Date: / /

Lecture Notes: Syllabus, I/O 1.13.25

Standard IO:

$ program 2> error #redirect err
$ program > output 2>&1 #redirect both

Programming Challenges I/O:

1. output is line oriented
2. input may have sentinels
3. input may specify # items to read
4. Normally do not have to handle bad input

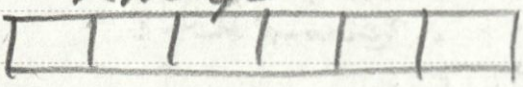
* must correctly format output exactly for grading

see exercise 01-A

Date. / /

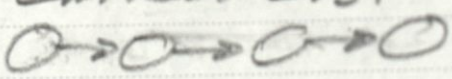
Lecture Notes: Arrays, Lists 1.15.25

Arrays



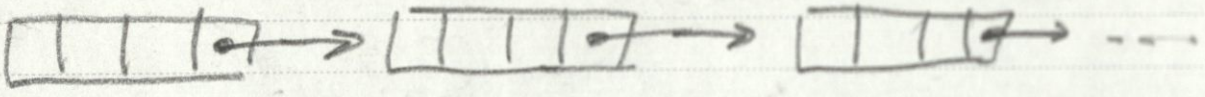
- contiguous container of elements
- contiguous - next to each other in ram
- random access: O(1)
- poor insertions and deletions
- static amount of space

Linked List



- sequence of nodes that point to each other
- efficient insertions
- bad random access O(n) lookup
- not contiguous
- more memory efficient?
- overhead for pointers

Deques:



random access

locality

combines benefits of both arrays and linked lists

Date. / /

Whiteboarding: Tips

1. Ask questions to resolve ambiguity
what are the data types? assumptions?

2. Design an algorithm
what is the complexity? what trade-offs?

3. Write pseudocode first
with the goal of converting to real code later

Lecture Notes: Complexity 1.17.25

Big-O: describes the efficiency of an algorithm in terms of the amount of data it must process

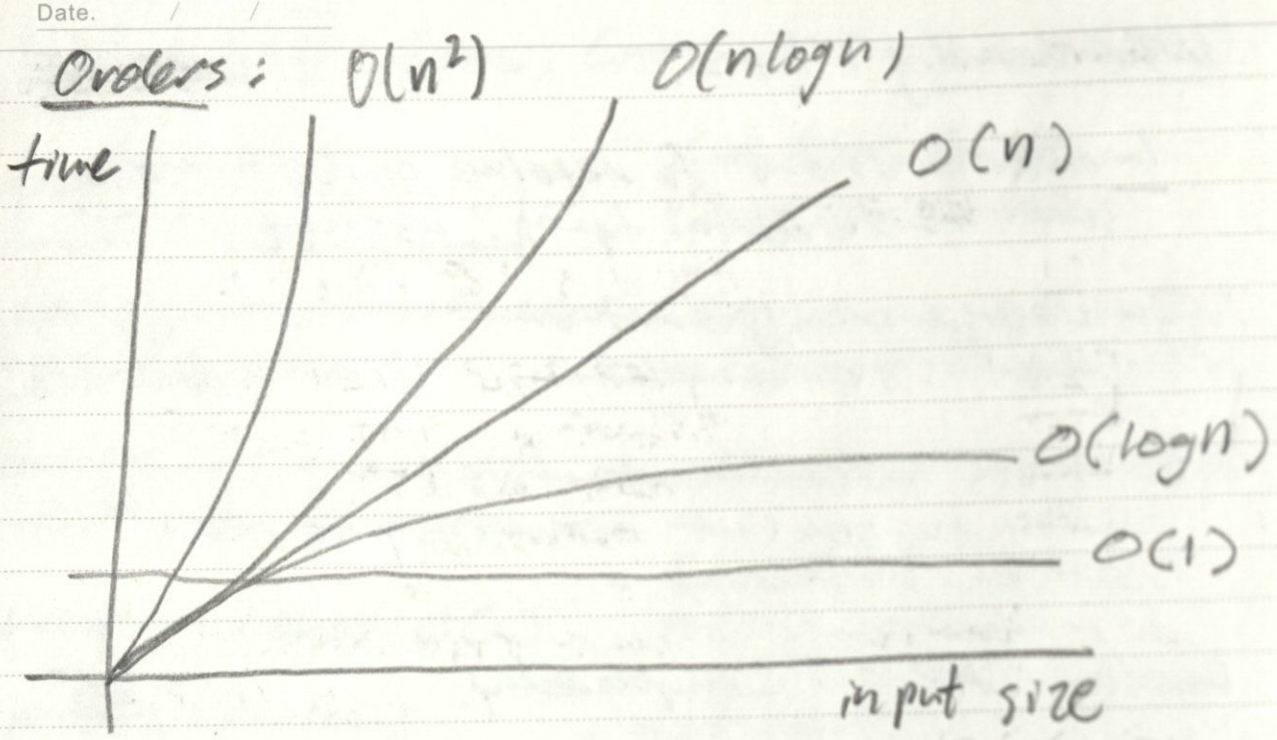
time complexity: # of operations an algorithm performs relative to input size

space complexity: amount of additional memory an algorithm requires relative to amount of input

Def. a computation is $O(f(n))$ time if for large n, the computation time is at most $kf(n)$ for some k

