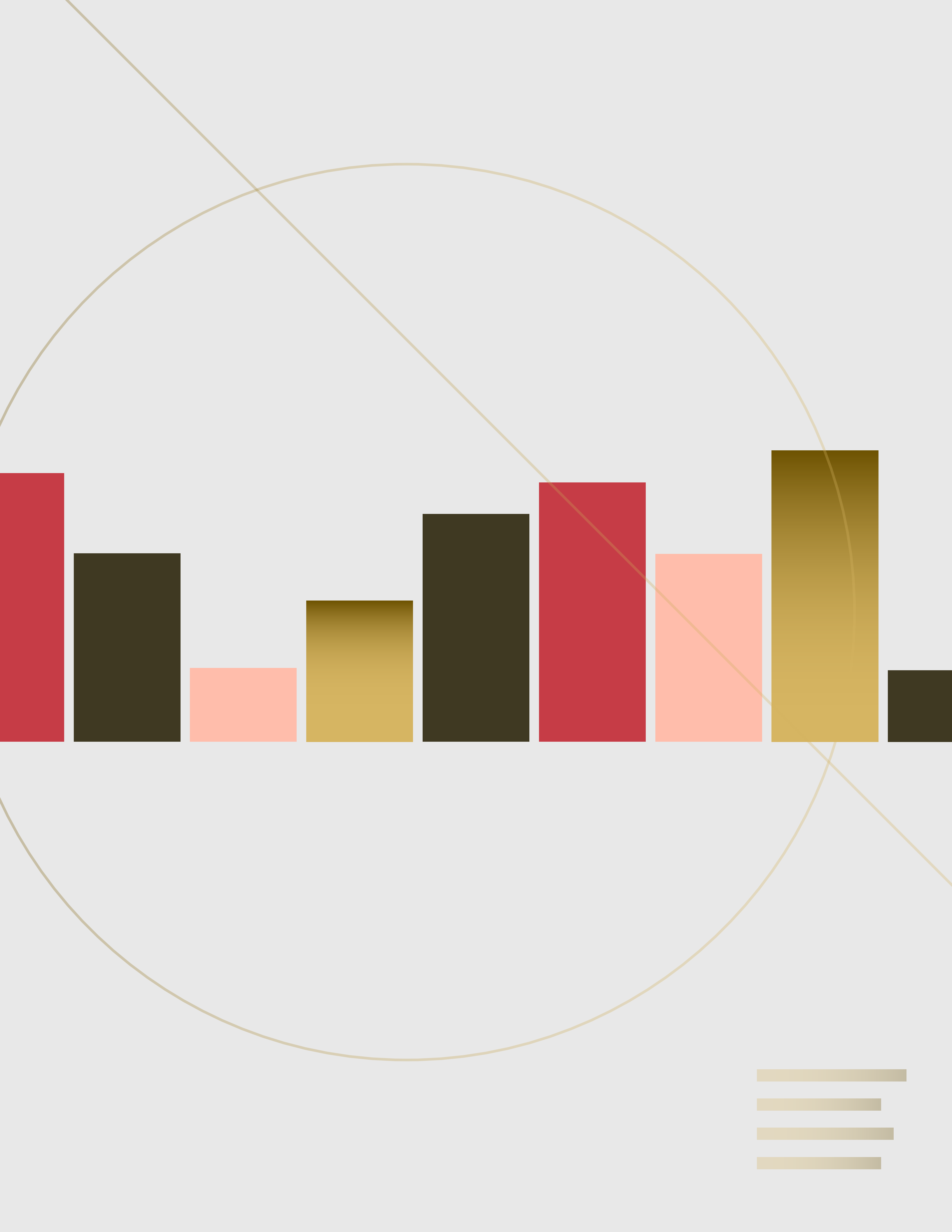
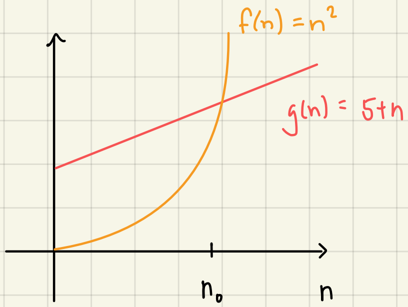


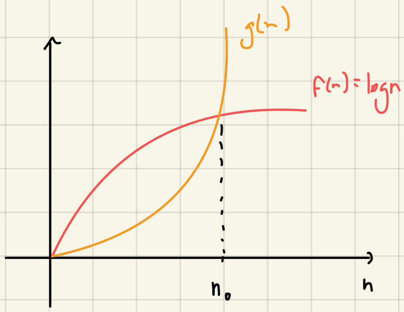
September 9th 2021 (Lecture 1)



$f \in \Omega(g)$

$f(n) = \Omega(g(n))$

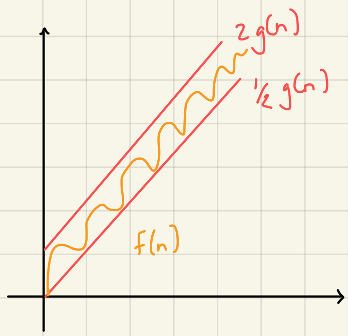
For all n big enough and for some c small enough f(n) is at least a constant c times g(n)
$f(n) \geq c\,g(n)$



$f \in O(g)$

$f(n) \in O(g(n))$

For all n big enough and for some c big enough f(n) is at most a constant c times g(n). $f(n) \leq c\,g(n)$



$f \in \Theta(g)$

$f(n) \in \Theta(g(n))$

For all n big enough, f and g grow at the same rate, i.e. $c_1, c_2 > 0$ ...
$c_1 g(n) \leq f(n) \leq c_2 g(n)$

Properties of Big-O

Suppose $f(n) = O(a(n))$ and $g(n) = O(b(n))$
↳ $c > 0$

- ☐ Sum: $f(n) + g(n) = O(a(n) + b(n))$
- ☐ Product: $f(n) \cdot g(n) = O(a(n) \cdot b(n))$
- ☐ Constant Multiplication: $c \cdot f(n) = O(a(n))$
- ☐ Transitivity: $f(n) = O(g(n))$ and $g(n) = O(h(n))$
  $f(n) = O(h(n))$
- ☐ Max degree: $f(n) = a_0 + a_1 n + \dots + a_d n^d \rightarrow f(n) = O(n^d)$
- ☐ Polynomial is subexponential: $d > 0 \longrightarrow n^d = O(a^n)$, $a > 1$
- ☐ Polylogarithmic is subpolynomial: $d > 0 \rightarrow (\log n)^d = O(n^r)$, $r > 0$

- ☐ little-o: "the growth of f is nothing compared to the growth of g"

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

- ☐ little-omega: "the growth of f is strictly dominates the growth of g."

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

- ☐ $A_1 A_2 \dots A_n$ an algorithm? :
  ↳ Standard

$$[A_1 A_2]_{ij} = \overbrace{\text{dot product}}^{m} \Rightarrow O(m^3)$$

↑ mxm matrices m^2

Consider: 3x500, 500x2, and 2x2000

$(A_1 A_2) A_3 = 3 \cdot 500 \cdot 2 + 3 \cdot 2 \cdot 2000 = 15\,000$

$(A_1)(A_2 A_3) = 500 \cdot 2 \cdot 2000 + 3 \cdot 500 \cdot 2000 = 10^6$

Complexity:

- ☐ Time: How fast does the algorithm run?
- ☐ Space: How much (extra) space does the algorithm require?

Note: Time complexity typically is lower bounded by space complexity.

Types of Analysis

1) Empirical Method: complexity measured by number of cycles, using instrumentation and profiling.
2) The Theoretical Method: complexity measured by number of primitive operations, using math and theoretical computer science.
   ↳ Derive upper and lower bounds on complexity

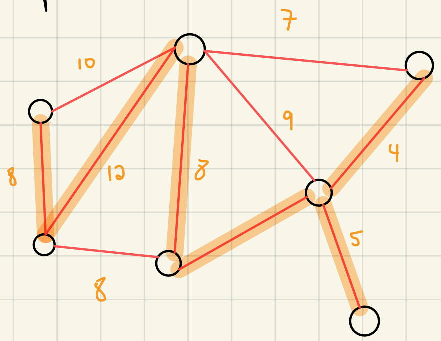
Time Complexity Analysis

- Complexity as a function of input size
- Measured in terms of number of primitive operations
- Worst case, best case, average case
- Abstracting to asymptotic behaviour / order of growth
- For recursive analysis — use the master theorem

Note: When using primitive operations model of computation, we will implicitly assume that a word contains $O(\log n)$ bits, for input size n.

September 13th 2021 (Lecture 2)

Example:

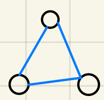


$G = (V, \mathcal{E})$

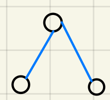
$e = (u,v) \in E$, $u \in V$; $v \in V$

example 1

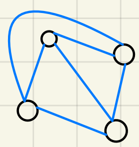
- Complete graph $(|V| = 3)$
  - K_3



- Incomplete graph



  - K_4

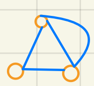


$\binom{4}{2} = \frac{4!}{2!(4-2)!} = \frac{4 \cdot 3}{2} = \boxed{6}$

Simple Graph

A simple graph is a graph w/ no multi-edges and no self-loops

ex.



ex.

