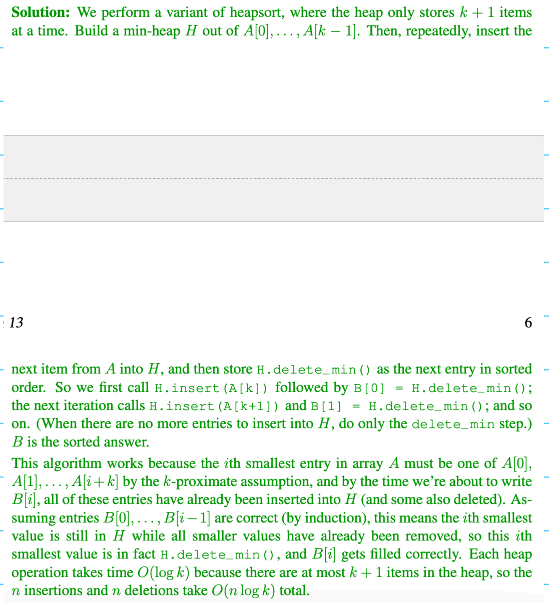


Solution: To prove $O(nk)$, we show that each of the n insertion sort rounds swap an item left by at most $O(k)$. In the original ordering, entries that are $\geq 2k$ slots apart must already be ordered correctly: indeed, if $A[s] > A[t]$ but $t - s \geq 2k$, there is no way to reverse the order of these two items while moving each at most k slots. This means that for each entry $A[i]$ in the original order, fewer than $2k$ of the items $A[0], \ldots, A[i-1]$ are greater than $A[i]$. Thus, on round i of insertion sort when $A[i]$ is swapped into place, fewer than $2k$ swaps are required, so round i requires $O(k)$ time. Therefore, for worst-case $O(n)$ rounds of insertion sort, the total time is $O(nk)$.

It's possible to prove a stronger bound: that $a_i = A[i]$ is swapped at most k times in round i (instead of $2k$). This is a bit subtle: the final sorted index of a_i is at most k slots away from i by the k-proximate assumption, but a_i might not move to its final position immediately, but may move **past** its final sorted position and then be bumped to the right in future rounds. Suppose for contradiction a loop swaps the pth largest item $A[i]$ to the left by more than k to position $p' < i - k$, past at least k items larger than $A[i]$. Since A is k-proximate, $i - p \leq k$, i.e. $i - k \leq p$, so $p' < p$. Thus at least one item less than $A[i]$ must exist to the right of $A[i]$. Let $A[j]$ be the smallest such item, the qth largest item in sorted order. $A[j]$ is smaller than $k + 1$ items to the left of $A[j]$, and no item to the right of $A[j]$ is smaller than $A[j]$, so $q \leq j - (k + 1)$, i.e. $j - q \geq k + 1$. But A is k-proximate, so $j - q \leq k$, a contradiction.

O(kn)

b)



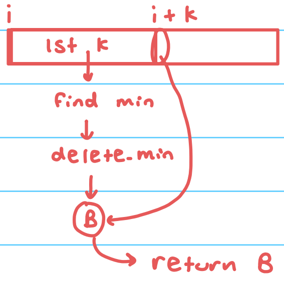
Solution: We perform a variant of heapsort, where the heap only stores $k + 1$ items at a time. Build a min-heap H out of $A[0], \ldots, A[k-1]$. Then, repeatedly, insert the

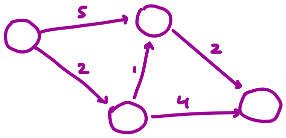
next item from A into H, and then store `H.delete_min()` as the next entry in sorted order. So we first call `H.insert(A[k])` followed by `B[0] = H.delete_min()`; the next iteration calls `H.insert(A[k+1])` and `B[1] = H.delete_min()`; and so on. (When there are no more entries to insert into H, do only the `delete_min` step.) B is the sorted answer.

This algorithm works because the ith smallest entry in array A must be one of $A[0]$, $A[1], \ldots, A[i+k]$ by the k-proximate assumption, and by the time we're about to write $B[i]$, all of these entries have already been inserted into H (and some also deleted). Assuming entries $B[0], \ldots, B[i-1]$ are correct (by induction), this means the ith smallest value is still in H while all smaller values have already been removed, so this ith smallest value is in fact `H.delete_min()`, and $B[i]$ gets filled correctly. Each heap operation takes time $O(\log k)$ because there are at most $k + 1$ items in the heap, so the n insertions and n deletions take $O(n \log k)$ total.

an obj. is only k spots away from target

- input array A
- first k elts. must contain min elt.
- heapify it (min)
- B → delete min on A





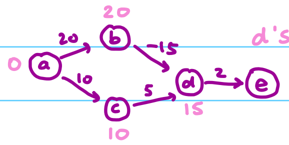
10/24

DIJKSTRA: (SSSP on non-negatively weighted graph)

- distance estimates $d[v]$ start @ ∞, except source $d[s] = 0$
- consider nodes in increasing distance
    - for all outgoing nodes, relax them.
        - **RELAX:** if $d[v] > d[u] + w(u,v)$, set $d[v] = d[u] + w(u,v)$

intution: want to relax each edge exactly once! need to make sure that when you relax edges from a node, that node's distance is final

- doesn't work for negative edge weights, need to get further @ every edge

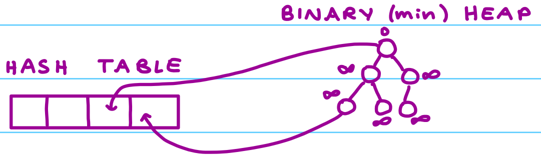


d's dist. not final b/c negative set us back (not as far)

CHANGEABLE PRIORITY QUEUE: $O(|V|\log|V| + E)$

- build
- insert
- delete min $\Theta(\log n)$

NEW! → - decrease key $\Theta(1)$ ← Fibonaucci Heap



Store every $v \in V$ keyed by distance estimate

perform one delete min for every node & one decrease key for every edge

DIJKSTRA on different metrics: $x \le y \longrightarrow x + z \le y + z$

- min, + $[0, \infty)$
- max, × $[1, \infty)$
- min, max $(-\infty, \infty)$
- max, min $(-\infty, \infty)$

1) $x \le y \longrightarrow x * z \le y * z$

2) need identity to be best elt.

3) need dummy value to be worst elt

Introduction to Algorithms: 6.1210 Recitation 14

Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

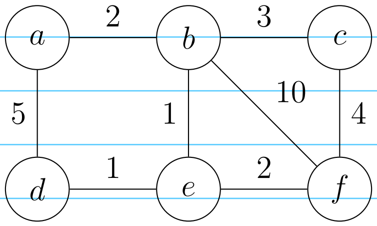
Recitation 14

Lecture Summary

- Dijkstra's Algorithm
- Why Dijkstra works
- Dijkstra Runtime Analysis

Exercises

1. Using the graph below, apply Dijkstra's algorithm from source node a to find shortest paths.



   (a) What are the current $d[v]$ values right before the second vertex is deleted from the queue? What about before the fourth vertex is deleted?

   (b) What is the final order of nodes added to the visited set? Are there multiple possible orders?

   (c) For which two vertices v does $d[v]$ change more than once? Recall the first change for each vertex is from ∞ to a finite number.

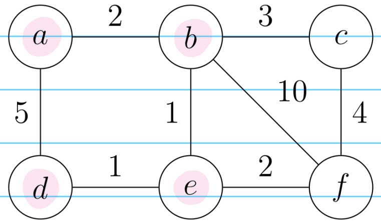
* multiple sources w/ single source R.T.

2. In a graph $G = (V, E)$ with positive edge weights, there are special nodes, $s_1, s_2, \ldots, s_k$, where k is not necessarily a constant. For every node v in the graph, we want to find the distance from the closest special node to v. Describe a $O(|E| + |V| \log |V|)$ algorithm to do so.

3. CIA officer Mary Cathison needs to drive to meet with an informant across an unwelcome city. Some roads in the city are equipped with government surveillance cameras, and Mary will be detained if cameras from more than one road observe her car on the way to her informant. Mary has a map describing the length of each road and knows which roads have surveillance cameras. Help Mary find the shortest drive to reach her informant, being seen by at most one surveillance camera along the way.

4. Ash is trying to cycle from Pallet Town to Viridian City without destroying Misty's bike. For every trail e in the area, he knows the probability $p(e)$ of destroying the bike if he cycles along e. Help Ash find the safest path to Viridian City (the path that minimizes his probability of destroying the bike). Assume that all probabilities are independent and that arithmetic operations take constant time.

①



a) d[v]

a	0	0	0	0	0	0	0
b	∞	2	2	2	2	2	2
c	∞	∞	5	5	5	5	5
d	∞	5	5	4	4	4	4
e	∞	∞	3	3	3	3	3
f	∞	∞	12	5	5	5	5
		↑ a	↑ b	↑ e	↑ d	↑ f	↑ c

could be either

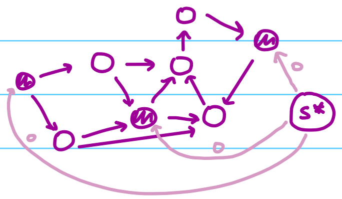
before 2nd vertex is deleted from queue:

{a: 0, b: 2, c: ∞, d: 5, e: ∞, f: ∞}

b) a, b, e, d, f, c

c) d & f

☆ ②



super node s*

connect s* to each $s_1, \ldots s_k$ w/ edges of weight 0

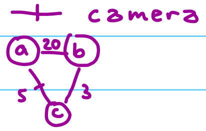
run dijktras from s*

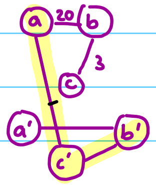
* multiple sources w/ single source R.T.

|V|+1 nodes, |E|+k edges ← k could be all E edges

O(|V|log|V|+ |E| + |V|) → O(|V| log |V| + |E|)

★ ③ duplicate graph





when seeing a camera, go to bottom duplicate graph.

if see another camera on dupe path, don't take it

Dijkstra to find shortest

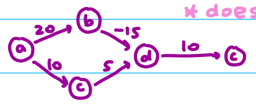
④ set edge weight to 1-p

multiply probabilities (instead of adding edges)

find largest probability

10/29 **PATH RELAXATION LEMMA**

Suppose s→v₁→v₂ ··· vₙ→t is shortest path from s to t. then, if each of these edges is relaxed in order, $d(t) = \delta(t)$ ← will get correct SP @ t



*doesn't hold for negative weights

DAG SP:

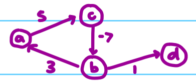
IDEA: use topological order to make sure distances are accurate when using a node

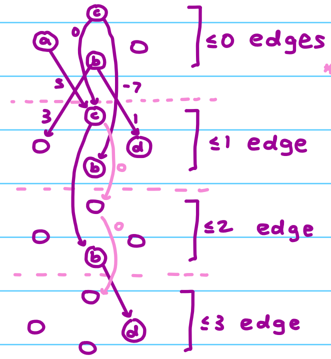
ALG: start distances from ∞. get top. order. w/ full DFS. relax outgoing edges of each
node in this order. $O(|V| + |E|)$

REMINDER relax: if $d[v] > d[u] + w(u,v)$, set $d[v] = d[u] + w(u,v)$

negative edges, cycles... **BELLMON-FORD:** $O(|V||E|)$ ← calculate reachability

INTERP 1: reduction to DAG SP





*can also draw 0-weight edges to itself

*how far we can get after # edges

any shortest path takes @ most |V|-1 edges.