Date. / /

Orders:



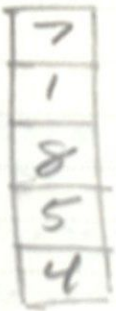
Coding Styles:

Linters: tools to help with coding style

Date. / /

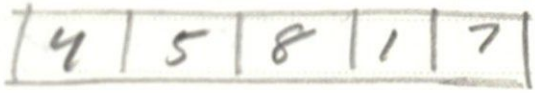
Lecture Notes: 1.22.25

Stacks: 45817



LIFO
"last in first out"
used in DFS
backtracking
function calls

Queues:



FIFO
"first in first out"
used in BFS
Scheduling
threading
unix pipeline

Implementation:

stack = [ ]
- pop()
- append()

queue = [ ]
- popleft()
- append()

Deques: 45817

import collections
q = collections.deque()

- amortized constant time insertion and deletion from front and back
- constant time random access
- some cache locality

Date. / /

Lecture Notes: 1.24.25

Testing: Overview

- determines whether code is correct
- like proofs for math

functional	performance	unit
• checking correctness	• meets time or resource constraints	• check individual components

penetration, integration, load

Lecture Notes: 1.27.25

linear search: go through list one-by-one in O(N) time

binary search: split search space in half each time in O(logN) time
- need a way of determining which half of search space our target is

algorithm:
1. compute midpoint
2. check if midpoint is target
3. target < midpoint: go left
4. target > midpoint: go right
5. repeat until found or boundary

Date. / /

Python Standard Library:

Linear Search:

```
if target in contains:
    print("Found it")
```

Binary Search:

```
import bisect
index = bisect.bisect_left(data, target)
if index < len(data) and
    data[index] == target:
    print("Found it")
```

Lecture Notes: 1.29.25

Benefits of Sorting:

1. Min/Max values
2. Median
3. Frequency / duplicates

O(n log(n)) sorting:

```
V = sorted(V)  // returns a copy
V.sort         // in-place
```

↳ uses Timsort & Mergesort

```
C++: sort(V.begin(), V.end())
```

↳ introsort & quicksort

Date. / /

Reversing Directions:

C++

sort(v.rBegin(), v.rEnd())

Python

sorted(v, reverse=True)
reversed(sorted(v))
v.sort(reverse=True)

Stable Sorting: maintains relative order of equivalent elements

$4_1, 6_2, 1, 5 \rightarrow 1, 4, 5, 6_1, 6_2$

timsort, insertion sort, and merge sort are all stable

Lecture Notes: 2.3.25

set: collection of unique items
map: collection of key: value pairs

can be used to check membership, counting, memoization

built with hashing

Date. / /

Set:
- insert(value)
- remove(value)
- search(value)
- membership
- Unix Permissions

Map:
- insert(key, value)
- remove(key)
- search(key)
- Table of k:v pairs
- File System
- Javascript

Sets and Maps: Ordered vs Unordered

- Sets are unordered: means NO random access
- Maps are ordered: maintain order you insert item

Ordered:
- each element needs a comparison operator defined
- implemented with a BST

Unordered:
- each element needs to be immutable
- typically implemented with a hash table

Sets and Maps: Python

```
my_set = set()          my_dict = dict()
my_set = {}             my_dict = {}

my_set.add(1)           my_dict[key] = value
```

Date. / /

Implementation:

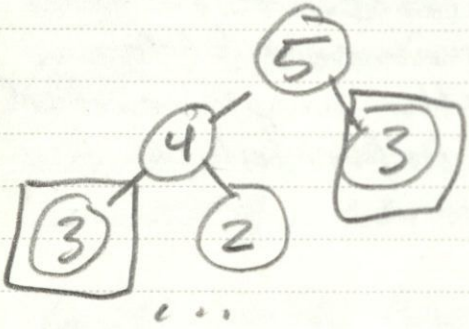
Binary Search Tree:
Time: $O(\log n)$
Space: $O(k \cdot n)$

Hash Table:
Time: $O(1)$
Space: $O(n)$
$O(k(m+n))$

Techniques:
Counting: keeping track of frequencies
Memorization: Caching subproblem results
- fibonacci

History: keeping track of info over time

Ex. Fibonacci sequence
for problems with overlapping subproblems, can store result of previous subproblems for future re-use



Date. / /

Lecture Notes:

2.5.25

Sets and Maps: Techniques

1. Counting: Using a set or map to count frequencies
2. Memoization: using a map to cache subproblems
3. History: using a set or map to track seen information

see ex is_palindrome.py
- counting

see ex memo_fib.py
- memoization

Date. / /

Lecture Notes: 2.10.25

Complete Search: sometimes the best approach is to try every possibility

ie brute force

- guarantees, it will find the right answer
- very inefficient
- first pass solution