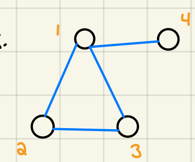
ex.

Representing Graphs $n = |V|$

(1) Adjacency List Representation:

An array Adj of $|V|$ lists for $u \in V$, Adj$[u]$ contains list of all vertices v st $(u,v) \in E$

ex.



Adj

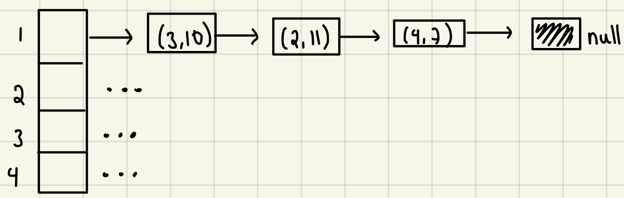
1 → 3 → 2 → 4 → null
2 → 1 → 3 → null
3 → 2 → 1 → null
4 → 1 → null

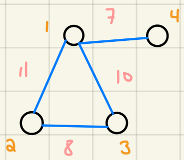
Space: $O(|V| + |\mathcal{E}|)$

A weighted graph is a graph st for each edge (u,v), there is a weight $\omega(u,v)$.

weight function
$\omega : V \times V \longrightarrow \mathbb{R}$

Weighted Version of Adj.





(2) Second Representation :

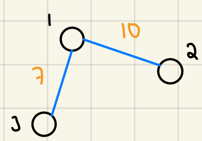
Adjacency Matrix A of size $|V| \times |V|$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{o.w} \end{cases}$$

or

Weighted graph $a_{ij} = \begin{cases} \omega(i,j) & \text{if } (i,j) \in E \\ \infty & \text{if } (i,j) \notin E \end{cases}$

ex.



$$\begin{bmatrix} \infty & 10 & 7 \\ 10 & 0 & \infty \\ 7 & \infty & \infty \end{bmatrix}$$

Definition – Spanning Tree

$T \subseteq E$ is a spanning tree of $G=(V,E)$ if (V,T) is a cycle and and is connected.

A graph $(G=(V,E))$ is connected if $\forall\, u,v \in V$ there is a path from u to v using edges in E.

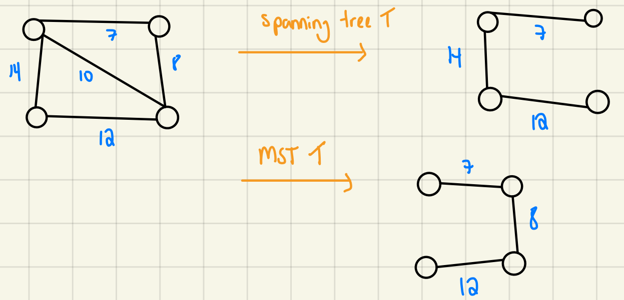
Definition – Cycle Property

For any cycle C in the graph, if th weight of an edge e of C is larger than th individual weights of all other edges of C, then this edge cannot belong to an MST.

* Something doesn't belong in th MST.

Definition – MST

Let G be weighted graph. We say T Minimum Weight Spanning Tree if T is spanning tree and its weight $w(T) := \sum_{i} w(u,v)$ is the minimum among all spanning trees.

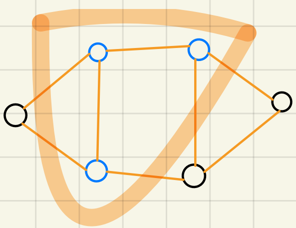


$W(T') = 14 + 7 + 12$
$= 33$

Definition – Cut

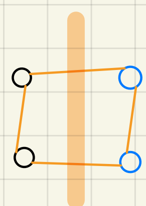
subtree

A cut $(S, V \setminus S)$ of undirected graph $G = (V, E)$ is a partition of V into two non-complet sets.



Legend

○ $\in S$
○ $\in V \setminus S$



Definition – Crosing Edge

An edge $(u,v) \in E$ crosses the cut $(S, V \setminus S)$ if one vertex is in S and the other one is in $V \setminus S$.

Algorithm Idea

Start with $A = \emptyset = \{\}$.

Incrementally, add an edge that belongs to MST T.

Assumption All edge weights are distinct.

Nishant say fact. Assumption $\Rightarrow$

Unique MST, "the" MST

Cut Property Theorem

Let $(S, V \setminus S)$ be a cut and let $e = (u,v)$ be min cost edge that crosses the cut.
Then edge e belongs to the MST.

* Something does belong in MST.

September 16th 2021 (Lecture 3)

Cut Property Theorem

Let $(S, V \setminus S)$ be a cut and let $e=(u,v) \in \mathcal{E}$ be a minimum weight crossing edge for the cut. The the MST contains e.

(i.e. e is a "safe edge")

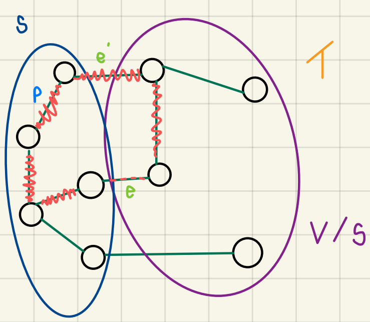
Proof (Exchange Argument)

Suppose the MST T doesn't contain e $(e \notin T)$.
So, since T is spanning tree, there is path P from u to v.

Let $T' = T \cup \{e\}$. So, there is cycle in T'.
Let $T'' = T' \setminus \{e'\}$. Since broke cycle, T'' is spanning tree. $\longrightarrow$ $w(T'')$

$= w(T) - w(e') + w(e)$

$< w(T)$ $\lightning$



Greedy MST Algorithm

$A = \emptyset$ for $j = 1 \rightarrow |V| - 1$

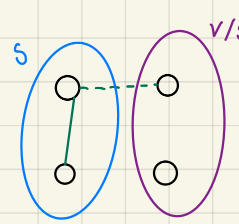
Find a cut $(S, V/S)$ st no edge in A cross cut e.
Add minimum weight crossing edge e for that cut.

$\left[A \leftarrow A \cup \{e\} \right]$

Prim's Algorithm

$A = \emptyset$ while $|A| < |V| - 1$

1) Find edge (u,v) of minumum weight that connects A to an isolated vertex.
2) $A \leftarrow A \cup \{u,v\}$.



Example cut used by Prim's Algorithm.

Proof

In each iteration let cut be specified.
S = set of vertices in tree A.

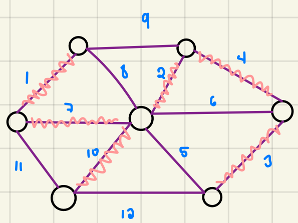
Prim adds minimum weight crossing edge for cut $(S, V \setminus S)$

(Apply CP Theorem)

Kruskal's Algorithm

$A = \emptyset$ while $|A| < |V| - 1$
1) Find edge (u,v) of minimum weight not in A.
2) If no u-v path using edge in A,
   Then $A \leftarrow A \cup \{(u,v)\}$
return A

Example

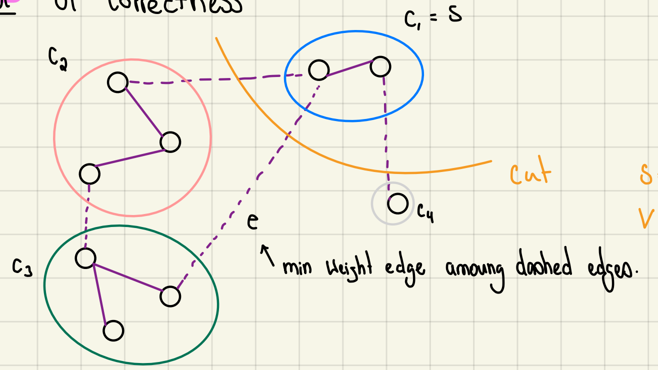


Edges Added

$A = \{1\}$
$A = \{1, 2\}$
$A = \{1, 2, 3\}$
$A = \{1, 2, 3, 4\}$
$A = \{1, 2, 3, 4, 7\}$
$A = \{1, 2, 3, 4, 7, 10\}$

Proof of Correctness

Kruskal add edge e.



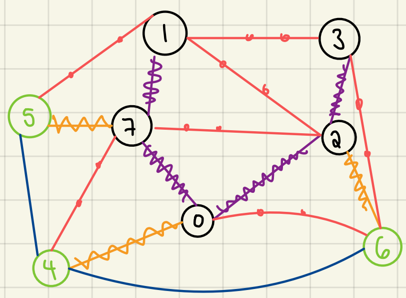
$S = C_1$
$V/S = V/C_1$

September 20th 2021 (Lecture 4)

Prim's Algorithm: Lazy Implementation

Note. A = set of edges, slides T = A.