ALG: make |V| layers
run DAG SP - just look @ last layer distance

*not as important INTERP 2: relax every edge v times. eventually will get all paths in order

NEGATIVE CYCLE DETECTION: add another layer

anything that changed has path through negative cycle

- can identify negative cycle by tracing parent pointers
- can DFS to get to all points reachable from these cycles & remove them.

BF & DAG SP on different metrics:

minimize sum	· min, +	$(-\infty, \infty)$	(1) $x \le y$, $x \overset{+}{*} z \le y \overset{+}{*} z$
minimize prod.	· min, ×	$[0, \infty)$	(2) ~~need identity to be best elt~~ need some other best elt.
			(3) need dummy val to be worst elt

Recitation 15

Lecture Summary

- Weighted (and in particular negative-weighted) edges complicate SSSP
- Bellman-Ford approach
- DAG relaxation

Exercises

1. **Round i of Bellman-Ford**

   Let G be a weighted graph with vertices s and v, and assume there is a shortest path from s to v that uses at most i edges. Prove that when running Bellman-Ford, after relaxing the edges into the ith layer of the duplicated graph, the shortest distance to v is already known.

2. **Phunmacks**

   Alice, Bob, and Casey are best friends who live in different corners of a rural school district. One day, they decide to meet at some intersection in the district to play tee-ball. Each child will bike to the meeting location from their home along dirt roads. Each road segment between intersections has a level of **fun** associated with biking along it in a certain direction. Road fun-ness may be positive, but could also be be negative, e.g., when a road is difficult to traverse in a given direction, or passes by a scary dog, etc.

   The children would like to maximize their total fun, which accumulates additively over the road segments they individually bike over. Help the children plan their day by finding an optimal tee-ball location, or return a continuously-fun bike loop in their district, if one exists. You may assume that each child can reach any road in the district by bike.

3. **Ez Money**

   Your friend Mash Coney was shopping online and noticed that someone was selling 3 Super Potions in exchange for 5 Tofu Sandwiches. She saw another deal selling 4 Poké Balls for 1 Super Potion, and another one selling 1 Tofu Sandwich for 2 Poké Balls.

   Mash worked out the math and realized that if she invests 5 Tofu Sandwiches, she can exchange them for 3 Super Potions, exchange those for 12 Poké Balls, and finally get 6 Tofu Sandwiches. She just got a free Tofu Sandwich! Notice that this sort of opportunity would not exist if the second deal were 3 Poké Balls for 1 Super Potion.

Sadly, Mash can only handle the business side[1], so she recruits you for your 6.006 expertise to code an algorithm that will find these arbitrage opportunities from online stores.

Given an array D of n deals, describe an $O(n^2)$ time algorithm to find **any** opportunity that leaves you with more of the same kind of item than you started with. Each deal is of the form (A, x, B, y), indicating that someone will sell you y number of commodity B in exchange for x number of commodity A, where and x and y are positive integers.

Mash is willing to initially invest any amount of any commodity, as long as she gets more of them back. Your algorithm should return an array of commodities in the order you would execute the exchanges (e.g., `[Tofu Sandwich, Super Potion, Poké Ball, Tofu Sandwich]`), or `None` if no such opportunity exists. Briefly justify the correctness of your algorithm, and argue the runtime.

☆ ① PF by IND:

P(i) := for any v s.t. there is a shortest path from s to v with ≤ i edges, after relaxing all edges into level i, $\delta(s_0, v_i) = \delta(s, v)$

B.C. $i = 0 \rightarrow$ SP is just node itself. $\delta(s_0, s_0) = 0 = \delta(s, s)$

IND STEP: assume true for i-1. consider any $s \rightarrow \ldots \rightarrow u \rightarrow v$ $\leq i$ edges.

- if shortest path < i edges, already know $\delta(s_0, v_{i-1}) = \delta(s, v)$ by I.H.
  - relaxing $v_{i-1} \xrightarrow{0} v_i$ keeps it same ✓
- if shortest path takes exactly i edges, consider last edge $u \rightarrow v$

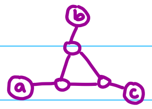
  $s \rightarrow \ldots \rightarrow u$ gives shortest path to u. i-1 len path, so $\delta(s_0, u_{i-1}) = \delta(s, u)$ by I.H.

  1 edge relaxation
  - relaxing $u_{i-1} \rightarrow v_1$ gives $\delta(s_0, v_1) = \delta(s, v)$

② want shorter paths = more fun

edge weights = - fun (negate) * SP alg. finds smallest weights

run BFS from a. check for neg. cycles & return if exist.



otherwise run from b & c for all u,

compute $\delta(a, u) + \delta(b, u) + \delta(c, u)$ & pick smallest u (maximize fun)
can be neg.

[1] https://i.imgur.com/gl3Zc1e.png

③ * manipulate weights to work. want alg. w/ trade route

3 super potions → 5 tofu sandwiches

4 PB → 1 super pot.

1 tofu → 2 PB

$\frac{3}{5} \cdot \frac{4}{1} \cdot \frac{1}{2} = \frac{12}{10} = \frac{6}{5}$ → incr. ← want this to be neg.

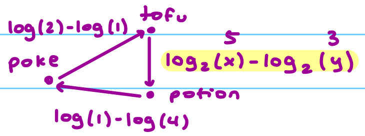
FLIP: $\frac{5}{3} \cdot \frac{1}{4} \cdot \frac{2}{1} = \frac{10}{12}$

*** CYCLE DETECTION!**

$+ \rightarrow \times$

★ LOG TRANSFORMATION

$- \rightarrow \div$



* parent ptrs & storage

cycle: $(\log 5 - \log 3) + (\log 1 - \log 4) + (\log 2 - \log 1)$

$= \log \frac{5}{3} + \log \frac{1}{4} + \log 2$

$= \log\left(\frac{5}{3} \cdot \frac{1}{4} \cdot 2\right) = \log\left(\frac{10}{12}\right) < 1$ → B.F. can find negative cycles now!

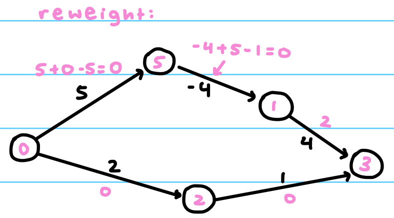
10/31 APSP

	a	b	c	d
a				
b				
c				
d				

GOAL: output a $|V| \times |V|$ table

- if DAG: run DAG SP from each node $O(|V||E| + |V|^2)$ ← SSSP alg. from every node
- if weights, non-negative: run Dijkstras from every node $O(|V| \cdot |E| + |V|^2 \log|V|)$

→ how to make weights non-negative (1) while preserving shortest path (2)?



(1) $\phi(v)$ is some function on V

for each v, subtract $\phi(v)$ from all incoming edges

add $\phi(v)$ to all outgoing edges

all paths from s to t change only by $+\phi(s) - \phi(t)$

(2) set $\phi(v)$ to be SSSP $\delta(v)$ from some start node.

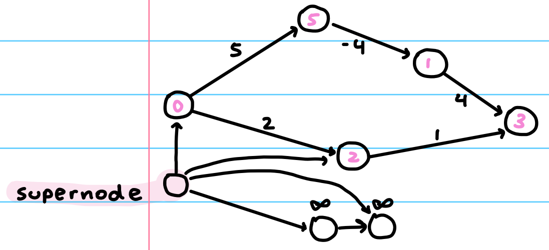
then, new weight for (u,v) is

$w'(u,v) = w(u,v) + \delta(u) - \delta(s)$ non-negate, or else $\delta(v) > \delta(u) + w(u,v)$ & δ is not actually shortest paths

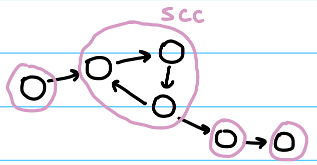
non-neg. b/c SP to u shouldn't be > than SP to v. or else didn't relax correctly.

to get SSSP δ, add supernode w/ edges to all nodes. run BF from it. $O(|V||E|)$

- once reweighted, run Dijkstras from every node $O(|V||E| + |V|^2 \log|V|)$



* use supernode with Bellmon Ford to transform edge weights of all nodes.