Combinations: $O(2^n)$ Permutations: $O(n!)$

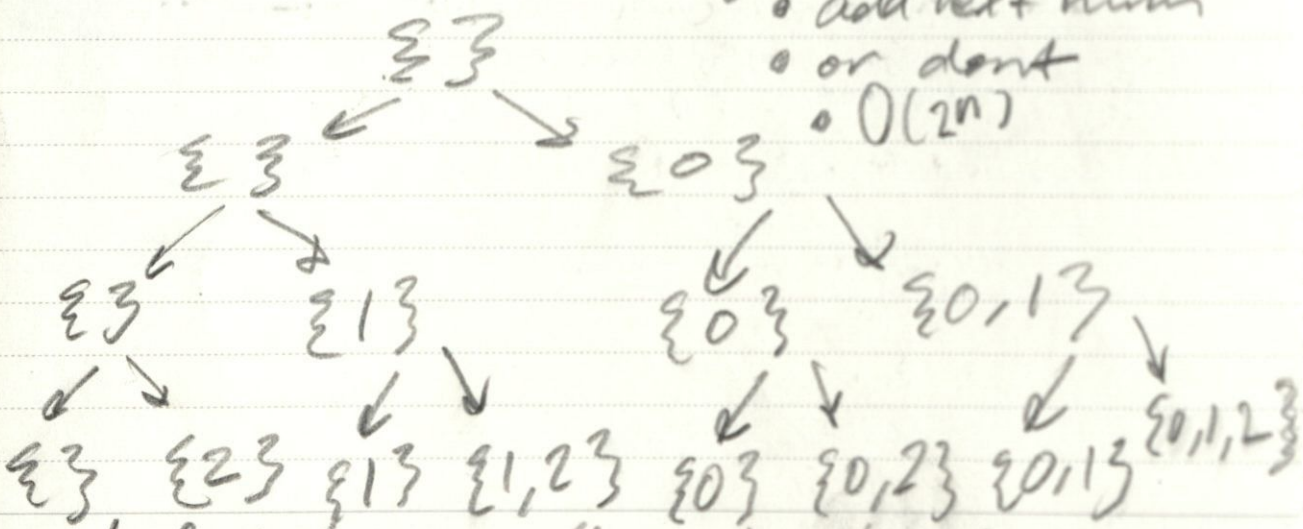
Subsets:

- itertools.combinations
- bitmask
- recursive solution

Recursive: Combinations

$S = \{0, 1, 2\}$

- add next num
- or don't
- $O(2^n)$



leaf nodes are all combinations

Date. / /

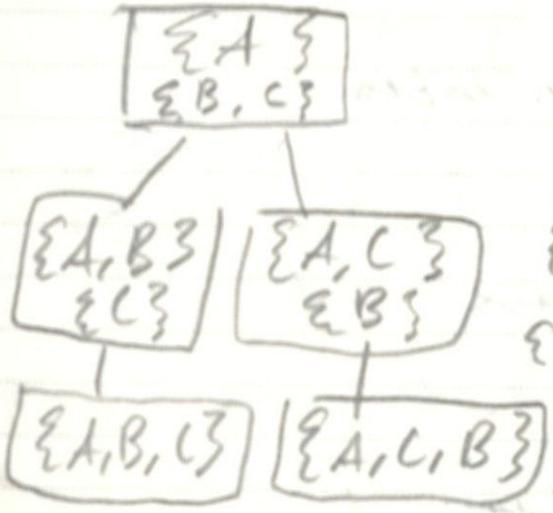
Lecture Notes: 2.12.25

Complete Search:

Permutations: order matters
Combinations: order does not matter

itertools.permutations (same as combinations)

Permutations Diagram: $S = \{A, B, C\}$



{B}
{A, C}
{B, A}
{B, C}
{B, A, C}
{B, C, A}

{C}
{A, B}
...

note. permutations are all the same length as the initial set.

ie. no empty set etc.

Date. / /

Backtracking:

- start w/ empty sol
- add 1 item and see if sol violates constraints
- if it does not, recursively add another item
- otherwise back-up to prev stage and try alternatives

Lecture Notes: 2.17.25

Review: Set ADT

Set Implementation:

BST:
Time: O(log N)
Space: O(N)

- ordered set

Hash Table:
Time: O(1)
Space: O(N)

- unordered set

Bitsets: instead of storing elements, we can store whether an element is in a set by marking a bit

	7	6	5	4	3	2	1	0	index
574 →	1	0	1	1	0	0	0	0	bit

note. not counting, just checking membership

- use ints as sets
- use logical bit operators to set and clear individual bits

Date. / /

Bitsets: Subsets

To generate subsets using bitsets, start with $\emptyset$, and then increment by 1

Each bit represents membership of element in subset

Bitset	A	B	C	Subset
0	0	0	0	$\{\}$
1	0	0	1	$\{C\}$
2	0	1	0	$\{B\}$
3	0	1	1	$\{B, C\}$
4	1	0	0	$\{A\}$
5	1	0	1	$\{A, C\}$
6	1	1	0	$\{A, B\}$
7	1	1	1	$\{A, B, C\}$