Prim's Algorithm: Eager Implementation



Challenge: Find min weight edge w/ exactly one endpoint in T.

Observation: For each vertex v, need only min weight edge connecting v to T.

- MST includes at most one edge connecting v to T. Why?
- If MST includes such an edge, it can take cheapest edge. Why?

Grey: Not in tree, not one hop away from our tree
Black: In our tree
Red: Not in tree, One hop away from our tree

Prim (graph G)

```
PQ = empty priority queue of vertices
cost = array of size n
edge = array of size n
Color all vertices grey

Visit(0)
While (PQ not empty)
    u = PQ.DeleteMin()
    A = A ∪ edge[u]
    Visit(u)
```

Visit (vertex u)

```
color u black
for all edges (u,v)
    if v is grey
        colour v red
        PQ.insert(v, w(u,v))
        cost[v] = w(u,v)
        edge[v] = (u,v)
    else if (v is red) and (w(u,v) < cost[v])
        PQ.DecreaseKey(v, w(u,v))
        cost[v] = w(u,v)
        edge[v] = (u,v)
```

Kruskal's Algorithm (slide #48 - Lecture 3)

n = number of vertices
m = number of edges

Worst case: $n \cdot m = n\binom{n}{2}$
$= \Theta(n^2)$!!

Challenge: Would adding edge v-w to tree T create a cycle? If not, add it.

- $E+V$
- V ⟵ run DFS from v, check if w is reachable (T at most V-1 edges)
- $\log V$
- $\log^* V$ ⟵ use the union-find data structure!
- 1

Union-Find

- Maintain a set for each connected component in T.
- If v and w are in same set, then adding v-w would create a cycle.
- To add v-w to T, merge sets containing v and w.

Dynamic Connectivity Problem (Incremental)

1) Start w/ a graph w/ only vertices (no edges)
2) Edges arrive sequentially
3) Keep track of connected components as new edges arrive

Generic Algorithm

```
// A = ∅
for each edge (u,v) in sequence of edges
    if CONNECTED(u,v) == 0
        then UNION(u,v)  // A ← A∪{u,v}
return A
```

CONNECTED(u,v)
   = FIND(u) == FIND(v)

FIND(u): return the component id of u
label

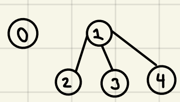
Algorithm 1:

Data Structure id - array of integers
id if vertex i belongs to component k, then id[i]=k.
Initially set id[i]=i for i=0,1,...,n-1.
Find(i) return id[i] // O(1)

UNION(i,j)
   linear scan through id
   for each element equal to id[j] set it to id[i]

$\boxed{3}$ Union(1,3)

0	1	2	3	4
0	1	1	1	1



September 23rd 2021 (Lecture 5)

CONNECTED(i,j) = FIND(i) == FIND(j) // Worst-case $\Theta(n)$

FIND(i) = id[i] // id[id[id[...id[i]...]]] // $O(\log n)$

// Keep calling id recursively until id[i]=i

```
While (id[i] != i) {
    i <- id[i]
} return i
```

UNION(i,j) // already know the route of i and j ($=a$, $=b$) // $O(1)$

id[a] ← b XOR id[b] ← a

WEIGHTED-QUICK-UNION(i,j)

// assume we keep track of size (#nodes) in each tree

```
if tree w/ a is larger
    id[b] <- a
Otherwise
    id[a] <- b
```

Proposition

Weighted-quick-union ensures that all nodes have depth $\leq \log_2(n)$, where n is # vertices

Proof

1) Let v be some node.
   Depth of v increases (by 1) only if root of v changes

2) Root of v changes only if size of v's tree at least doubles

   Let S_j be size in the tree of v after j label changes (root changed)

   $\boxed{n \geq S_j} \geq 2S_{j-1} \geq 2 \cdot 2 \cdot S_{j-2} \geq \ldots \geq 2 \cdot 1$

   $\Rightarrow 2^j \leq n \iff j \leq \log_2(n)$

   # vertices in the tree of v

   $\uparrow 2^{j-1} \cdot S_1 = 2^{j-1}$

   $\uparrow 2^j \cdot S_0 = 2^j$

   $\boxed{S_1 = 2 \quad S_0 = 1}$

Given mixture of m CONNECTED UNION operations, runtime = $O(m \log(n))$

Path Compression

After call to FIND(i) make each node in the i-to-root path a direct descendent of the root.

ex. $2^{16} \xrightarrow{\log} 16 = 2^4 \xrightarrow{\log} 4 \xrightarrow{\log} 2 \xrightarrow{\log} 1$.

(recursive properties)

$\rightarrow 2^{16}$ base $\log^* n$ # times repeatedly take at most 1.

Better Bound on Runtime

$O(m\alpha(n))$

$\alpha(n) = \min\{k \; A_k(1) \geq 1\}$

$A_0(1) = 2$

$A_1(1) = 3$

$A_2(1) = 7$

$A_3(1) = 2047$

$A_4(1) \gg 2^{2047} \gg 10^{80}$

Runtime upper bound for weighted-quick-union w/ path compression:

$O((m\log^* n) + n)$, m is # operations

$\hookrightarrow \log^*(n) = \begin{cases} \log^*(\log(n)) & \text{if } > 1 \\ 0 & \text{if } = 1 \end{cases}$

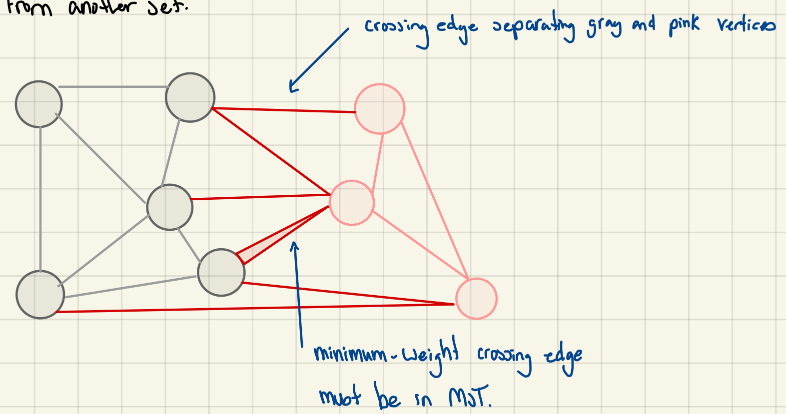
Single Source Paths Problem

1) Warmup: Single-pair shortest path prob.
   source vector s
   destination vertex t

2) Single-source shortest paths problem ☆

A Useful Tool for finding the MST: Cut Property by Josh Hug on YT

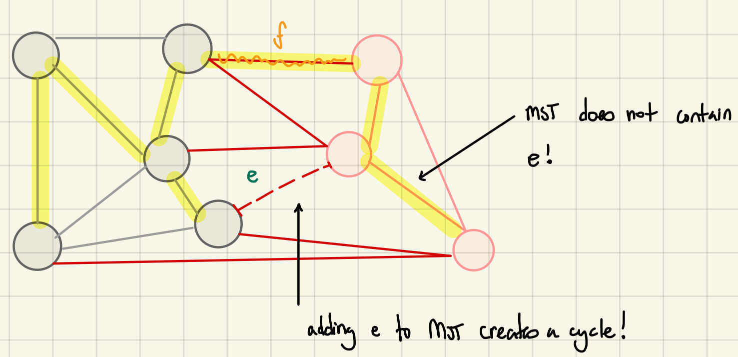
- A cut is an assignment of a graph's nodes to two non-empty sets.
- A crossing edge is an edge which connects a node from one set to a node from another set.



- Cut Property: Given any cut, minimum weight crossing edge is in the MST.

Proof: Suppose that the minimum crossing edge e were not in the MST.

- Adding e to the MST creates a cycle.
- Some other edge f must also be a crossing edge
- Removing f and adding e is a lower weight spanning tree
- Contradiction!



Generic MST Finding Algorithm

Start with no edges in the MST
- Find a cut that has no crossing edges in the MST.
- Add smallest crossing edge to the MST.
- Repeat until V-1 edges.