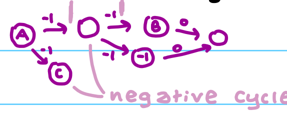


HANDLING NEGATIVE CYCLES:

1) compute SCCs & condensation graph $O(|V| + |E|)$

2) within each SCC in the original graph, run Bellmon Ford to learn which SCCs have negative cycles. $O(|V| \cdot |E|)$

3) weight condensation graph edges s.t. outgoing edges from a negative cycle SCC have weight -1, otherwise edges have weight 0.



any pair before & any pair after negative cycle is still valid (all pairs)

4) get DAG ASAP in condensation graph $\delta_H(A,B)$

- set $\delta_H(A,A) = -1$ if A contains negative cycle

5) create G' by removing all negative SCCs

6) run Johnsons on G' to get $\delta'(u,v)$

7) go through every u,v. let A,B be their SCCs.

if $\delta_H(A,B) < 0$, $\delta(u,v) = -\infty$.

otherwise $\delta(u,v) < \delta'(u,v)$

SUPERNODE: if want distance from ANY special node to each other node

APSP: if want distance between special nodes

Introduction to Algorithms: 6.1210 Recitation 16
Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

Recitation 16

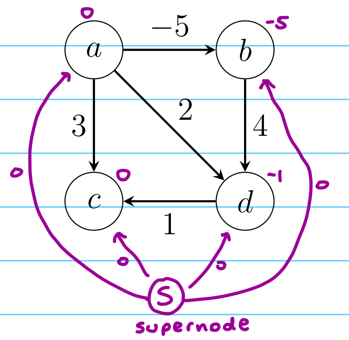
Lecture Summary

- SSSP recap
- APSP via SSSP $|V|$ times
- Johnson's algorithm

Exercises

1. **Johnson's practice**

   Consider the following graph. Show that if we were to increment all edge weights by $\underline{+5}$ to make them non-negative, shortest paths would not be preserved. Then, using a proper reweighting strategy, run Johnson's algorithm to get an APSP matrix.



Shortest paths:

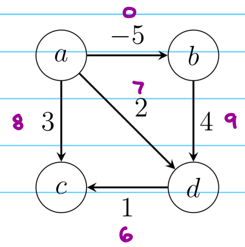
a: s→a 0

b: s→a→b -5

c: s→a→b→c 0

d: s→a→b→d -1

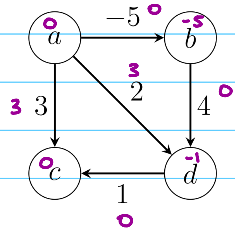
+5:



Ex:

SP a→d = 7 (a→d) NOT preserved.

$w'(u,v) = w(u,v) + \delta(u) - \delta(v)$



	a	b	c	d
a	0	0	0	0
b	∞	0	0	0
c	∞	∞	0	∞
d	∞	∞	0	0

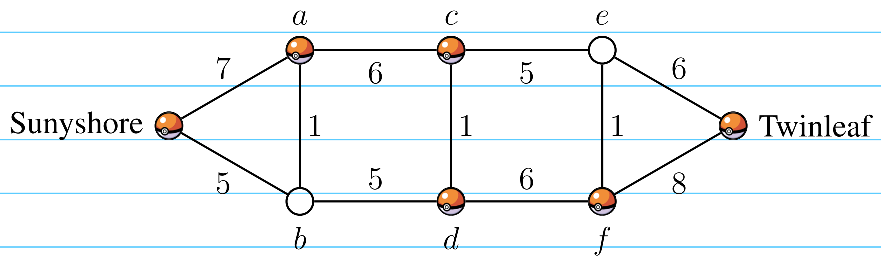
transform back:
$- \phi(s) + \phi(t)$
$\longrightarrow$

	a	b	c	d
a	0	-5	0	-1
b	∞	0	5	4
c	∞	∞	0	∞
d	∞	∞	1	0

2. **Twinleaf Travels**

Ash is walking from Sunyshore City to Twinleaf Town. He wants to take the shortest path, but he can only walk m miles before needing to rest at a Pokémon Center.

The road network is provided as a weighted undirected graph $G = (V, E, w)$ along with the subset $P \subseteq V$ of vertices that have Pokémon Centers. Each weight $w(e)$ denotes the **positive** length in miles of road e. The goal is to find a shortest path from node $s \in V$ to node $t \in V$ that does not travel more than m miles between Pokémon Centers, or report that it is not possible. Assume that $s, t \in P$, and that the graph is a single connected component.



(a) In the graph above, Pokémon Centers are marked with Pokéballs. Find the shortest path from Sunyshore City to Twinleaf Town when $m = \infty$ and also when $m = 10$.

(b) Give an algorithm to solve the general problem with arbitrary m and arbitrary G, and justify its runtime.

3. **Useless Nodes**

Let $G = (V, E)$ be a directed weighted graph with no negative cycles. We say a vertex v is *useful* if there are two vertices $s, t \neq v$ such that v is on a minimum-weight path from s to t. (If there are multiple minimum-weight paths from s to t, v only has to be on one of them.)

Describe an $O(|V|^3)$-time algorithm to determine exactly which vertices of G are useful.

* lots of edges = complete graph $|V|^2$

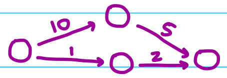
② a) $m = \infty$: $s \rightarrow a \rightarrow b \rightarrow d \rightarrow c \rightarrow e \rightarrow t$

★ $m = 10$:

b) run Johnsons to get $\delta(u,v)$ for every u,v pair *

build G' with just centers, make edge b/t x & y if $\delta(x,y) \leq m$

run Dijkstras on new graph G' b/t s & t.

③ 

* $|V|^2 \cdot |V|$

outer FL: all of $|V|$

inner: all of $|V|^2$

run Johnsons, get APSP for v, for all pairs a,b, check

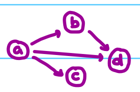
TEST 2 $\delta(a,v) + \delta(v,b) = \delta(a,b)$. if it does for any a,b, v is not useless.

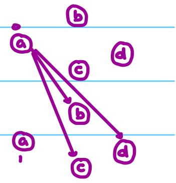
11/5 **EXAM REVIEW:**

DFS:

- topological order (dependencies) for a DAG; cycle detection for non-DAG
- SCCs via KS: useful when there is some equivalence b/t SCCs in original graph.

PROBLEM SOLVING:





- modify graph, not algo
- **GRAPH DUPLICATION**
  - some (constant) state, each level in graph represents different state
  - describe layers & edges
  - run SSSP

SUPERNODE: want to search from many sources as if same "distance from any"

vs.

APSP: need pairwise distances "distance b/t any two"

$\min d(s,u) + d(u,v)_0 + d(v,t)$ for all u, v

↑

EXAM 2

11/5 **GREEDY ALGS:**

- order input in some way
- apply greedy choice
- repeat on remaining items

Ex: activity selection problem (select most non-overlapping items)

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

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

STEPS FOR DESCRIBING GREEY ALG:

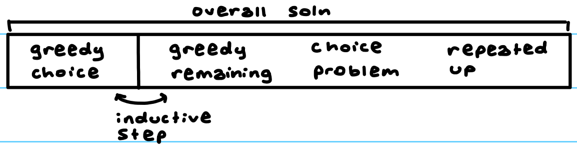
1) **GREEDY CHOICE:** schedule earliest finishing item first

2) **GCP (greedy choice property):** ∃ at least one optimal soln that contains greedy choice

3) **PROOF OF GCP:** compare some optimal S that doesn't make GC, prove you can make it @ least as optimal by swapping to make GC.

4) **PROOF OF ALG BY INDUCTION:** ind. step must argue that remaining problem is of some form (self-reduction) and that solving smaller problems optimally & adding 1st choice back in gives optimal overall

5) Give **ALG & RUNTIME**



Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

Recitation 17

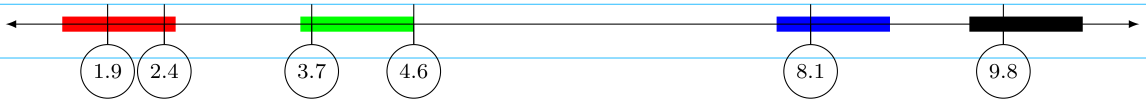
Lecture Summary

1. Greedy Algorithm / Proof Template
   - Identify the *Greedy Choice*
   - Prove the corresponding *Greedy Choice Property*
   - Prove the corresponding *Self-Reduction*
   - Algorithm: order input, then iterate GC
2. Activity Selection Problem
3. Activity Scheduling Problem

Exercises

Exercise 1: Given a set $X = \{x_0 \leq x_1 \leq \cdots \leq x_{n-1}\}$ of points in the real line, find minimum number of intervals of unit length that cover all points (e.g. the interval [1.25,2.25] covers the point 2.0).

As an example consider the points, $\{1.9, 2.4, 3.7, 4.6, 8.1, 9.8\}$. Then, the minimum number of unit intervals is 4 as shown by the red, green, blue, and black intervals as shown in the figure below.



Hint: What observation can you make of the optimal solution?

Exercise 2: We are given n processes. Each process i takes t_i time to complete. We need to schedule processes sequentially, i.e., find a permutation σ such that process $\sigma[1]$ is scheduled before $\sigma[2]$, which is scheduled before $\sigma[3]$, and so on. The finishing time of process $\sigma[i]$ is then

$$C_{\sigma[i]} = \sum_{j=1}^{i} t_{\sigma[j]}$$

Design an algorithm that finds a permutation σ such that the average finishing time is minimized. The average finishing time is defined as

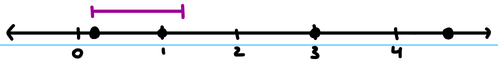
$$\frac{1}{n} \sum_{i=1}^{n} C_{\sigma[i]}$$

Exercise 3: You have come across a treasure consisting of several different types of precious metals. The treasure chest holds containers of n metals, where the i-th container has weight w_i and total value v_i. Unfortunately, you can only carry a total weight of W - otherwise, you would just take the entire treasure chest with you! Determine how much weight x_i of metal you will take from each container (note that necessarily $0 \leq x_i \leq w_i$) in order to maximize the total value of the treasure you take.

For example, if $W = 2, w_1 = 2, v_1 = 2, w_2 = 1, v_2 = 1$, then you could take 1.5 unit weight of metal 1 and 0.5 unit weight of metal 2, for a total value of 2.

Exercise 4: Given a value V, if we want to make a change for $\$V$, and we have an infinite supply of each of the denominations in currency, i.e., we have an infinite supply of { \$1, \$5, \$10, \$25 } valued coins/notes, what is the minimum number of coins and/or notes needed to make the change?