Date. / /

Lecture Notes: Greedy Algs 2.19.25

A greedy algorithm constructs a solution by always taking the best local choice (choice at the moment)

directly constructs final solution and never takes back a choice

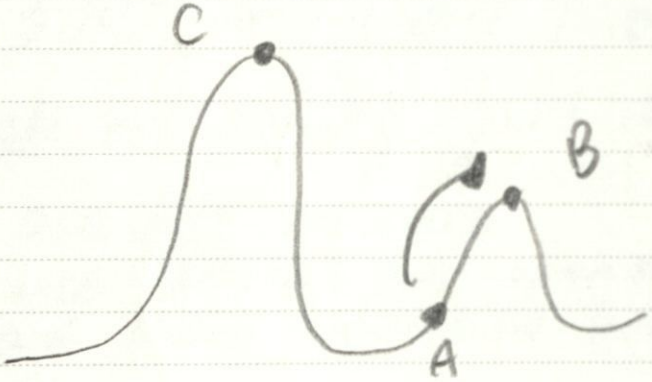
Usually very efficient

Locally optimized solutions are globally optimal

Ex. Making Change

17¢ = 10+5+1+1
36¢ = 25+10+1
64¢ = 25+25+10+1+1+1+1

Limitations: not guaranteed to find the optimal solution



Date. / /

Lecture Notes: 2.24.25

Dynamic Programming combines the correctness of complete search and the efficiency of greedy algorithms

- cache/memoize results of subproblems in memory to prevent repeated computations
- Efficiently solves a set of search and optimization problems

Caching: Memorizing Computations

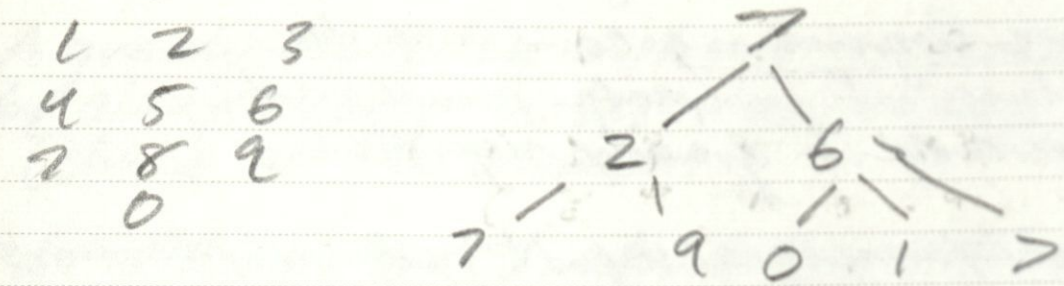
- identify overlapping subproblems
- store results in lookup table

Steps:

1. identify recursive structure
2. memoize recursive step in cache
3. return memoized solutions

Date. / /

Caching: Knight's Dialer



* can either work forwards or backwards with these

Coin Problem: (1, 3, 4)

coins	0	1		1	1			
amt	0	1	2	3	4	5	6	7

min(t[i-1], t[i-3], t[i-4])

backward

Stair Problem:

#stairs	0	1	2	3	4
#ways	0	1	2		

table[n] = table[n-1] + table[n-2]

Date. / /

Lecture Notes: 2.26.25

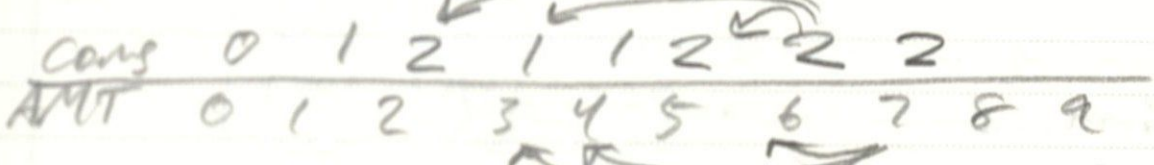
Making Change: (1, 3, 4)

Greedy Alg: 6 → (4, 1, 1)
Correct Sol: 6 → (3, 3)

i.e. greedy failed

Use DP:

Bottom Up (Backward):

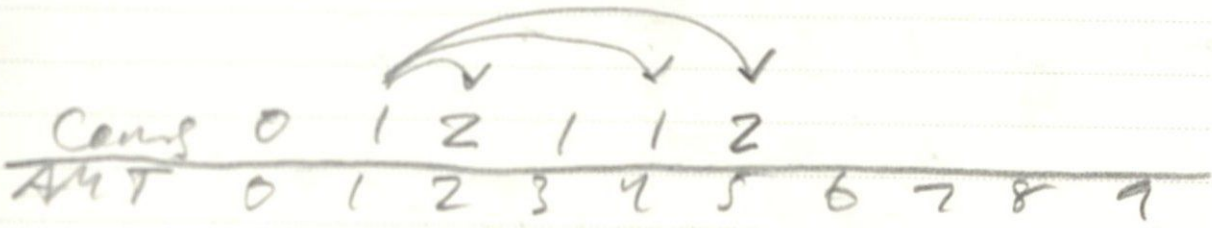


use $\min(i-1, i-3, i-4) + 1$

$= \min(\min(5, 3, 2), 1, 1) + 1$

; note. initialize coins to ∞!

Bottom Up (Forward):



Table[i + coin] = min(Table[i + coin], Table[i] + 1)

note initialize coins to ∞!