9/27 September 27th 2021 (Lecture 6) By: ZiHao / Rob

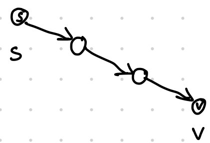
An s-v path (of # edges k) is denoted as $p = (v_0, v_1, \dots, v_k)$

$s = v_0$ $v = v_k$

Sequence of edges $\underbrace{(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)}_{k \text{ edges}}$

$$w(p) = \sum_{j=1}^{k} w(v_{j-1}, v_i)$$

path p



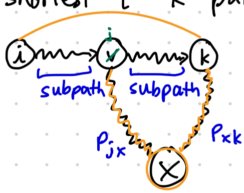
Optimal Substructure

An optimal solution to a problem contains within it an optimal solution to subproblems

example problem : Find shortest path from $\overbrace{i \text{ to } k}^{\text{vertices}}$

example subproblem: Find shortest path from j to k

shortest i - k path



$w(P_{jk}) < w(P_{\cancel{v}^{\,j}x}) + w(P_{xk})$

this 'a' could be 'd'

Lemma

Subpaths of shortest paths are also shortest paths

Formally: Let $P_k = (v_1, v_2, \dots, v_k)$ be shortest $v_1 - v_k$ path

Take arbitrary i, j such that $1 \le i \le j \le k$

and let $P_{ij} - (v_i, v_{i+1}, \dots, v_j)$ be subpath of P_{1k}

Then P_{ij} is a shortest $v_i - v_j$ path

Proof Shortest $v_1 - v_k$ path

Claim of proof $v_i \rightsquigarrow v_j$ is shortest path.

$$v_1 \xrightarrow{P_{1i}} v_i \xrightarrow{P_{ij}} v_j \xrightarrow{P_{jk}} v_k$$

Proof of Claim

Suppose $\exists$ shorter path P'_{ij} $(w(P'_{ij}) < w(P_{ij}))$

Then $w(P_{1i}) + w(P'_{ij}) + w(P_{jk}) < w(P_{1i}) + w(P_{ij}) + w(P_{jk})$

(P_{1k} is not shortest)

Notation

Let S be source vertex

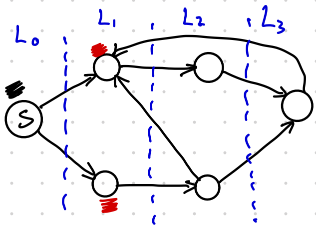
Let $\delta(u,v)$ be weight of shortest u-v path

Let $d[v]$ be upper bound on weight of shortest s-v path

Let $\pi[v]$ be predecessor of v in the algorithm's current best-known shortest s-v path

"predecessor array"

BFS for unweighted graphs (pg 6 of lecture6.pdf)



This was drawn over many times

Runtime of BFS

$O(|V| - |E|)$

Tool ← Important

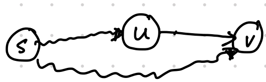
RELAX (u,v)

If $d[u] + w(u,v) < d[v]$

$d[v] \leftarrow d[u] + w(u,v)$

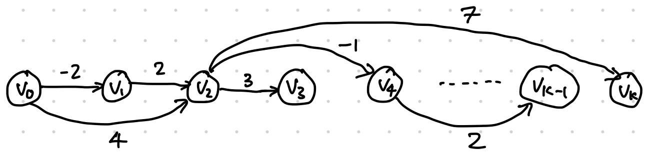
$\pi[v] \leftarrow u$

Updating path to v with u as predecessor.



Note: $\pi[v]$ is an array of predecessors. Think as it stores the path (linked list)

Weighted DAG - Directed Acyclic Graph



Topologically Sorted

Algorithm - At the end of this algorithm you'll have a predecessor array.

1) Use topological sort (via DFS) to obtain topological ordering of vertices
2) For each vertex u (in topological order)

    For all adjacent vertices v, call RELAX(u,v)

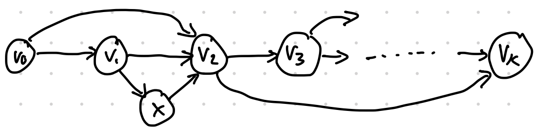
Proof of Correctness

Consider shortest path from s to v $(v_0, \dots, v_k)$ with $v_0 = s$ and $v_k = v$

Since the vertices are processed in topological order,

the sequence of RELAX calls include subsequence

RELAX(v_0, v_1), RELAX(v_1, v_2), ..., RELAX(v_{k-1}, v_k)



RELAX (v_0, v_1)
RELAX (v_0, v_2)
RELAX (v_1, x)
RELAX (v_1, v_2)
RELAX (x, v_2)
RELAX (v_2, k)
RELAX (v_2, v_3)

Consider the operations RELAX(v_0, v_1), ..., RELAX(v_{k-1}, v_k)

Claim

$$d[v] = \overbrace{d[v_k] = \delta(s, v_k)}^{\star} = \delta(s, v)$$

Proof (induction on k)

Base case: $k=0$ $\quad d[v_0] = d[s] = 0 = \delta(s,s)$

IH: Just before RELAX(v_{j-1}, v_j) we know $d[v_{j-1}] = \delta(s, v_{j-1})$

IS: After RELAX(v_{j-1}, v_j), $d[v_j] = \delta(s, v_j)$

(Proof) $d[v_j] \le d[v_{j-1}] + w(v_{j-1}, v_j) \overset{(IH)}{=} \delta(s, v_{j-1}) + w(v_{j-1}, v_j) = \delta(s, v_j)$

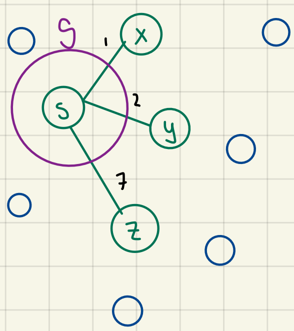
October 4th 2021 (Lecture 7)

Dijkstra's Algorithm

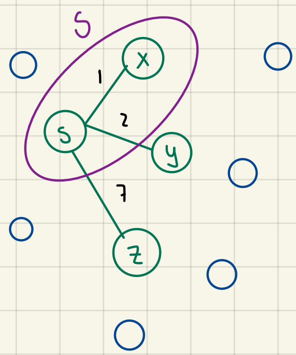
Input: A simple directed graph G w/ nonnegative edge-weights and a source vertex s in G
Output: A number d[u] for each vertex u in G such that d[u] is th weight of th shortest path in G from s to u

Dijkstra(V, E, s): S= set of vertices
  S = {s} simple source
  d[s] = 0 ← minimum cost
  While S ≠ V
    For all v ∉ S such that there is an edge (u,v) for some u ∈ S: ← cheapest and compare!
      Set cost c[v] = min{(u,v): u in S} d[u] + w(u,v)
    Of the vertices, let v be one for which c[v] is minimum
    Add v to S
    Set d[v] = c[v]

ex.



(s,x) → 0+1 = 1
(s,y) → 0+2 = 2
(s,z) → 0+7 = 7
→
RELAX(u,v)
  if d[u] + w(u,v) < d[v]
    then d[v] = d[u] + w(u,v)
    π[v] ← ~~π[u]~~ u



Dijkstra(V, E, s):
  For v in V   set to infinite
    d[v] = ∞; π[v] = null;
  d[s] = 0
  S = ∅
  Q = BuildPriorityQueue(V, d)
    While Q not empty
      u = DeleteMin(Q)
      S = S ∪ u
      For v in Adj[u]
        Relax(u,v)

RELAX(u,v):
  if d[u] + w(u,v) < d[v]
    d[v] = d[u] + w(u,v)
    π[v] = u

Dijkstra vs Prim

Dijkstra(V,E,s):	Prim(V,E,s):
For v in V	**For** v in V
d[v] = ∞; π[v] = null;	d[v] = ∞; π[v] = null;
d[s] = 0	d[s] = 0
S = ∅	S = ∅
Q = BuildPriorityQueue(V, d) △	Q = BuildPriorityQueue(V, d)
While Q not empty	**While** Q not empty
n calls → u = DeleteMin(Q) ☆	u = DeleteMin(Q)
S = S ∪ u	S = S ∪ u
For v in Adj[u]	**For** v in Adj[u]
If d[u] + w(u,v) < d[v]	**If** w(u,v) < d[v]
d[v] = d[u] + w(u,v)	d[v] = w(u,v)
π[v] = u	π[v] = u
at most m calls → UpdatePQ(v, d[v]) ○	UpdatePQ(v, d[v]) ← Decrease operation!

← We don't care about the path for Prim!

△ $O(n)$ for binary or Fibonacci heap
☆ $O(\log n)$/call for binary or Fibonacci heaps
○ $O(\log n)$/call for binary heap
$O(1)$/call for Fibonacci heap

Correctness

In any iteration... $\forall\, v \in S$

Claim: For all v in S, the algorithm's path P_v from s to v is a shortest s-v path

Proof by Induction (Induction on $|S|$)

(I) **Base Case:** $|S| = 1$, with $S = \{s\}$, we know $d[s] = 0 = \delta(s,s)$ ✓

Clearly, $P_s = (s)$ is a shortest s-s path (of length zero!)

Induction Step: Suppose the claim holds for $|S| = k$
Prove that it holds for $|S| = k+1$

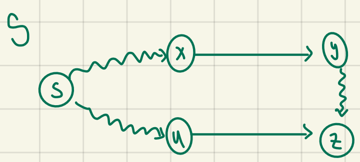
(III)

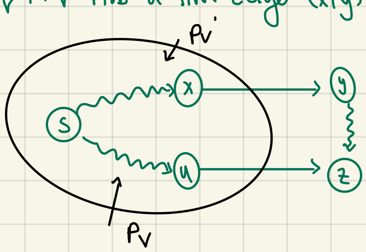
(claim holds for $|S| = k+1$)

Suppose claim holds for $|S| = k$

Let $|S| = k$ and suppose Alg. is about to add v to S and let P_v be the path to v.