①



1) GC: choose interval starting from 1st point: $[x_0, x_0+1]$

2) GCP: there is an optimal soln including $[x_0, x_0+1]$

3) Proof of GCP: assume S is optimal & doesn't use $[x_0, x_0+1]$

must have some interval covering x_0, call this I_0

can replace I_0 with $[x_0, x_0+1]$

4) Proof by Strong Induction on # of points:

assume S is optimal. G is soln from greedy. WTS: S & G are same.

by GCP, can assume S & G have same 1st interval (start the same)

on remaining points, greedy is optimal $|S'| = |G'|$

$|S| = |S'|+1$ size = # of intervals

$|G| = |G'|+1$ adding back our interval in beginning

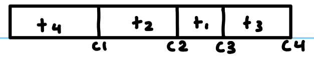
$|S| = |G|$ ■

5) count = 0 $I_0 = [x_0, x_0+1]$, remove points from beginning that overlap

count += 1, repeat w/ remaining x.

RT: alg. scans input list (sorted as given) → $O(n)$

CORRECTNESS: ... strong ind.

② 

σ $\begin{array}{|c|c|c|c|} \multicolumn{1}{c}{1} & \multicolumn{1}{c}{2} & \multicolumn{1}{c}{3} & \multicolumn{1}{c}{4} \\ \hline 4 & 2 & 1 & 3 \\ \hline \end{array}$

min $\frac{1}{n} \sum_i c_i$

1) GC: schedule k w/ min t_k first

2) GCP: there is an optimal soln w/ min t_k 1st

3) let S be opt. schedule w/ $S[1] = j \neq k$. make S' that swaps j w/ k

everything after both stays same.

everything between can only decrease

☆ 4) let S be optimal

G be from Greedy

by GCP, can assume S & G schedule same first $S[1] = G[1] = k$

consider S' & G' $\sum_i C_{S'_i} = \sum_i C_{G'_i}$

$\sum_i C_{S_i} = t_k \cdot n + \sum_i C_{S'_i} = t_k \cdot n + \sum_i C_{G'_i} = \sum_i C_{G_i}$

5) sort processes by finish time & return sorted array

RT: O(nlogn) sort, return pointer to array → O(nlogn)

CORRECTNESS: . . . strong ind

③ 1) GC: first take as much of most valuable per unit weight (v_i / w_i)

2) GCP: ∃ optimal soln that takes as much as possible from the metal w/ max v_i/w_i

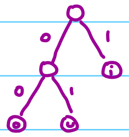
☆ 3) Proof of GCP: ??

↓ correctness, etc.

11/14

HOFFMAN'S ALG: encode letters as binary strings

a:5 e: 10 i: 3 $\underbrace{o: 2 \quad u: 1}_{+= 3}$



GREEDY CHOICE: can pick the smallest frequencies as siblings

GREEDY ALGS:

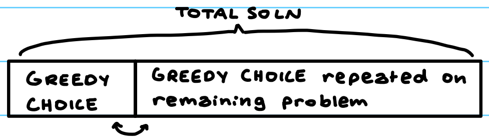
1: Greedy Choice

2: GCP (exists an optimal soln. that makes GC)

3: Proof of GCP (can make 1st choice Greedy, prove w/ swapping arg.)

4: Proof by induction (compare greedy to optimal)

5: Alg. & Runtime



INDUCTION:

① you can make greedy choice & be left with a problem of same form

② Combining w/ smaller problem given optimal soln. overall

Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

Recitation 18

Lecture Summary

1. Prefix Free Codes
2. Huffman Algorithm

Exercises

1. Alice wants to throw a party and she is trying to decide who to invite. She has n people to choose from, and she knows which pairs of these people know each other. She wants to invite as many people as possible subject to the constraint:

   For each guest, there should be at least five other guests that they already know.

   Describe and analyze an $O(n^2)$-time algorithm that computes the largest possible number of guests Alice can invite, given a list of pairs, where a pair (i, j) represents guest number i and guest number j knowing each other.

2. A *wiggle sequence* is a sequence where the differences between successive numbers strictly alternate between positive and negative. The first difference (if one exists) may be either positive or negative. A sequence with one element and a sequence with two non-equal elements are trivially wiggle sequences. For example, the array $[1, 7, 4, 9, 2, 5]$ is a wiggle sequence, but the array $[1, 7, 4, 5, 7]$ is not.

   Given a integer array A of length $n > 0$ with no consecutive equal elements, we want to find the length of the longest wiggle subsequence.[1]

   (a) Prove that there is an optimal solution (i.e. max-length wiggle subsequence) that includes $A[0]$.

   (b) We say that *A starts by increasing* if $A[0] < A[1]$ or A has only one element. Prove that if A starts by increasing, every optimal solution which includes $A[0]$ also starts by increasing.

   (c) Give a greedy algorithm to compute the longest wiggle subsequence, prove its correctness, and analyze its runtime.

[1] A *subsequence* of A contains a subset of the elements of A, in the order they appear in A, but not necessarily consecutive.

① 1) GC: pick anyone who doesn't know ≥ 5 ppl. remove them.

2) GCP: ∃ optimal soln. that makes the GC.

3) Proof of GCP: assume an opt. soln. that doesn't exclude person p w/ < 5 friends.

→ this is already a contradiction →←

5) make graph. go through each node & delete if find one w/ < 5 edges.

keep doing this until all nodes have ≥ 5 edges. $O(n^2)$ n: # ppl

4) Pf by Ind on # ppl:

assume opt. soln. S & greedy alg G

by GCP, assume both remove same 1st person.] how can we just assume?

by IH, greedy is opt. on remaining soln.

☆ ② A = [1, 3, 4, 2, 8, 9, 5]

S: [1, 4, 2, 9, 5]

a) BC: if length(A) ≤ 2 → optimal S must choose A[0]

otherwise, if len > 2, assume $[S_1, S_2, \dots, S_n]$ doesn't use A[0].

WLOG, assume $S_1 < S_2$ (incr. @ beginning) ★ do we need to make a case for when $A[0] > S_1$? yes!

IF $A[0] \le S_1$: replace S_1 w/ A[0]

IF $A[0] > S_1$: put A[0] in front, sequence becomes longer

★ we don't have to argue for case when they're = ?

b) A: [3, 4, 5, 2, 8, 9, 5]

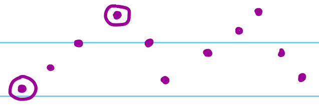
S: [3, 5, 2, 9, 5]

assume A[0] < A[1]. some optimal S starts w/ A[0] & starts by decr.

could just add A[1] after A[0] →← contradicts decreasing

c) 0: select A[0].

1: GC: select the 1st peak ($i > 0$ s.t. $A[i-1] < A[i] > A[i+1]$ or $A[i-1] > A[i] < A[i+1]$)



← pick the peak b/c it'll give you more choices later on

3: WLOG S starts by increasing

Pf of GCP: assume opt. S doesn't pick such an i for its 1st choice.

$[\overset{A[0]}{S_0}, \overset{GC}{S_1}, \dots, S_n]$

call index of S_1 j

$S_0 < S_1$ (part b)

S_2 must come from an index after i

CASE 1: $A[j] \le A[i]$ → replace (just pick the bigger one)

CASE 2: $A[j] > A[i]$ → can add A[i] & A[i+1] to soln to get better soln.

11/19 **DYNAMIC PROGRAMMING**

EX: COIN ROW

[5, 1, 2, 10, 6, 2]

v1: **Recursion** $\max \begin{cases} \text{coins}([1,2,10,6,2]), \\ 5+\text{coins}([2,10,6,2]) \end{cases}$

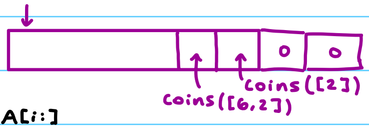
v2: **Memozation** (top down)

- store answers in DAA
- recursive calls don't repeat work

v3: **Pure DP (top down)**

- start from smaller problems
- build in reverse topological order

SRTBOT: follow this format! x = DP table



SUBPROBLEM: $x(i)$ = max value you can get from $A[i:]$

RELATE: relate $x(i) = \max\{x(i+1), A[i] + x(i+2)\}$

TOPOLOGICAL: $x(i)$ depends on strictly larger i

BASE CASE: $x(n-1) = A[n-1]$
$x(n-2) = \max\{A[n-2], A[n-1]\}$] bounding the problem

OUTPUT: $x(0)$

RUNTIME: $O(n)$

Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

Recitation 19

Lecture Summary

- Exponential recursive approach
- Memoization so we don't keep recomputing the same thing
- Implementing the same solution without recursion
- SRTBOT framework

Exercise 1: Simplified Blackjack

We define a simplified version of the game **blackjack** between one **player** and a **dealer**. A **deck of cards** is an ordered sequence of n cards $D = (c_0, \ldots, c_{n-1})$, where each card c_i is an integer between 1 and 10 inclusive.

Blackjack is played in **rounds**. In one round, the dealer will draw the top two cards from the deck (initially c_0 and c_1), then the player will draw the next two cards (initially c_2 and c_3), and then the player may either choose to draw or not draw one additional card.

The player wins the round if the **value** of the player's hand (i.e., the sum of cards drawn by the player in the round) is more than the value of the dealer's hand, and at most 21. The game ends when a round ends with fewer than 5 cards remaining in the deck.

Given a deck of n cards with a **known order**, describe an $O(n)$-time algorithm to determine the maximum number of rounds the player can win by playing simplified blackjack with the deck.