Date. / /

Squirrel Hunting: PAD w/ 0's for bounds

					0	0	0	0	0	0
1	5	2	3	6	0	1	6	8	11	17
4	3	2	1	2	0	5	9	11	12	19
3	8	4	2	1	0	8	17	21	23	24
0	5	2	3	4	0	8	22	24	27	31
3	1	4	2	1	0	11	23	28	30	32

Max(left, up) + current_cell

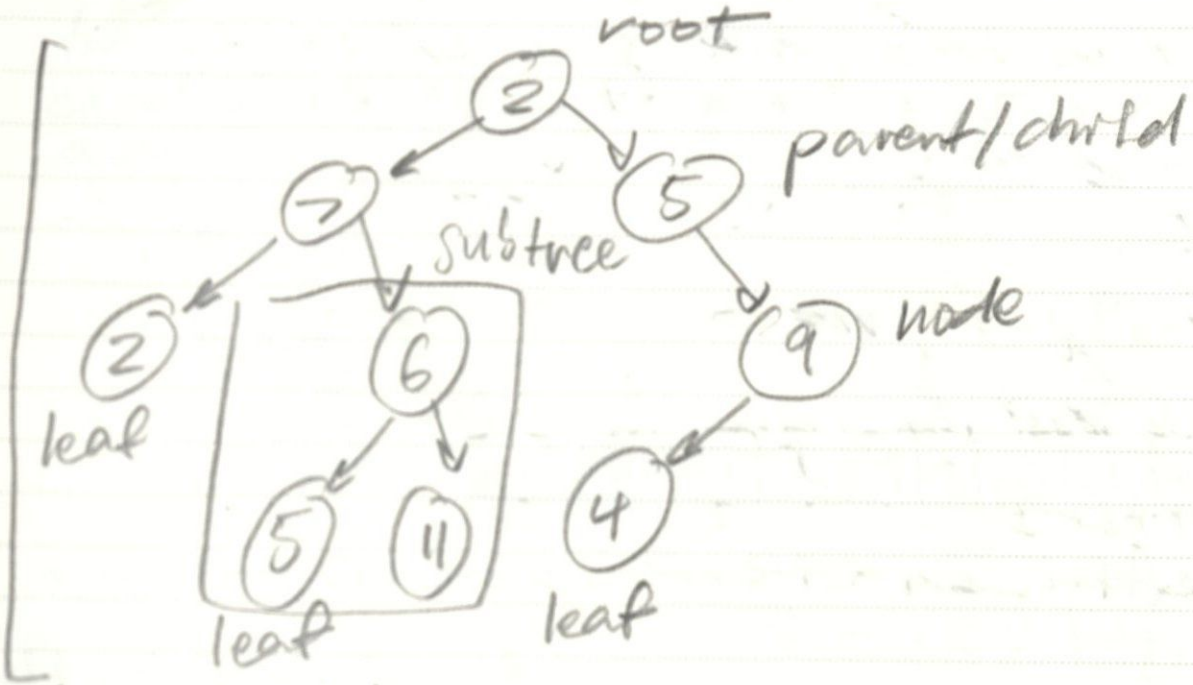
How did you get to 32?

follow squeal that correlates w/ accum_cell - curr_cell

Date. / /

Lecture Notes: 3.17.25

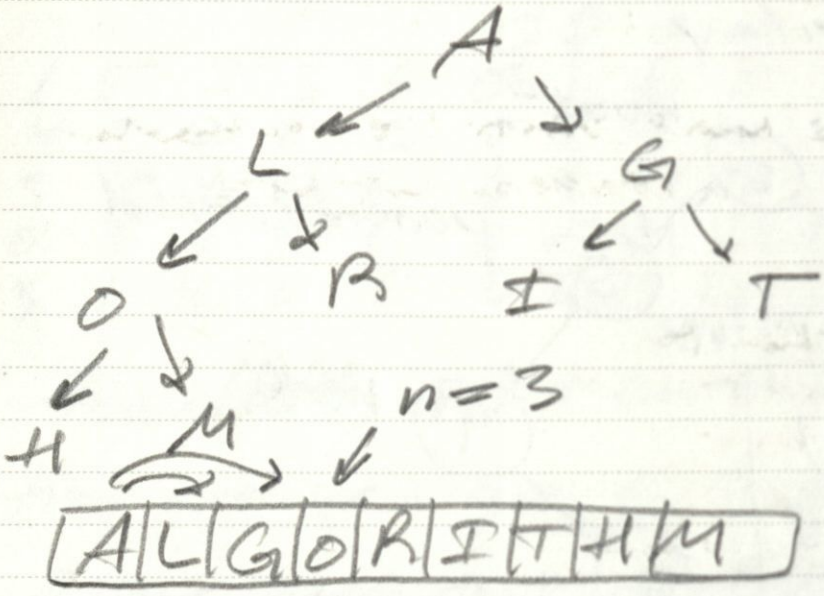
Trees: Vocabulary



full: every node has 0 or 2 children
complete: every level has max children except last (left → right)

Date. / /

Representations:



BFS Rep

A	L	G	O	R	I	T	H	M

left: $2n+1$

Right: $2n+2$

Parent: $\lfloor (i-1)/2 \rfloor$

Binary Heap: binary tree where a parent is larger than all of its children (or smaller)

→ used in priority queues

must also be complete

BST: for every parent node
1) left subtree must be less than parent
2) right subtree must be greater

* must be balanced *

Date: / /

Lecture Notes 3.19.25

Tree Traversals:

traversing a tree means visiting each node exactly once in a deterministic order

- Breadth-first
- Depth-first
  - pre-order
  - in-order
  - post-order

* implementation in slides

Divide and Conquer:

top-down technique that consists of dividing the problem into smaller subproblems and combining the partial solutions

* exs of this in slides

Date. / /

Lecture Notes: 3/24/25

Graphs:

collection of vertices and edges

Representation:

- Adjacency List (more used)
- Adjacency Matrix

Traversals:

- BFS
- DFS
- Dijkstras — computes shortest path

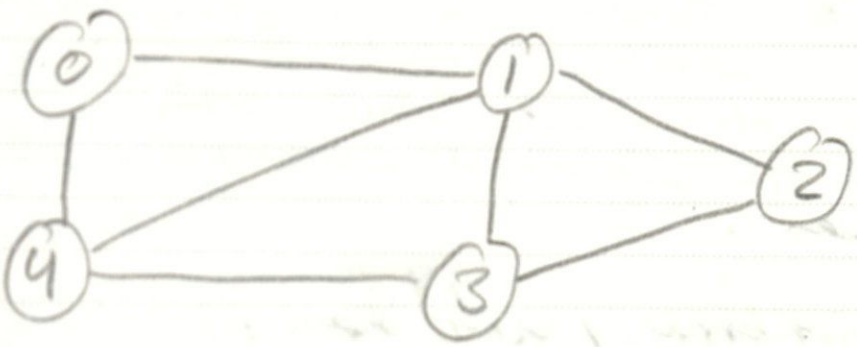
Vocabulary:

edge: connection between vertices
directed: has a direction
weighted: edge has an associated cost
vertices: circles with values
cyclic vs acyclic
connected: can reach any node from any node (vs disconnected)

simple: only one edge b/w 2 nodes
no self loops

Date. / /

Representation:



Adjacency List:

```
graph = {
        0 : [1, 4]
        1 : [0, 2, 3, 4]
        2 : [1, 3]
        3 : [1, 2, 4]
        4 : [0, 1, 3]
    }
```

Adjacency Matrix:

```
graph = [ [0 1 0 0 1],
          [1 0 1 1 1],
          [0 1 0 1 0],
          [0 1 1 0 1],
          [1 1 0 1 0] ]
```

Date. / /

Lecture Notes: 3.28.25

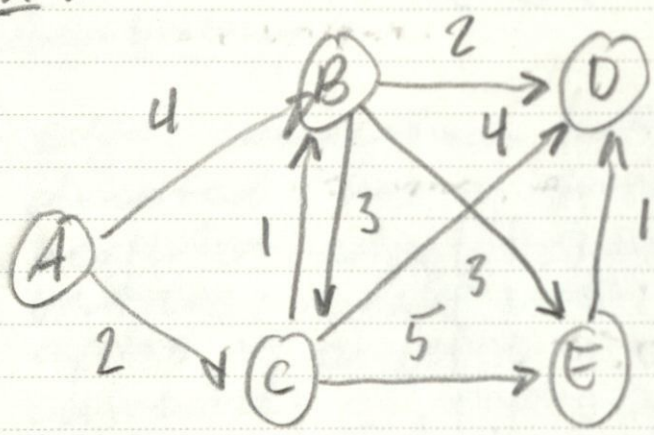
Dijkstra's: single source shortest path algorithm

determines the shortest path between two vertices

using our traversal alg., we repeatedly remove a vertex from the frontier until we find the shortest path

to choose next vertex, select the cheapest via minheap
- based on total distance

Ex.



Distances = {
A: 0
B: ∞ 4 3
C: ∞ 2
D: ∞ 6 5
E: ∞ 7 6
}

Frontier = {B, 4, D, E, }
Visited = {A, C, B, D, E}

Date. / /

Pseudo-Code:

```
Dijkstra(g, v):
  frontier = PQ
  visited = { }

  while frontier:
    v = frontier.popl()

    if v in visited:
      continue

    for u in g[v]:
      frontier.append(u)
```

for frontier we use a priority Q to store vertex (cost, name_prev) and order by cost

for visited, we use a map to record either
  a) node → prev
  b) node → t_cost

we can terminate loop when we reach target vertex

Date. / /

Lecture Notes: 3.31.25

Spanning Tree: acyclic graph that has a path btw any two vertices

Minimum Spanning Tree: spanning tree with the least cost

Prim's Algorithm: local expansion
Kruskal's Algorithm: global expansion

MST:
A spanning tree of a graph $G=(V, E)$ is a subset of edges forming a tree connecting all vertices

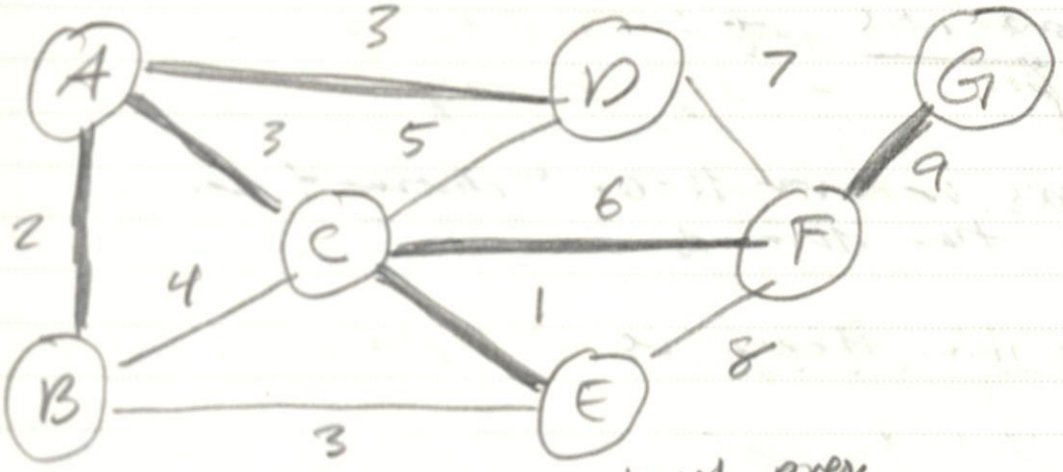
- Connected
- Undirected

MST is a spanning tree with the least total cost

, sum of edge weights is a minimum

Date: / /

Ex.