Consider an arbitrary alternative path P_v'. P_v' has a first edge (x,y) that crosses the cut $(S, V \setminus S)$





Suppose $\exists\, P_v$ ...

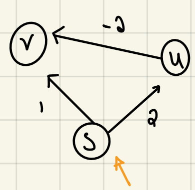
$w(P_v') < w(P_v)$

Path P_v' cannot be shorter than P_v

$$\begin{aligned} w(P_v') &\geq \delta(s,x) + w(x,y) \\ &= d[x] + w(x,y) && \text{(inductive hypothesis)} \\ &\geq d[u] + w(u,v) && \text{(v is next vertex added to S)} \\ &= \delta(s,u) + w(u,v) && \text{(induction hypothesis)} \\ &= w(P_v) \end{aligned}$$

Dijkstra's Algorithm - Negative Weights

What would Dijkstra do?



"Greed is good"

"Greed is not good (when a graph has negative edge weights)"

Bellman-Ford Algorithm

Path Relaxation Property:

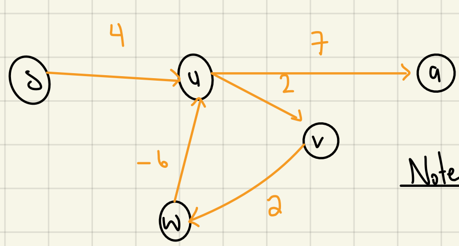
number → d; Predecessor Array (Backtrack to source) (solution) → π

Let $p = (v_0, v_1, \ldots, v_k)$ be a shortest path from v_0 to v_k. Initialize d and π w/ source s. Suppose that a sequence of Relax calls occur which includes the subsequence:

- RELAX(v_0, v_1), RELAX(v_1, v_2), ..., RELAX(v_{k-1}, v_k)

Then after the last Relax call in this subsequence and for all times thereafter, we have

$$d[v_k] = \delta(s, v_k)$$



Note: $u \longrightarrow v \longrightarrow w \longrightarrow u$ is a negative cycle.

* Priority does not matter, can view in any order!

◦ Dijkstra has lower runtime when no negative edge weights.

An observation: Suppose shortest path from vertex s to vertex t consists of 1 edge $p = (v_0, v_1)$ w/ $s = v_0$ and $t = v_1$.

Then after calling RELAX(v_0, v_1).

$$d[t] = d[v_1] = \delta(v_0, v_1) = \delta(s, t)$$

→ Shortest path from s to t has been found!

How to ensure RELAX(v_0, v_1) gets called?

Initialize d and π w/ source s
For each edge $(u,v) \in E$
    RELAX(u,v)

An observation: Suppose shortest path from vertex s to vertex t consists of 2 edge
$p = (s = v_0, v_1, v_2 = t)$

Then after calling RELAX(v_0, v_1), RELAX(v_1, v_2):

$$d[t] = d[v_2] = \delta(v_0, v_1) = \delta(s, t)$$

$\rightarrow$ Shortest path from s to t has been found!

How to ensure RELAX(v_0, v_1), RELAX(v_1, v_2) gets called?

Initialize d and π w/ source s
For $j = 1 \rightarrow 2$
    For each edge $(u,v) \in E$
        RELAX(u,v)

- [ ] If no negative cycles, shortest path from vertex s to vertex t consists of (at most) $n-1$ edges: $p = (v_0, v_1, \ldots, v_k)$ w/ $k \leq n-1$.

After calling RELAX(v_0, v_1), RELAX(v_1, v_2), ..., RELAX(v_{k-1}, v_k):

$$d[t] = d[v_k] = \delta(v_0, v_k) = \delta(s, t)$$

(shortest path from s to t has been found!)

How to ensure subsequence RELAX(v_0, v_1), ..., RELAX(v_{k-1}, v_k) of calls occur?

Initialize d and π w/ source s
For $j = 1 \longrightarrow n-1$
    for each edge $(u,v) \in E$
        RELAX(u,v)

BELLMAN-FORD(G, V, s)
  Initialize d and π w/ source s
  for $j = 1 \to n-1$
    for each edge $(u,v) \in \mathcal{E}$
      RELAX(u,v)
  for each edge $(u,v) \in \mathcal{E}$
    If $d[v] > d[u] + w(u,v)$
      Return False
  Return True

RELAX(u,v)
  If $d[u] + w(u,v) < d[v]$
    $d[v] = d[u] + w(u,v)$
    $\pi[v] = u$

Correctness

Claim 1: If there are no negative cycles:

a) The Algorithm correctly finds the shortest paths ($d[v] = \delta(s,v)$ for all v) and predecessor array is correct.

Proof: This we already showed in the derivation of the algorithm! The desired subsequence of calls to RELAX occurs, which is all that is required.

B) The algorithm returns True.

Proof: We only need to verify that...
$$d[v] \le d[u] + w(u,v) \text{ for all edges } (u,v) \in E$$
From Claim 1 (A), this is equivalent to...
$$\delta(s,v) \le \delta(s,u) + w(u,v) \text{ for all edges } (u,v) \in E.$$

This must be the case. Why? An s-v path that first visits u then follows edge (u,v) cannot have less weight than the shortest s-v path.

Claim 2: If there is a negative cycle, the algorithm detects it and returns False.

Proof: Assume there is a negative cycle...

$(v_0, v_1, v_2, \dots, v_k)$ where $[v_0 = v_k]$

$\sum_{j=1}^{k} w(v_{j-1}, v_j) < 0$

Suppose for contradiction that [algorithm returns true]

$\iff$ All edges $(u,v) \in E$, where $d[v] \le d[u] + w(u,v)$

Sum over edges in cycle: $\sum_{j=1}^{k} d[v_j] \le d[v_j] \le \sum_{j=1}^{k} d[v_{j-1}] + w(v_{j-1}, v_j)$

$\sum_{j=1}^{k} d[v_j] = \sum_{j=1}^{k} d[v_{j-1}]$

$\Rightarrow 0 \le \sum_{j=1}^{k} w(v_{j-1}, v_j)$

October 7th, 2021 by ZiHan / Rob (Lecture 8)

Single source shortest paths = Bellman Ford

All ~~paths~~ Pairs shortest paths

These notes include everything written on the chalkboard and includes some information from the slides, so make sure to check the slides

Cutoff for Midterm is Shortest Paths

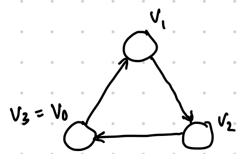
Bellman Ford (continued)

Proof of Correctness

Assume there is negative cycle

$(v_0, v_1, v_2, \ldots, v_k)$ $[v_0 = v_k]$

$$\sum_{j=1}^{k} w(v_{j-1}, v_j) < 0$$



$(v_0, v_1), (v_1, v_2), (v_2, v_3)$

Suppose for contradiction that

[algorithm returns true]

$\Updownarrow$

all edges $(u,v) \in E$: $d[v] \le d[u] + w(u,v)$

Start from 1 Start from 0

Sum over edges in cycle : $\sum_{j=1}^{k} d[v_j] \le \sum_{j=1}^{k} d[v_{j-1}] + w(v_{j-1}, v)$

$$\sum_{j=1}^{k} d[v_j] = \sum_{j=1}^{k} d[v_{j-1}]$$

∴ subtract from both sides using this equality

$$0 \le \sum_{j=1}^{k} w(v_{j-1}, v)$$

Single Source Shortest Path Algorithms

Type of Graph	Algorithm	Time Complexity
unweighted graph	BFS	$O(n+m)$
DAG	Topological Sort / DFS-Based	$O(n+m)$
weighted directed graph (non-negative weights)	Djikstra's - Binary Heap Djikstra's - Fibonacci Heap	$O(m \log n)$ $O(n \log n + m)$
weighted directed graph (any weights)	Bellman-Ford	$O(nm)$

All-Pairs Shortest Path Algorithms

first approach - run single-source shortest paths algorithms n times, once per choice of source vertex

Type of Graph	Algorithm	Time Complexity	Dence Graph Time Complexity
non-negative weights	Djikstra's - Binary Heap	$O(nm \log n)$	$O(n^3(\log n))$
	Djikstra's - Fibonacci Heap	$O(n^2 \log n + mn)$	$O(n^3)$
any weights	Bellman - Ford	$O(n^2 m)$	$O(n^4)$

All-Pairs Shortest Paths

Problem: Find ~~the~~ path from i to j
^ shortest

Subproblem: Find the shortest path from i to j where intermediate vertices belong to

$\{1, 2, \ldots, \cancel{k}\}$ $n-1$

= Find shortest path from i to j where intermediate vertices belong to

$\{1, 2, \ldots, n\}$

Slides

Equal or Bigger

Need to store upper bound on shortest paths for every pair of vertices.

Switch from array d to matrix D of size $m \times n$.

D_{ij} = Upper bound on shortest path from i to j.

Switch from predecessor array π to predecessor matrix Π.

Π_{ij} = Predecessor of j in some shortest path from source i.