Ex: $[\overbrace{1 \quad 5}^{\text{dealer}} \quad 6 \quad 7 \quad 8 \quad 3 \quad 4 \quad 5]$

$w(d, p) = \begin{cases} 1 & \text{if win} \\ 0 & \text{if lose} \end{cases}$

S: $x(i)$ = max # rounds that can be won starting @ i

R: $x(i) = \max \begin{cases} w(c_i + c_{i+1}, c_{i+2} + c_{i+3}) + x[i+4] & \leftarrow \text{whether we won when not taking extra card} \\ w(c_i + c_{i+1}, c_{i+2} + c_{i+3} + c_{i+4}) + x[i+5] & \leftarrow \text{if we win when drawing additional card} \end{cases}$

T: $x(i)$ only depends on strictly greater. $x(i)$ depend on $x(j)$ where $i < j$

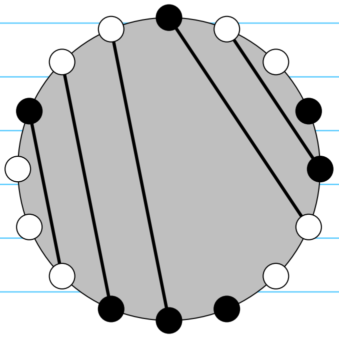
B: $i > n-5 \rightarrow x(i) = 0$

O: $x(0)$

T: $O(n)$ subproblems, each takes constant time

Exercise 2: Wafer Power

A start-up is working on a new electronic circuit design for highly-parallel computing. Evenly-spaced along the perimeter of a circular wafer sits n ports for either a power source or a computing unit. Each computing unit needs energy from a power source, transferred between ports via a wire etched into the top surface of the wafer. However, if a computing unit is connected to a power source that is too close, the power can overload and destroy the circuit. Further, no two etched wires may cross each other. The circuit designer needs an automated way to evaluate the effectiveness of different designs, and has asked you for help. Given an arrangement of power sources and computing units plugged into the n ports, describe an $O(n^3)$-time dynamic programming algorithm to match computing units to power sources by etching non-crossing wires between them onto the surface of the wafer, in order to maximize the number of powered computing units, where wires may not connect two adjacent ports along the perimeter. Below is an example wafer, with non-crossing wires connecting computing units (white) to power sources (black).



A= (0, 0, 1, 1, 0, 0)

x(0, n) ← includes all pts

1: x(1, n) ← don't connect 0 w/ anything

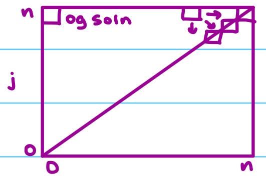
2: 1 + x(1, k) + x(k+1, n) ← connect 0 w/ some k

S: $x(i, j) = \max$ # connections among ports from $[i, j)$

R: $x(i,j) = \max \left(\begin{matrix} x(i+1, j), \quad O(n) \\ 1 + x(i+1, k) + x(k+1, j), \quad i+2 \leq k < j,\ A[0] \neq A[k] \end{matrix} \right.$

$x(0,m) = \max \left(\begin{matrix} \{x(1,n)\} \\ \cup \{1 + x(1,k) + x(k+1, n) \mid 2 \leq k \leq n-1,\ A[0] \neq A[k]\} \end{matrix} \right)$ ← have to handle b/c circular

T: $x(i,j)$ only depends on $x(i', j')$ where $j' - i' < j - i$



O(n²) fill in

T: $O(n^3)$ → each step $O(n)$, $O(n^2)$ total fill in

11/21 **LONGEST COMMON SEQUENCE**

S: $T(i,j) = $ len of LCS of $A[i:n]$, $B[j:m]$

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

THEIR → HI

HABITTS → ABT

	T	H	E	I	R	
H						0
A						0
B						0
I				1	0	0
T				0	0	0
S				0	0	0
	0	0	0	0	0	

LONGEST INCREMENTING SUBSEQUENCE:

[8, 5, 10, 6, 7, 9] → [5, 6, 7, 9]

S: $T(i,j) = $ len of LIS of $A[i:n]$ w/ $\begin{pmatrix} 0 \le i \le n \\ -1 \le j \le i-1 \end{pmatrix}$ numbers larger than $A[j]$

R: $T(i,j) = \max \begin{cases} 1 + T(i+1, j) \text{ if } j = -1 \text{ or } A[i] > A[j] \\ T(i+1, j) \end{cases}$

:

O: $T(0, -1)$

Alternate:

S: $T(i) = $ len of LIS of $A[i:n]$ using $A[i]$

R: $T(i) = 1 + \max\limits_{i < j < n} \{ T(j), A[i] < A[j] \}$

Introduction to Algorithms: 6.1210 Recitation 20
Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

Recitation 20

Lecture Summary

- Longest Common Sub-Sequence (LCS) problem
- Longest Increasing Sub-Sequence (LIS) problem

Exercise 1: Edit Distance

A plagiarism detector needs to detect the similarity between two texts, string A and string B. One measure of similarity is called **edit distance**, the minimum number of **edits** that will transform string A into string B. An edit may be one of three operations: delete a character of A, replace a character of A with another letter, and insert a character anywhere in A. Describe a $O(|A||B|)$ time algorithm to compute the edit distance between A and B.

Exercise 2: Arithmetic Parenthesization

You are given an arithmetic expression without any parentheses: $a_0 *_1 a_1 *_2 a_2 \cdots *_{n-1} a_{n-1}$, where each a_i is an integer and each operator $*_i \in \{+, \times\}$. Describe a dynamic program to place parentheses (with correct syntax) to maximize the value of the resulting expression. For example:

$$8 + 4 \times 3 + 5 \quad \rightarrow \quad (8+4) \times (3+5) \quad = \quad 96$$
$$8 + (-4) \times 3 + (-5) \quad \rightarrow \quad 8 + \big((-4) \times (3 + (-5))\big) \quad = \quad 16$$

① HATED WAITED
↓
WATED
↓
WAITED

S: $x(i,j) = \min$ # edits to turn $A[i:]$ to $B[j:]$, $0 \le i \le |A|$, $0 \le j \le |B|$

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

⋮

O: $x(0, n \text{ max})$

T: $O(n^2)$ subproblems, $O(n)$ per subproblem → $O(n^3)$

② $(a_0 *_1 a_1 *_2 a_2)(\cdot *_{n-1} a_{n-1})$

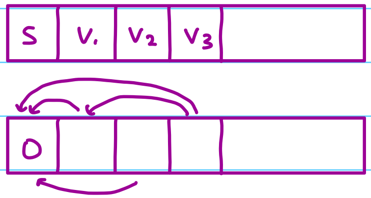
Hint 1: $x(i,j)$ depends on $x(i,k)$, $x(k,j)$ for b/t i & j

Hint 2: build 2 seperate tables for min & max. they will use each other

S: $x(i, j, opt) =$ opt value attainable from $a_i *_{i+1} a_{i+1} \ldots *_{j-1} a_{j-1}$ $0 \le i \le j \le n$ $opt \in \{\min, \max\}$

11/26 DP w/ shortest paths

DAG SP:



*solve going right

S: $x(v)$ = distance to v

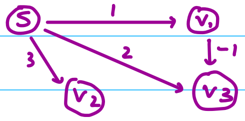
R: $x(v) = \min\{x(u) + w(u,v) \mid u,v \in \mathbb{R}\}$

T: reverse top order of G

B: $x(s) = 0$

O: whole table

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

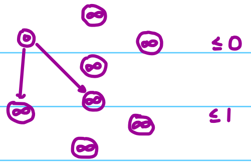


BF DP:

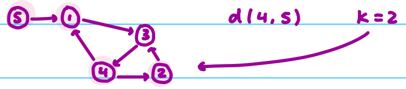
S: $x(v,k)$: weight of SP from s to v using ≤ k edges

R: $x(v,k)$: $\min\begin{cases}\min\{x(u,k-1) + w(u,v) \mid u,v \in E\\ x(v,k-1)\end{cases}$

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



FLOYD WARSHALL: APSP



$v = \{1, 2, \dots, n\}$ arbitrary ordering

S: $d(u,v,k)$ = SP using only vertices in $\{u,v\} \cup \{1,2,\dots,k\}$

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

T: $O(|V|^3) \cdot O(1)$

GENERAL DP TIPS:

- define a subproblem that captures what you need (prefix/suffix, picking a pt to slice)
- try to make recursive relation

0

Introduction to Algorithms: 6.1210 Recitation 21
Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

Recitation 21

Lecture Summary

1. Multiple examples to solve shortest path using Dynamic Programming.
2. SSSP in DAG using Dynamic Programming.
3. Bellman Ford using Dynamic Programming.
4. New algorithm for APSP: **Floyd–Warshall**. It takes in $O(|V|^3)$ time and matches Johnson's runtime for dense graphs.

Exercise 1: Bowling along Shortest Paths

There are n bowling pins in a row, and the ith pin has an associated value v_i for $i \in \{1, \ldots, n\}$. You can knock down pins individually or in pairs: knocking pin i down individually earns v_i points, and knocking pins $i-1$ and i down together earns $v_{i-1} \cdot v_i$ points. A pin can only be counted once, and you don't have to knock down all the pins (maybe $v_i < 0$).

a) Design a DP algorithm to find the maximum possible score, and analyze its runtime.

b) Design an algorithm that solves the same problem by constructing a graph and then running a single source shortest path algorithm. Analyze its runtime.

Exercise 2: Counting Shortest Paths

Given an undirected weighted graph $G = (V, E, w)$ with n vertices and $m \geq n$ edges, and a vertex s, describe an $O(nm)$ time algorithm to compute the number of shortest paths from s to each other vertex $v \in V$. Assume that all cycles have positive weight, but note that edges may have negative weight.

Exercise 3: Jogging on a Budget

Your new fitness tracker has been helping you plan a jogging route through town. Starting at home s, you want to run **exactly** 5 miles, after which you will be too tired to jog and will call your favorite cow-themed ride-sharing service Uder to take you back home.

The tracker has a built-in town map with n intersections and m two-way road segments $\{u, v\}$ of length $w(u, v)$, which is always a multiple of 0.1 miles. Using GPS, the innovative device only updates your jogging progress whenever you fully traverse a road segment from one intersection to the other. The tracker also has the Uder app, listing the price of being picked up at an intersection v as p_v Moo-lah.

Design an $O(n + m)$ time dynamic program to choose a jogging route that minimizes the amount of Moo-lah you will have to spend to ride home after running exactly 5 miles.