(0, A, A)

cost ↑ Target ↑ Prev

~~(2, B, A)~~

~~(3, C, A)~~

(3, D, A)

(6, C, B)

(5, E, B)

(8, D, C)

(4, F, C)

target	prev
A	A
B	A
C	A
D	A
E	C
F	C
G1	F

dijkstra numbers

want isolated cost not total cost

Prim's MST:

1) Use dijkstra's ad pick any node as start node
2) Only add edge's cost to PQ don't accumulate

Date. / /

Lecture Notes 4.2.25

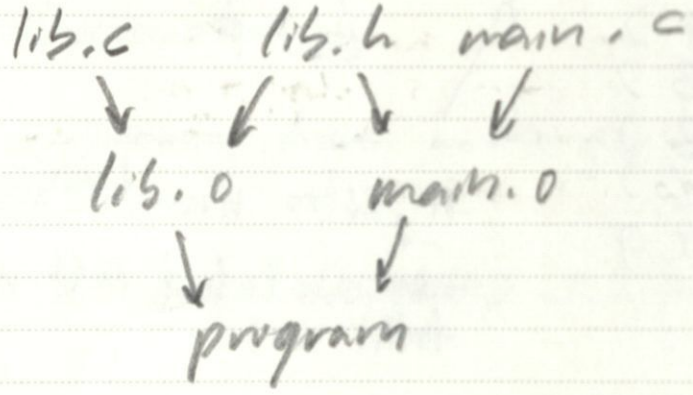
Topological Sort:

models a large collection of actions, objects, or events that depend on each other through a DAG

Motivation: Makefiles

target: sources

```
lib.o: lib.c lib.h
main.o: main.c lib.h
program: lib.o main.o
```



frontier: ~~queue~~, any data structure
visited: set
degrees: dictionary

Date. / /

Ex. frontier :
lib.c, lib.h, main.c

visited : {
lib.c, lib.h
}

degrees: {
lib.c : 0
lib.h : 0
main.c : 0
lib.o : ~~2~~ + 0
main.o : ~~2~~ 1
program : 2
}

degree = # incoming edges

1. init frontier w/ all nodes of degree 0
2. pop frontier add to visited
3. update neighbor's degrees
4. add node from degrees to frontier
if degree is now 0

Date. / /

Lecture Notes: 4.7.25

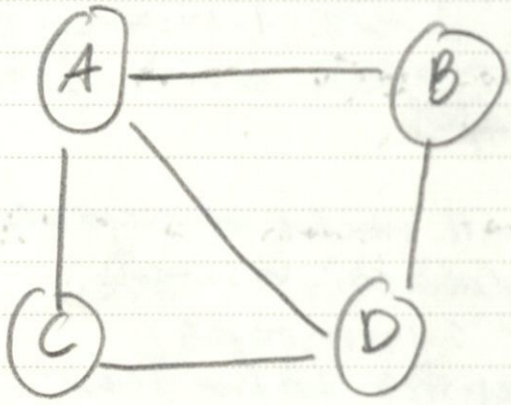
Eulerian Path: goes through every edge at least once

german Sewage problem

Hamiltonian Path: goes through every vertex at least once

traveling salesman problem

Ex.



Eulerian: $A \to B \to D \to C \to A \to D$

Hamiltonian: $A \to C \to D \to B$

& there can be multiple paths

Circuit. Path that starts and ends on the same node

Date. / /

Rules.

Eulerian:

1) All nodes have an even degree
2) All nodes have an even degree except for exactly two

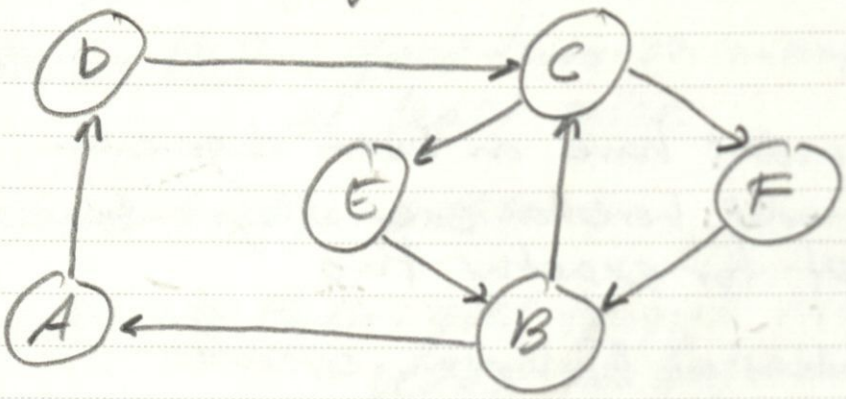
(1) guarantees an Eulerian circuit

Hamiltonian: NP-hard

1) connected
2) for each pair of vertices either degree(u) + degree(v) $\geq n$ u and v are adjacent
3) each vertex has degree $\geq n/2$

Date. / /

Hierholzer's Algorithm:



1) build a circuit
2) mark all edges used as used
3) look for extra edges connected to circuit
4) build a subcircuit with extra edges
5) add subcircuit to circuit
6) repeat until all edges visited

Date. / /

Lecture Notes: 4.9.25

Flows.