What is cheaper? Excluding n or Including n.

For $k = 0, 1, \ldots, n$: let $D_{ij}^{(k)}$ be the weight of the shortest path from i to j for which all intermediate vertices are in $\{1, \ldots, k\}$

$D_{ij}^{(k)}$ ↑ restrict intermed vertices to the set $\{1, 2, \ldots, k\}$

all intermediate vertices in $\{1, \ldots, k\}$

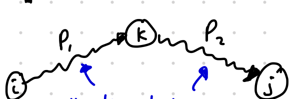
$D_{ij}^{(k)}$ ← restart intermediate vertices to the set $\{1, 2, \ldots, k\}$

Case 1

all intermediate vertices in $\{1, \ldots, k-1\}$

$D_{ij}^{(k)} = D_{ij}^{(k-1)}$

Case 2



all intermediate vertices in $\{1, \ldots, k-1\}$

$D_{ij}^{(k)} = D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$

Let p be a shortest path from i to j. Clearly, all intermediate vertices in path p are in $\{1, \ldots, n\}$. Also, we can break down p into at most 2 paths whose intermediate vertices are in $\{1, \ldots, n-1\}$.
Slides

These set problems are getting easier and easier, with more and more restrictions.

Floyd-Warshall Algorithm → Try trace through w/ a 4 by 4 example! Or/And Code the Algorithm.

Recurrence: $D_{ij}^{(k)} = \min\{ D_{ij}^{(k)},\ D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \}$

Base Case: $D_{ij}^{(0)} = w(i,j)$ → Because no intermediate vertices can be used — Why?

★ **Floyd-Warshall(W)** (CLRS p.696)

Eventually you are allowed to use vertices 1 to n.

```
D(0) = W
for k = 1 -> n
   for i = 1 -> n
      for j = 1 -> n
         Dij(k) = min { Dij(k), Dik(k-1) + Dkj(k-1) }
return D(n)
```

Time Complexity $O(n^3)$

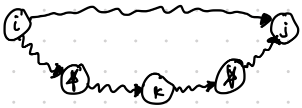
What about that predecessor matrix? How do we print a shortest path?

Case 1 $D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \geq D_{ij}^{(k-1)}$
Path will not change. Reuse predecessor from before: $\Pi_{ij}^{(k)} = \Pi_{ij}^{(k-1)}$

Case 2 $D_{ik}^{(k-1)} + D_{kj}^{(k-1)} < D_{ij}^{(k-1)}$
Path updates to (path from i to k)(path from k to j). $\Pi_{ij}^{(k)} = \Pi_{kj}^{(k-1)}$

"Set predecessor of j in shortest path from source i using intermediate vertices in $\{1,\dots,k\}$ to be predecessor of j in shortest path from k using intermediate vertices in $\{1,\dots,k-1\}$."

Correctness: $D_{ij}^{(n)}$ is weight of shortest path with intermediate vertices in $\{1,\dots,n\}$. This is the shortest path itself!



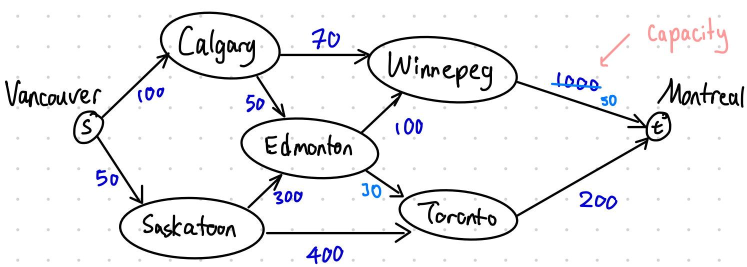
(*) belong to $\{1, 2, \dots, k-1\}$

Midterm # 1 – Cut Off!

Flow Network Example

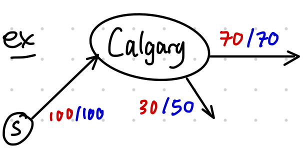
☆ We can use "both" paths simultaneously.

find a way to send as much stuff from Vancouver to Montreal



- the amount that goes into a vertex must also come out.
- the numbers written in blue is the capacity of an edge, the "max amount of flow that can be sent using that edge" at a time.

↳ ex



flow/capacity

Flow Network

Abstraction for material flowing through the edges.

Digraph $G = (V, E)$ w/ source $s \in V$ and sink $t \in V$*

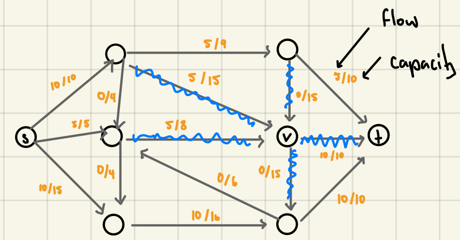
Nonnegative integer capacity $c(e)$ for each $e \in E$

* No parallel edges
No edge enters s
No edge leaves t

Maximum Flow Problem

Definition An st-flow (flow) f is a function that satisfies:

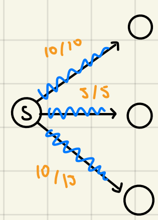
- For each $e \in E$: $0 \le f(e) \le c(e)$ [capacity]
- For each $v \in V - \{s,t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } e} f(e)$ [flow conservation]



inflow at v: $5+5+0 = 10$

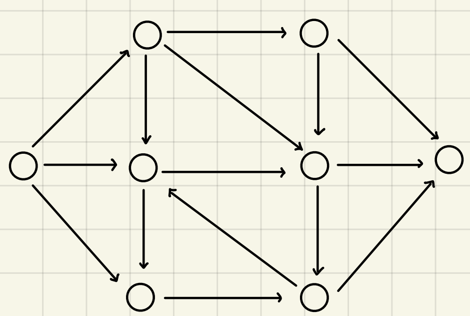
outflow at v = $10 + 0 = 10$

Definition The value of a flow f is: $val(f) = \sum_{e \text{ out of } s} f(e)$.



Value = $5+10+10 = 25$.

Graph



October 14th 2021 (Lecture 9)

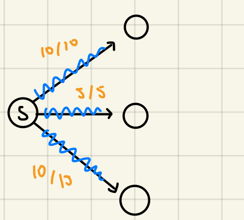
We have that $0 \leq f(e) \leq c(e)$, Where f is flow and c is Capacity.

$$\boxed{\text{Value of flow} \quad v(f) = \sum_{e \text{ out of } s} f(e)}$$

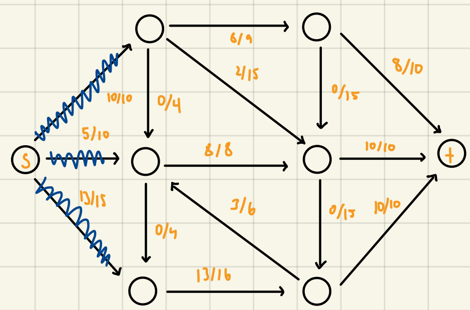
Max-Flow Problem:

find a flow of maximum value.

V= 10 + 5 + 10 = 25



V= 10 + 5 + 13 = 28



Minimum Cut Problem

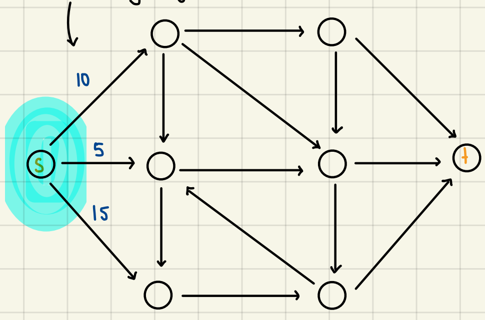
Definition A st-Cut (cut) is a partition (A, B) of the vertices w/ s ∈ A and t ∈ B.

Definition Its capacity is the sum of the capacities of the edges from A to B.

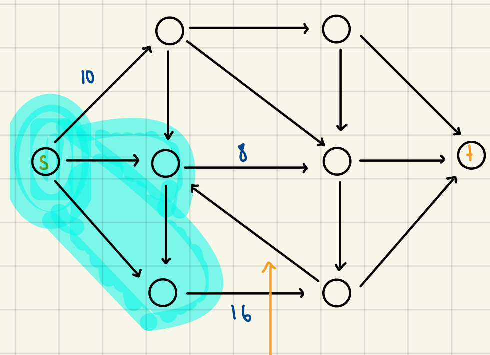
$$\boxed{cap(A,B) = \sum_{e \text{ out of } A} c(e)}$$

Capacity = 10 + 8 + 16 = 34

crossing edges



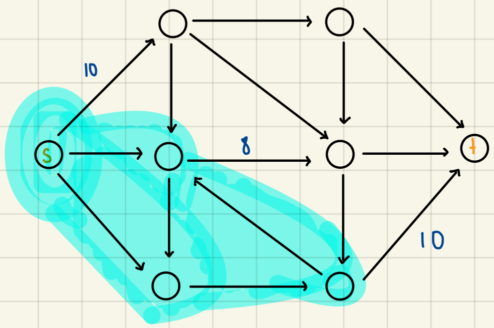
Capacity = 10 + 5 + 15 = 30



don't count edges from B to A

Min-cut Problem:

Find a cut of minimum capacity.