① a) S: $T(i)$ = max # of points we can get from $A[0:i]$, $0 \le i \le n$

R: $T(i) = \max \begin{cases} T(i-1) + A[i] & \leftarrow \text{add} \\ T(i-2) + A[i] \cdot A[i+1] & \leftarrow \text{multiply} \\ T(i-1) & \leftarrow \text{move onto next step} \end{cases}$

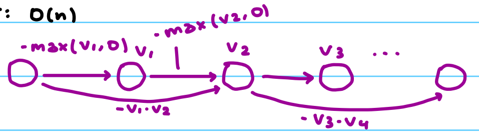
T: decreasing

B: $T(0) = 0$, $T(1) = \max\{0, A[0]\}$

O: $T(n)$

T: $O(n)$

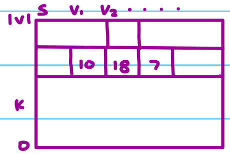
b)



graph v_k: max points we could've gotten @ k.

• can choose not to pick the vertex → $-\min(0, v_1)$

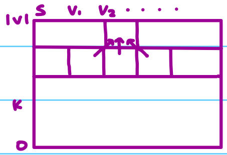
run DAG SP



only consider the paths that create the SPs

② S: 1) $\overbrace{y(v,k)}^{\delta_k(v)}$ = weight of SP from s to v using EXACTLY k edges

2) $x(v,k)$ = # shortest paths from s to v using exactly k edges



R: $\delta_k(v) = \min\{\min\{y(u, k-1) + w(u,v) \mid (u,v) \in E$

$x(v,k) = \sum x(u, k-1) \mid u, v \in E$ & $\overbrace{\delta_{k-1}(u) + w(u,v) = \delta_k(v)}^{\text{if it satisfies our } k}$

T: $\delta_k(v)$ depends on $\delta_{k-1}(u)$, $x(v,k)$ depends on $\delta_{k-1}(u)$, $\delta_k(v)$, $x(u, k-1)$

B: $k=0$: $\delta_0(s) = 0$, $\overset{\text{others}}{\delta_0(v) = \infty}$

$x(s,0) = 1$, $\underset{\text{others}}{x(v,0) = 0}$

SP to itself ↗

12/3 **PSEUDOPOLYNOMIAL DP**

SANDWICH CUTTING:

S: $s(\ell)$ = max val atainable for sandwich of length ℓ

R: max $v(i) + S(\ell - i)$

T: decreasing i

B: $S(0)=0$

O: $S(L)$

T: L subproblems, each $O(L)$ → $O(L^2)$

S

1: 8
2: 11
3: 13
4: 20
5: 25

SUBSET SUM: $A=[a_0, \dots, a_{n-1}]$ positive ints

Ex: $A: [2,5,7,8,9]$ $L=21$
$L=25$ x

- is there a subset of A that sums to L

S: $x(\ell, i)$ = does subset of $A[i:]$ sum to ℓ? Y/N (decision problem)

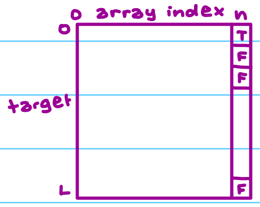
R: $x(\ell,i) = \text{OR}\begin{cases} x(\ell, i+1) & \text{move on \& don't take} \\ x(\ell - a_i, i+1) \text{ if } a_i \le \ell & \text{move on \& take it} \end{cases}$

T: increasing i

B: $x(\ell, n) = \begin{cases} \text{True if } \ell = 0 \\ \text{False o.w.} \end{cases}$

O: $x(L, 0)$

T: $\Theta(nL)$



*when depending on input value, can become big

→ PSEUDO POLY

POLYNOMIAL TIME:

time ≤ O(1) degree polynomial in input size (measured in words)

PSEUDO POLYNOMIAL TIME:

time ≤ O(1) degree polynomial in input size & integer inputs (actual R.T. can end up exponential)

Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

Recitation 22

Lecture Review

- Polynomial vs Pseudo-Polynomial Time
- Sandwich Cutting
- Subset Sum
  - Input: Set of n positive integers $A[1, \ldots, n]$ and a target T.
  - Output: Is there subset $A' \subset A$ such that $\sum_{a \in A'} a = T$?
  - Can solve with dynamic programming in $O(nT)$ time
  - The algorithm is only efficient in the special case where T is polynomial in the length of the input, but in general it could be exponential.

Exercises

We'll look at several problems related to Subset Sum.

1. Describe an algorithm to solve Partition: Given a set of n positive integers A, determine whether A can be partitioned into two subsets A_1 and A_2 of equal sum.

   For example, $A = \{1, 3, 4, 12, 19, 21, 22\}$ can be partitioned into $A_1 = \{1, 19, 21\}$ and $A_2 = \{3, 4, 12, 22\}$, which both have sum 41.

2. Adapt the Subset Sum algorithm to work on sets containing negative numbers.

3. **0-1 Knapsack.**

   We are given a capacity S and a list of n items, which each have a *size* s_i and a *value* v_i. The goal is to find the subset of items with total size at most S that has maximum total value. As before, "0-1" means that we can take each item only zero or one times; there's only a single instance of each item.

   Describe an $O(nS)$ time algorithm to solve 0-1 Knapsack.

   We've made CoffeeScript visualizers solving Subset Sum and 0-1 Knapsack:

   `https://codepen.io/mit6006/pen/JeBvKe`
   `https://codepen.io/mit6006/pen/VVEPod`

① run Subset sum on A with target $T = \frac{\sum_i A_i}{2}$

if true, then exist 2 halves

② S: $x(\ell, i)$ = some subset of $A[i:]$ sums to ℓ, $0 \le i \le n$, $\overset{\text{sum of negative } a_i}{\ell_{min}} \le \ell \le \overset{\text{sum of positive } a_i}{\ell_{max}}$

R: $x(\ell, i) = \text{OR}\begin{cases} x(\ell, i+1) \\ x(\ell - a_i, i+1) \text{ if } \ell_{min} \le \ell - a_i \le \ell_{max} \end{cases}$

same as subset sum problem

T: increasing i

B: $x(\ell, n) = \begin{cases} \text{True if } \ell = 0 \\ \text{False o.w.} \end{cases}$

O: $x(L, 0)$

T: $t_{max} - t_{min}$ = range of table we must cover = $\sum |a_i| = S$ → $O(nS)$

③ 0-1 knapsack: size S s.t. can take items up to this size

max val of items

Ex: $s = [5, 10, 12, 7, 6]$
$v = [10, 18, 4, 8, 6]$

S: $x(s, i)$ = max val we can pack into knapsack of size s using items starting @ i.

$0 \le i \le n, \quad 0 \le s \le S$

R: $x(s, i) = \max\begin{cases} v_i + x(s - s_i, i+1) \text{ if } s \ge s_i \leftarrow \text{take item} \\ x(s, i+1) \end{cases}$

T: increasing i

B: $x(s, n) = 0$ (empty suffix)

O: $x(S, 0)$

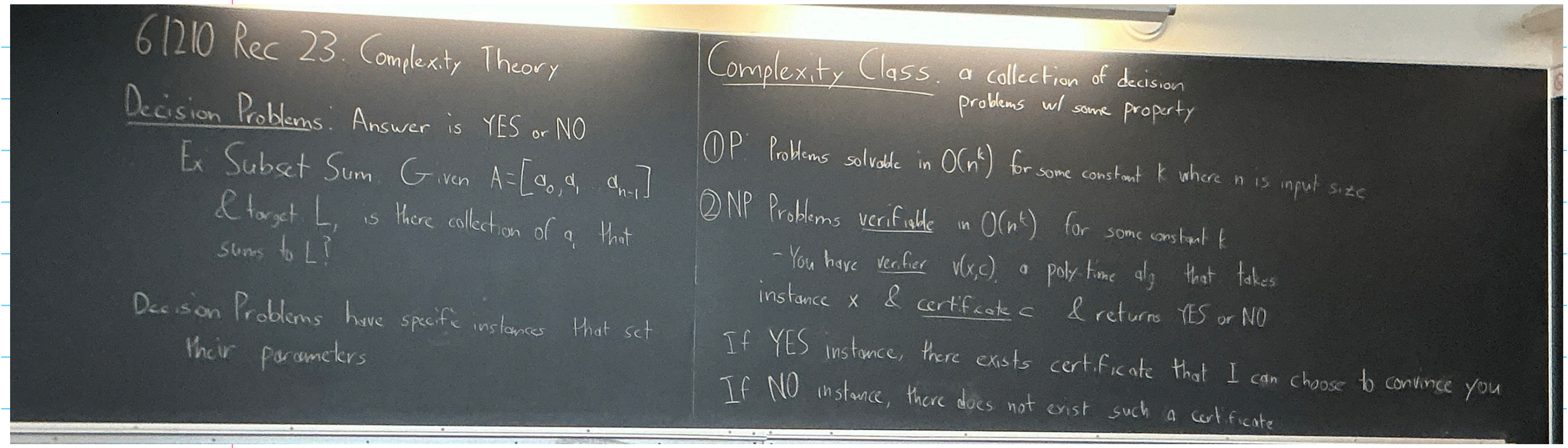
T: $O(nS)$

4. **Close Partition. (optional)** Given a set of n positive integers A, describe an algorithm to partition A into two subsets A_1 and A_2 such that the absolute difference $\left|\sum A_1 - \sum A_2\right|$ between their sums is minimized.

5. **(optional)** Solve Close Partition (from the previous problem) by turning it into an instance of 0-1 Knapsack. In other words, describe a polynomial-time reduction from Close Partition to 0-1 Knapsack.

6. **Unbounded Knapsack. (optional)** Unbounded Knapsack is the same as 0-1 Knapsack, except that there are many copies of each item available—you can take as many as you like. Design an algorithm to solve this problem.

12/5



6.1210 Rec 23. Complexity Theory

Decision Problems: Answer is YES or NO

Ex Subset Sum: Given $A = [a_0, a_1, \ldots a_{n-1}]$ & target L, is there collection of a_i that sums to L?