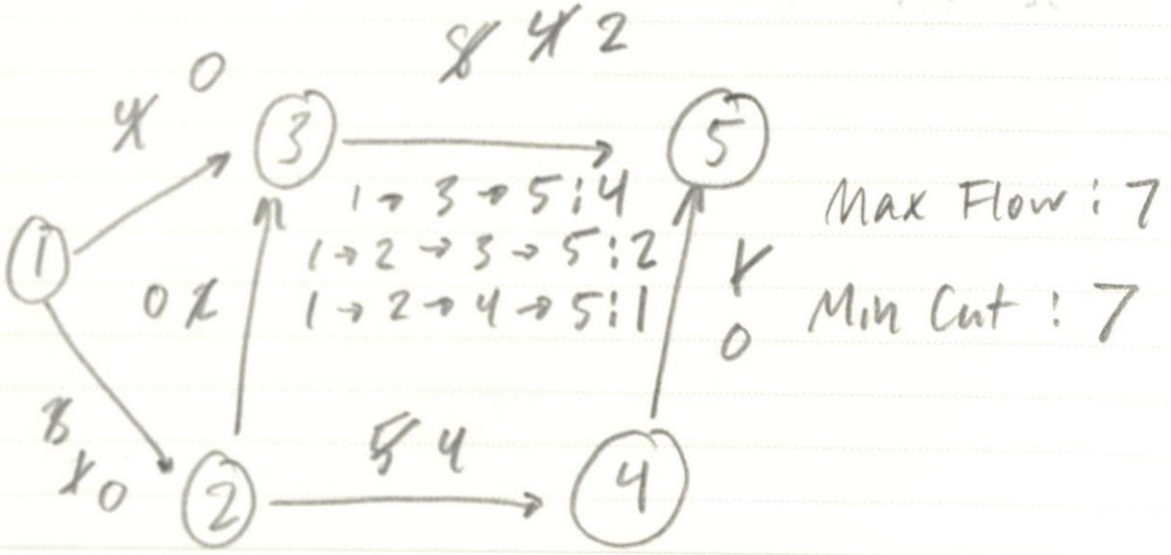
Source: vertex with only outgoing edges

Sink: vertex with only incoming edges

Max flow: maximum amount of "stuff" that can flow from source node to sink node using weighted edges

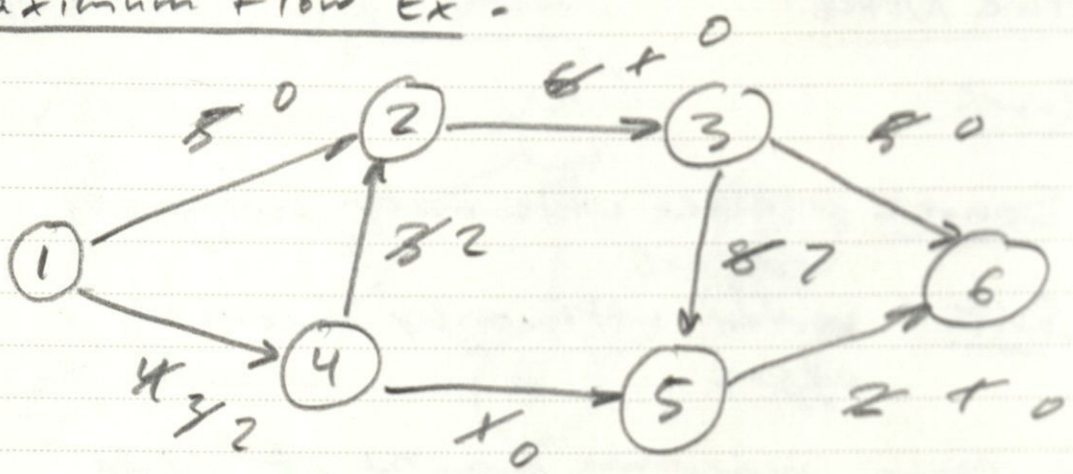
Flow Overview:

1) BFS Traversal → Paths
2) Find smallest edge on path
3) Subtract weight from edges in path
4) Don't take 0 edges



Date. / /

Maximum Flow Ex:



Max Flow = 7

1) BFS: 1 2 4 3 5 6

paths: 1 → 2 → 3 → 6 : 5
1 → 4 → 5 → 6 : 1
1 → 4 → 2 → 3 → 5 → 6 : 1

5 + 1 + 1 = 7 = Max Flow

Ford-Fulkerson: algorithm we just ran using DFS

Date. / /

Lecture Notes: 4.14.25

Prime #: only divisible by 1 and itself

Check:
- Brute force
- Sieve of Eratosthenes

GCD: largest positive integer that divides both of the integers

computed w/ Euclidean Alg