Capacity = 10 + 8 + 10 = 28

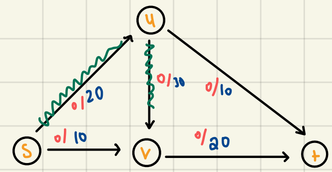
Towards a Max-flow Algorithm

Greedy Algorithm

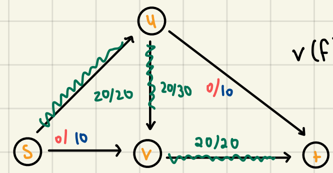
- Start w/ $f(e) = 0$ for all edge $e \in E$.
- Find an $s \rightsquigarrow t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck.

Simple Example

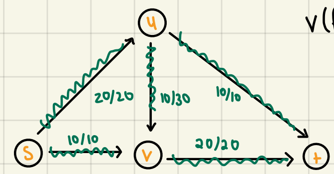
★ Arbitrarily Pick a Path to saturate!



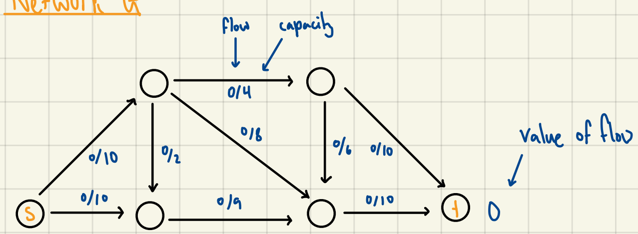
$v(f) = 20$



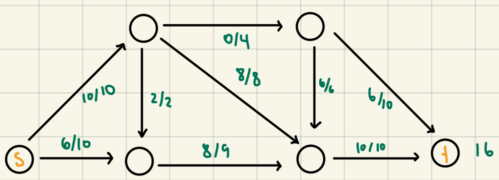
$v(f) = 30$



Network G

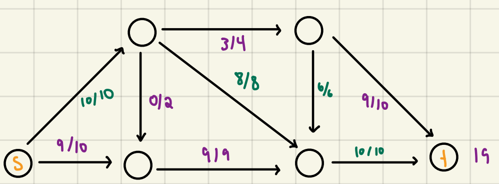


Repeat find paths w/ remaining space, until...



ending flow value = 16

Better Solution



ending flow value = 19

Residual Graph G_F

Let $V(G_F) = V(G)$

Forward Edge

For edge $e = (u,v) \in E(G)$
if $f(e) < c(e)$
then add e to G_F
with residual capacity $c(e) - f(e)$

Backward Edge

For edge $e = (u,v) \in E(G)$
if $f(e) > 0$
then add $e' = (u,v)$
with residual capacity $f(e)$

Claim

Let f' be a flow in G_F
Then $f + f'$ is a flow

Proof

Let e be a forward edge $\longrightarrow$ Want $0 \leq f(e) + f'(e) \leq c(e)$

$\updownarrow$

residual capacity of e. $c(e) - f(e)$
then $f(e) + f'(e) \leq \cancel{f(e)} + c(e) - \cancel{f(e)}$

Suppose $e = (v,u)$ be a backward edge
residual capacity of e is $f(u,v)$.
~~$f'(v,u) = -f(u,v)$~~

$\star\ f'(u,v) = -f'(v,u)$

① $f(u,v) - \underbrace{f'(v,u)}_{\geq 0} \leq c(u,v)$

$f'(v,u) \leq f(u,v)$

$f(u,v) + f'(u,v) = f(u,v) - f'(v,u)$
$\geq f(u,v) - f(u,v) = 0$

$\downarrow$

Augmenting Path

Definition An augmenting path is a simple $s \rightsquigarrow t$ path P in th residual graph G_f.

Definition The bottleneck capacity of an augmenting P is th minimum residual capacity of any edge in P.

Key Property Let f be a flow and let P be an augmenting path in G_f.
Then f' is a flow and $val(f') = val(f) + bottleneck(G_f, P)$.

Augment(f, c, P)

```
b ← bottleneck capacity of path P.
FOREACH edge e ∈ P.
   IF (e ∈ E) f(e) ← f(e) + b.
   ELSE  f(e^R) ← f(e^R) − b.
RETURN f.
```

Ford-Fulkerson Algorithm

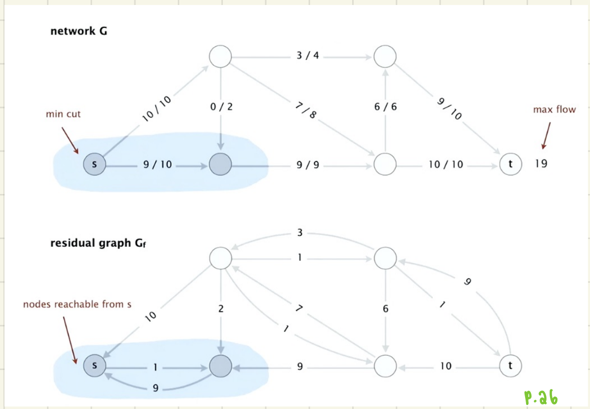
Ford-Fulkerson augmenting path algorithm.

- Start w/ $f(e) = 0$ for all edge $e \in E$.
- Find an augmenting path P in th residual graph G_f.
- Augment flow along path P.
- Repeat until you get stuck.

Ford-Fulkerson(G, s, t, c)

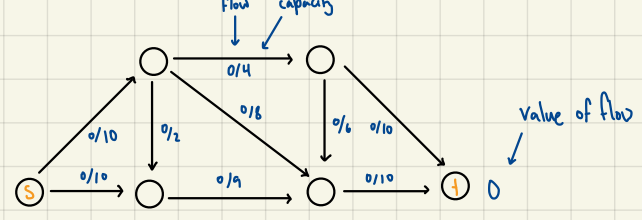
```
FOREACH edge e ∈ E: f(e) ← 0.
   G_f ← residual graph.
   WHILE (there exists an augmenting path P in G_f).
      f ← AUGMENT(f, c, P).
      Update G_f.
   RETURN f.
```

Note :

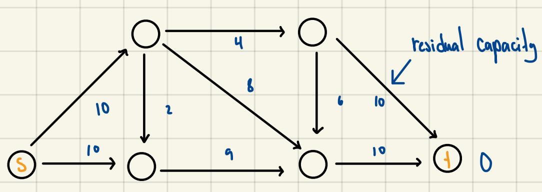


Example ☆ View Demo in Lecture Slides 10 p. 21 – 26

Network G



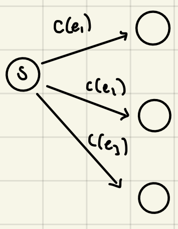
Residual Graph G_f



October 21st 2021 (Lecture 10)

Integer Capacities (Assumption)

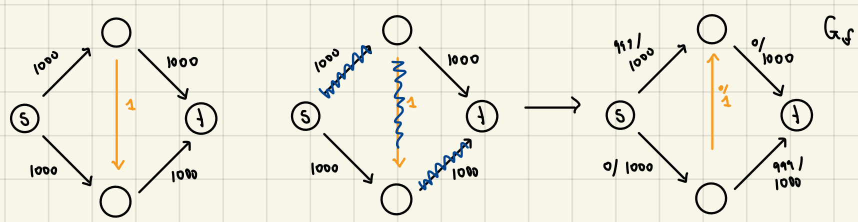
Given flow f If $\exists$ Augmenting path P in G_f then new flow f' will have $v(f') \geq v(f)+1$



All flows f

$$v(f) \leq \sum_{e \text{ out of } s} c(e)$$
$$= C$$

ex.



Any s-t (A,B) for any flow f $v(f) \leq cap(A,B)$

$$Cap(A,B) = \sum_{e \text{ out of } A} c(e)$$

[Missed Notes 22.06 - 23:10]

edge $e=(u,v)$ is "into A" if crossing edge with $u \in B$ and $v \in A$.