Decision Problems have specific instances that set their parameters

Complexity Class: a collection of decision problems w/ some property

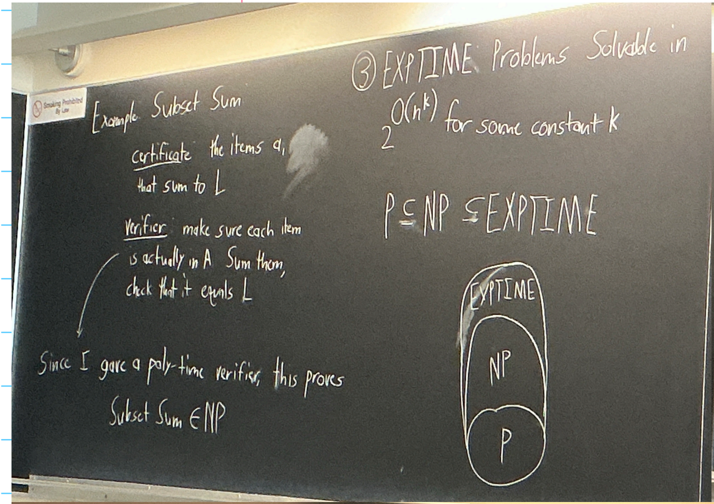
① P: Problems solvable in $O(n^k)$ for some constant k where n is input size

② NP: Problems verifiable in $O(n^k)$ for some constant k

- You have verifier $v(x,c)$, a poly-time alg that takes instance x & certificate c & returns YES or NO

If YES instance, there exists certificate that I can choose to convince you

If NO instance, there does not exist such a certificate



Example: Subset Sum

certificate: the items a_i that sum to L

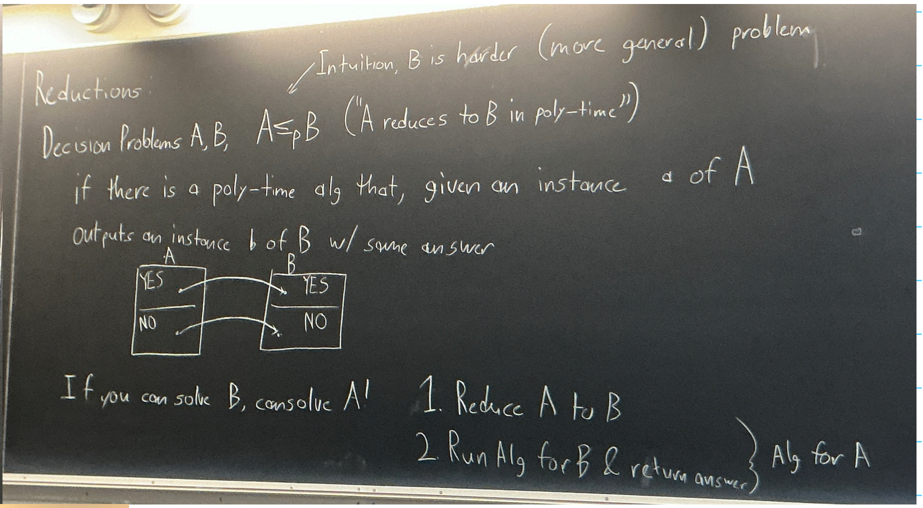
verifier: make sure each item is actually in A. Sum them, check that it equals L

Since I gave a poly-time verifier, this proves Subset Sum ∈ NP

③ EXPTIME: Problems Solvable in $2^{O(n^k)}$ for some constant k

$P \subseteq NP \subseteq EXPTIME$

EXPTIME
NP
P



Reductions:

Intuition, B is harder (more general) problem

Decision Problems A, B, $A \leq_p B$ ("A reduces to B in poly-time")

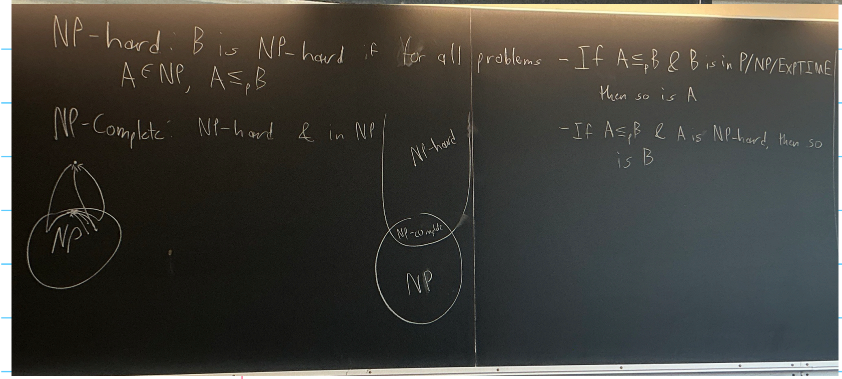
if there is a poly-time alg that, given an instance a of A outputs an instance b of B w/ same answer

A
B
YES
NO
YES
NO

If you can solve B, can solve A!

1. Reduce A to B
2. Run Alg for B & return answer

} Alg for A



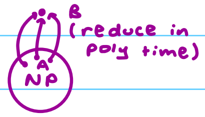
NP-hard: B is NP-hard if ∀ for all problems $A \in NP$, $A \leq_p B$

NP-Complete: NP-hard & in NP

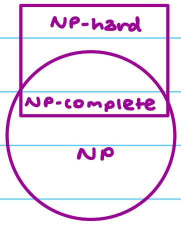
NP
NP-hard
NP-complete
NP

- If $A \leq_p B$ & B is in P/NP/EXPTIME then so is A

- If $A \leq_p B$ & A is NP-hard, then so is B



(reduce in poly time)



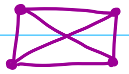
Introduction to Algorithms: 6.1210 Recitation 23
Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

Recitation 23

Lecture Summary

- Decision problems and problem classes
- Nondeterministic Polynomial Time NP
- Reductions
- NP-hardness

Exercises



1. In an undirected graph, a *clique* is a set of vertices such that every pair of them is connected by an edge. Given an undirected graph $G = (V, E)$ and an integer k, CLIQUE asks whether G has a clique of size k.

   Show that CLIQUE is in NP by designing a verifier for it: specify what the certificates represent, describe the verifier itself, analyze its runtime, and argue that the verifier takes polynomial time in n, the size of the input.

2. Consider two decision problems problems A and B. Assume there is a polynomial time reduction from A to B which takes an instance of A of size n and transforms it into an instance of B of size $O(n^2)$. Assume that B is in NP, and instances of B of size m have certificates of length $O(m^3)$. Circle **all** necessarily true statements:

   $a \xrightarrow{A \le_p B} b$
   n n^2
   $(n^2)^3 = n^6$

   (a) Instances of A of size n have certificates of length $O(n^2)$.

   (b) Instances of A of size n have certificates of length $O(n^3)$.

   [✓ (c)] Instances of A of size n have certificates of length $O(n^6)$.

   (d) B is NP-complete. X → doesn't mention NP-hard

   [✓ (e)] A can be solved in EXPTIME. → problems in NP are also in Exptime

① CLIQUE: does there exist fully connected graph of size k

CERTIFICATE: set U of k nodes that forms a clique

VERIFIER: check that |U| = k & check for all u,v ∈ U, u,v ∈ E

O(k²) ∈ O(n²) → polynomial time

☆ 3. For decision problems A B, and C, assume $A \leq_P B$ and $B \leq_P C$. If B is NP-Complete, select **all** statements that must be true:

$a \longrightarrow b$
$n \quad n? \quad n^{100} \quad (n^{100})^{15}$

[✓ (a)] $A \leq_P C$. ✓ reductions are transitive!

[✓ (b)] A is in NP. → B is NP-complete (NP hard & in NP), so A can reduce and be in NP

(c) A is NP-Hard. ✗ can't assume anything about $A \leq_P B$

(d) C is in NP. ✗ C may be in ✗ time, but we don't know if C ∈NP. could be smthg further out

[✓ (e)] C is NP-Hard. can infer hardness for $A \leq_P B$

4. Consider two decision problems A and B. The problem $A \cup B$ asks whether its input is a YES instance of A *or* a YES instance of B. Similarly, the problem $A \cap B$ asks whether its input is a YES instance of A *and* a YES instance of B.

Circle all necessarily true statements.

[✓ (a)] If $A, B \in$ P, then $A \cup B \in$ P. YES if poly time solveable, can just solve both

[✓ (b)] If $A, B \in$ P, then $A \cap B \in$ P. YES solve both

[✓ (c)] If $A, B \in$ NP, then $A \cup B \in$ NP. YES: certificate tells us which problem to use & has certificate for that problem.

☆ [✓ (d)] If $A, B \in$ NP, then $A \cap B \in$ NP. YES

5. Recall that in PARTITION, we are given a list of numbers A and asked whether it can be partitioned into two lists with the same sum. This problem is NP-complete.

We can define a decision problem 0-1 KNAPSACK as follows: we are given a capacity S, a target value V, and a list of items, which each have a *size* s_i and a *value* v_i. We are asked whether there's a subset of items with total size at most S and total value at least V. (The "0-1" in the name comes from the fact that each item can be taken zero or one times, but not multiple times or a fractional value.)

Prove that 0-1 KNAPSACK is NP-hard by describing a reduction from PARTITION to 0-1 KNAPSACK.

(5) construct reduction from PARTITION $\leq_p$ 0-1 KNAPSACK

0-1 knapsack

s_0	s_1	
v_0	v_1	

does there exist subset w/ size @ most S & value @ least V

for every elt a_i in A, create item i where $s_i = v_i = a_i$ (set s_i & v_i = to a_i)

$S = V = \frac{\Sigma a_i}{2}$

YES instance partition ⟶ YES 0-1 knapsack

NO partition → NO 0-1 knapsack

YES instance knapsack → YES partition ← contrapos.

12/10 **COMPUTABILITY THEORY**

basics
- algorithms (programs) are written in binary & take some arbitrary binary input
- alg. on particular input can answer YES, NO, or loop forever

HALTING PROBLEM:

input: $(\underset{\text{alg}}{P}, \underset{\text{input}}{x})$

out: YES if P halts on x

NO otherwise

UNDECIDEABLE: problem is undecideable if there is no alg that you can make that solves this problem

THM: HALT is undecideable (prove w/ Diagonalization)

TOTALITY:

input: P ← alg

out: YES if P halts on all inputs

NO otherwise

THM: totality is undecideable

Pf: HALT $\leq_p$ TOTALITY ← reduce - still must be just as hard

assume: alg T decides totality

construct H, a decider for HALT

H(P,x):=

1) define **Q(y):=** ignore input. run P on x. output YES. (as long as it doesn't get caught in ∞ loop)

2) return **T(Q)** → whether P halts on x.

EQUIV:

input: P, Q ← 2 programs

out: YES if P & Q output YES on <u>same</u> inputs

NO otherwise

2 diff. Solns

WTS: program is decideable, recognizeable:

Pf: write alg for it

WTS: program is undecideable

Pf: reduce from HALT

THM: Equiv is undecideable

Pf: HALT $\leq_p$ EQUIV

assume alg. E decides EQUIV

H(P,x):

decides HALT

1) define Q(Y): "ignore input. run P(x), output YES"

2) define R(Y): "ignore input, output YES" ← always says yes

3) return E(Q,R)

Introduction to Algorithms: 6.1210 Recitation 24

Massachusetts Institute of Technology
Brynmor Chapman, Srini Devadas, Henry Corrigan-Gibbs, Will Leiserson

Recitation 24

Computability

Exercise: Acceptance Problem

The following problem, similar to the Halting Problem, is called the Acceptance Problem.

ACCEPT

Input: a binary string, interpreted as a pair $(\mathcal{A}, x)$

- $\mathcal{A}$ is an algorithm
- x is some other binary string

Output:

- YES if $\mathcal{A}(x)$ eventually halts and outputs YES
- NO otherwise, i.e. if $\mathcal{A}(x)$ outputs NO or doesn't halt

(a) Prove that ACCEPT is *recognizable*. That is, prove that there exists an algorithm $\mathcal{B}$ such that $\mathcal{B}(\mathcal{A}, x) =$ YES iff ACCEPT$(\mathcal{A}, x) =$ YES.

(b) Prove that ACCEPT is undecidable by reducing from HALT.

DP Review

Exercise 1: Coin Row Problem Revisited

Here's a variation on the coin row problem from the very first DP lecture: there is a row of coins with positive integer values, and once again you want to maximize the total value of the coins you pick up. This time, the rule is that you can only pick up a run of r consecutive coins if there are at least r coins anywhere to the left of it that you don't pick up.

For example, here is a row of coins, with the coins you pick up in the optimal solution circled:

$$10 \quad \enclose{circle}{12} \quad 8 \quad \enclose{circle}{27}\,\enclose{circle}{25} \quad 11 \quad \enclose{circle}{9}\,\enclose{circle}{1}$$

In the optimal solution, you skip 10 and 8 so that you're allowed to pick up both 27 and 25. But you only skipped two coins on the left, so you aren't allowed to also pick up 11.

Note that at $r = 1$, the rule says that to pick up a single coin you must have skipped a coin to its left; in particular the first coin has to be skipped.

Design an algorithm to find the maximum value you can pick up, and that runs in $O(n^3)$ time when there are n coins.

ACCEPTANCE:

a) WTS: can make alg. that if A(x) = YES → outputs YES, A(x) = no / ∞ loop → outputs NO/loops forever

* can make a recognizer that loops forever

P(A, x) := "run A on x. do what it does."

b) HALT $\le_p$ ACCEPT

assume alg A for ACCEPT.

H(P, x) :=

1) construct P'(y): "run P on y. output YES"

2) run A on (P', x) if HALTS → outputs YES, o.w. → ∞ loop

① S: x[i, j] = max value you can pick up @ coin i while skipping @ least j of them.

R: $x[i,j] = \max \{ T(k, \max(i-k-2, j-1, 0)) + \sum_{t=k+1}^{i-1} c_t$ suppose last skipped coin = k. after, we pick up i-k-1 coins ending w/ i-1. only legal if skip ≥ i-k-2

T: x(i, j) depends on T(k, j') for k < i

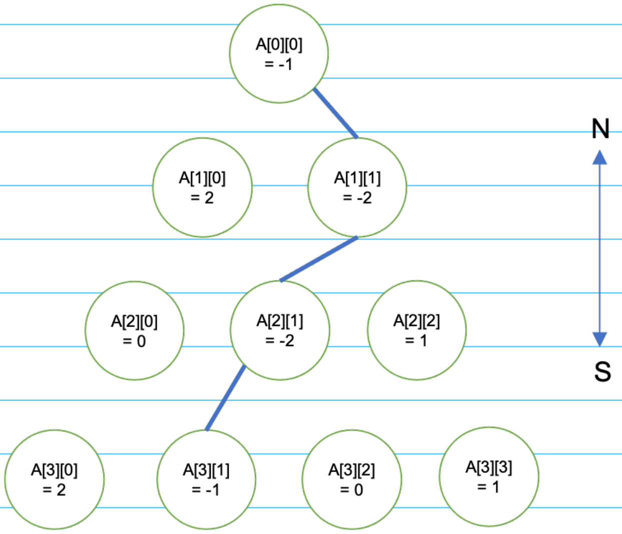
B: x(0, 0) = 0, T(0, j) = -∞

O: x[n, 0]

T: n^2 subproblems. O(n^2) & smartly eval options in decr. order of k & store running sum. subproblems computed in O(n) → total R.T. O(n^3)

Exercise 2: Srini's Journey Goes South

Srini is driving his JunkJet through Cityland. Cityland is a nation of $1+2+\cdots+n = \frac{n(n+1)}{2}$ cities, arranged in an equilateral triangular lattice. The j^{th} city on row i is $C[i][j]$. Cityland is oriented so that one of its axes of symmetry runs due north-south, as shown in the diagram below.



Srini starts at the northernmost city in Cityland. Each day, he drives from his current city to one of the two cities immediately south (i.e., immediately southeast or southwest) of it, meaning that if he's at city $C[i][j]$, the next day he can go to either $C[i+1][j]$ or $C[i+1][j+1]$. His journey ends when he reaches one of the n cities on Cityland's southern border.

Each city has a *disapproval score*, which is an integer. The disapproval scores **may be zero, negative, or positive**. The disapproval score for the j^{th} city on row i is $A[i][j]$. For example, if $n = 4$, the disapproval scores may be as shown above.

Over all possible journeys, Srini wants to know the **maximum product** of disapproval scores of the cities he visits. In our example above for the particular values of A, the maximum disapproval score corresponds to the path marked and has value $(-1)(-2)(-2)(-1) = 4$.

Design an $O(n^2)$-time Dynamic Programming algorithm that returns the cities on the path with maximum product of disapproval scores. You may assume that all arithmetic operations take constant time.

S: ~~x[i][j]~~ for $0 \le j \le i \le n$: Tmax(i,j) & Tmin(i,j) = max & min product of subpath

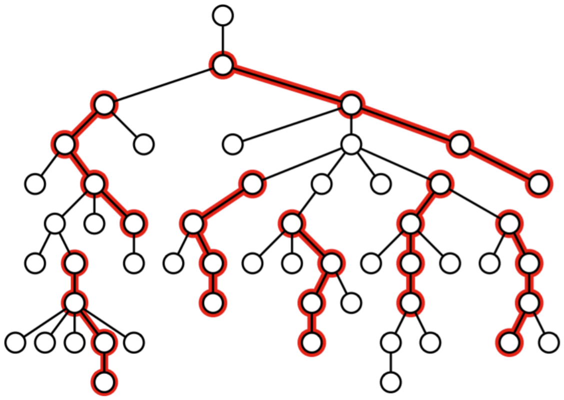
R: $$\text{Tmax}(i,j) = \begin{cases} A[i][j] \cdot \max(\text{Tmax}(i+1,j), \text{Tmax}(i+1,j+1)) & \text{if } A[i][j] > 0 \\ 0 & \\ A[i][j] \cdot \min(\text{Tmax}(i+1,j), \text{Tmax}(i+1,j+1)) & \text{if } A[i][j] < 0 \end{cases}$$

$$\text{Tmin}(i,j) = \begin{cases} A[i][j] \cdot \min(\text{Tmax}(i+1,j), \text{Tmax}(i+1,j+1)) & \text{if } A[i][j] > 0 \\ 0 & \\ A[i][j] \cdot \max(\text{Tmax}(i+1,j), \text{Tmax}(i+1,j+1)) & \text{if } A[i][j] < 0 \end{cases}$$

Exercise 3: Disjoint Paths

Alex has a rooted tree T with n vertices. Alex would like to find k-edge paths in the tree which are *directed towards the root*, meaning each vertex in the path (except the first) is the parent of the previous vertex. Note that each such path has $k + 1$ vertices. Alex further requires that the paths are all disjoint, meaning no two paths share any vertices.

Below is an example where $k = 3$ with seven disjoint paths. It's possible to find eight disjoint 3-edge paths in this tree.



(a) Describe, analyze, and prove correctness of an $O(n)$-time greedy algorithm to compute the maximum number of disjoint paths that can fit in the tree. Your algorithm is given a rooted tree T and an integer k as input, and it should output the largest possible number of disjoint k-edge paths directed towards the root in T. Do not assume that T is a binary tree. For example, given the tree above as input, your algorithm should return 8. Note that you only need to return the number of paths and not the location of the paths.

Prove the greedy choice property you rely on, and argue the correctness and runtime of your algorithm.

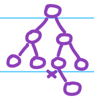
(b) Now suppose each vertex in T has an associated reward, and your goal is to maximize the total reward of the vertices in your paths, instead of the total number of paths. Give an example where your greedy algorithm does not return the optimal reward.

Hint: Try to come up with an example for small k.

(c) (optional) Describe an $O(kn)$-time algorithm to compute the maximum possible total reward from vertices in paths.

a) GC: consider node x w/ height k.

use a path starting from leaf & ending @ x.



GCP: there ∃ @ least 1 optimal soln containing GC.

Pf of GCP: consider optimal soln S that doesn't make GC.

if S doesn't use x at all, we can add new path (just as optimal; might free up smthg)

if S does use x, can shift path down

Alg:

·recursively compute height of each vertex

·when node x has height k,

1. add 1 to counter
2. remove x & its subtree from tree

O(n) post order traversal.

Pf by Induction:

BC: height < k

Ind Step: consider opt. soln

by GCP, can assume S makes same 1st step as G.

say we removed node x to get T.

We can consider our alg. to have run on T' up to this point

by IH, $|G'| \geq |S'| \rightarrow |G| = |S|$

greedy does as good as optimal