Flow Value Lemma

Let (A,B) be s-t cut $v(f) = \sum_{e \text{ out of } A} f(e) - \overbrace{\sum_{e \text{ into } A} f(e)}^{\geq 0}$

[Missed Notes 27:08 – 33:09]

1) $\sum_{v \in A} \sum_{\substack{e \text{ out of } v \\ \text{and out of } A}} f(e) - \sum_{v \in A} \sum_{\substack{e \text{ into } v \\ \text{and into } A}} f(e)$

[Missed Notes 35:02 – 36:45]

$$= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

[Missed Notes 38:08 – 41.02]

Any s-t cut (A,B) (Any flow)

$$v(f) \leq \sum_{e \text{ out of } A} f(e)$$

$$\leq \sum_{e \text{ out of } A} c(e) = cap(A,B)$$

Weak Duality (Relationship between flows and cuts)

Let f be any flow and (A,B) be any cut. Then...

$$\text{Max flows } f \quad v(f) \leq \text{Min s-t cuts } (A,B) \quad cap(A,B)$$

Proof

$$v(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

flow-value lemma

$$\leq \sum_{e \text{ out of } A} f(e)$$

$$\leq \sum_{e \text{ out of } A} c(e)$$

$$= cap(A,B) \blacksquare$$

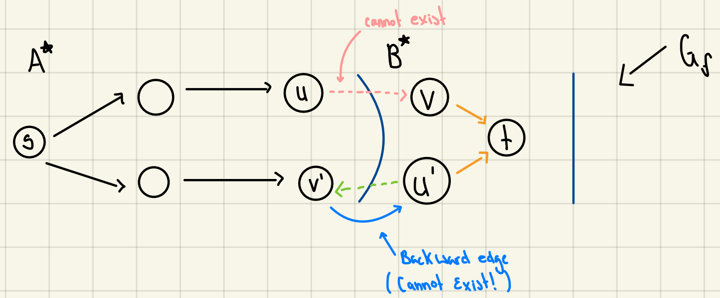
Proof of F·F Finds Max Flows

At termination no s-t path in G_f.

Let A^* be a set of vertices reachable from s in G_f.
Let $B^* = V \setminus A^*$.

$s \in A^*$, $t \in B^*$

ex.



[Missed Notes 57:58 – 1:03:48]

Shortest Augmenting Path

Q Which augmenting path?
A. The one with the fewest number of edges. (can find w/ BFS)

$O(mn \cdot m)$ — cost of BFS; # of paths

Shortest-Augmenting-Path(G, s, t, c)

```
FOREACH edge e ∈ E: f(e) ← 0.
  G_f ← residual graph.
  WHILE (there exists an augmenting path in G_f).
    P ← BREADTH-FIRST-SEARCH(G_f, s, t)
    f ← AUGMENT(f, c, P).
    Update G_f.
  RETURN f.
```

Overview of Analysis

L1. Throughout the algorithm, length of the shortest path never decreases.

L2. After at most m shortest path augmentations, the length of the shortest augmenting path strictly increases.

Theorem. The shortest augmenting path algorithm runs in $O(m^2 n)$ time.

Proof:

- $O(m+n)$ time to find shortest augmenting path via BFS.
- $O(m)$ augmentations for paths of length K
- If there is an augmenting path, there is a simple one.
  - $\Rightarrow 1 \leq K < n$
  - $\Rightarrow O(mn)$ augmentations. ∎

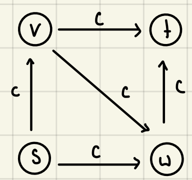
October 25th 2021 (Lecture 11)

Bad Case for Ford-Fulkerson

Q. Is generic Ford-Fulkerson algorithm poly-time in input size? ← m, n, and log C
A. No. If max capacity is C, then algorithm can take $\geq C$ iterations.

- $s \to v \to w \to t$
- $s \to w \to v \to t$
- $s \to v \to w \to t$
- $s \to w \to v \to t$
- ...
- $s \to v \to w \to t$
- $s \to w \to v \to t$

← each argument path sends only 1 unit of flow (# augmenting paths = 2C)



$m \geq n-1$
BFS $\cdot\ O(n+m)$
$= O(m)$

Choosing Good Augmenting Paths

Choose augmenting paths with:
- Max bottleneck capacity.
- Sufficiently large bottleneck capacity.
- Fewest number of edges.

- [ ] Edmonds-Karp 1972 (USA)
- [ ] Dinic 1970 (Soviet Union)

Show: # augmenting paths : $2mn$

For flow f and vertex v, let $\delta_f(s,v)$ be length of shortest s-v path in G_f ("shortest" means least number of edges)

Lemma

If Edmonds-Karp is run on a flow network, then throughout the algorithm, for all vertices $v \in V \setminus \{s,t\}$, the shortest path distance $\delta_f(s,v)$ never decreases.

$\delta_f(v)$ is distance of v from s in residual graph G_f.

Shortest path distance

$$\underset{\text{later time}}{\delta_{f'}(v)} \geq \underset{\text{earlier time}}{\delta_f(v)}$$

Proof (of Lemma)

Let f be the flow just prior to first augmentation that decreases some (shortest path) distance, and let f' be the next flow

Among all vertices whose distance decreases from G_f to $G_{f'}$, let v be the vertex with minimum $\delta_{f'}(s,v)$

Let P be shortest s-v path in $G_{f'}$, and let u be predecessor of v in P

After Augmentation: $\underbrace{s \rightsquigarrow u \to v}_{P \text{ is path in } G_{f'}}$

---

(1) $\delta_{f'}(s,v) = \delta_{f'}(s,u) + 1$

Because $\delta_{f'}(s,u) < \delta_{f'}(s,v) \Rightarrow \delta_{f'}(s,u) \geq \delta_f(s,u)$ (2)

Claim: In G_f, shortest s-u path is of the form $s \rightsquigarrow v \to u$

Claim $\Rightarrow$ $\delta_f(s,u) = \delta_f(s,v) + 1$ (3)

Suppose $\exists$ augmenting path such that $G_f \longrightarrow G_{f'}$ such that $\delta_{f'}(v) < \delta_f(v)$

Let v be vertex in $G_{f'}$ such that distance $\downarrow$ and among all such vertices, $\delta_{f'}(v)$ is minimum

(1) $\delta_{f'}(v) = \delta_f(u) + 1$ (predecessor in $G_{f'}$)
(2) $\delta_{f'}(u) \geq \delta_f(u)$ (u's distance can't $\downarrow$)

Suppose $\delta_{f'}(u) < \delta_f(u)$
then $\delta_{f'}(u) = \delta_{f'}(v) - 1$

[missing notes 27:00 – 33:40]

$$\begin{aligned}
\delta_{f'}(s,v) &= \delta_{f'}(s,u) + 1 && (1)\\
&\geq \delta_f(s,u) + 1 && (2)\\
&= \delta_f(s,v) + 2 && \text{(3)}
\end{aligned}$$

$\lightning \leftrightarrow$ Shortest path dist. from s to v actually increased by 2.

Proof

First, we claim (u,v) is not an edge in G_f.
Suppose (for contradiction) that (u,v) is edge in G_f.

$\downarrow$

$$\begin{aligned}\delta_f(s,v) &\le \delta_f(s,u)+1 && \text{(triangle inequality)}\\ &\le \delta_{f'}(s,u)+1 && (2)\\ &= \delta_{f'}(s,v) && (1)\ \lightning\end{aligned}$$

Indeed, (u,v) is not in G_f. But! (u,v) is in $G_{f'}$.

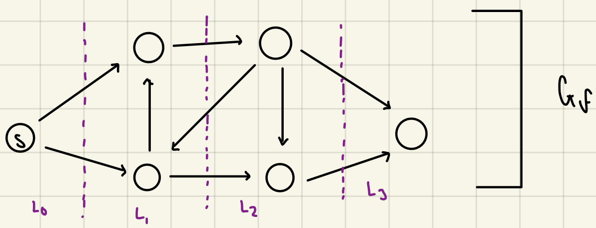
=> (v,u) belongs to the path along which flow was augmented in G_f.
=> (v,u) is edge in s.p. from s to u.

Edge (v,u) is in G_f and (v,u) belongs to the shortest s-u path in G_f.
v is a predecessor of u in shortest s-u path in G_f $(\delta_f(u) = \delta_f(v)+1)$

Theorem

If Edmonds-Karp is run on a flow network, then the algorithm performs $O(mn)$ flow augmentations.

Proof



Let P be augmenting path in G_f s.t. $P = (v_0, v_1, \dots, v_j)$ because P is a shortest path, $\forall i : v_i \in L_i$.
At least one edge (v_i, v_{i+1}) in P will be "bottleneck edge" — augmentation of flow uses all of this edge's residual capacity.
After augment: (v_i, v_{i+1}) is removed! and
we add backward edge (v_{i+1}, v_i)

Suppose that later, in some new residual graph $G_{f'}$, the edge (v_i, v_{i+1}) comes back after augment flow in $G_{f'}$.

=> (v_{i+1}, v_i) belongs to shortest s-t path in $G_{f'}$.

$$\begin{aligned}\delta_{f'}(s,v_i) &= \delta_{f'}(s, v_{i+1})+1\\ &\ge \delta_f(s, v_{i+1})+1\\ &= \delta_f(s, v_i)+2\end{aligned}$$

[missing notes 47:22 - 50:20]

Each time edge (u,v) is removed and comes back, shortest path distance from s to u $\uparrow$ by 2.

Fact Any shortest path distance $n-1$

\# of edge re-emergences is at most ≤ 2 (for some edge)

\# times one edge can be removed and come back $= O(n)$
\# edges $= m$

paths = $O(mn)$

Runtime of Edmonds-Karp-Dinic: $O(m^2 n)$ ← Cost of BFS in each iteration is $O(m)$.

Selecting the K^{th} Smallest Element

Selecting Medians and Order Statistics

★ For simplicity we'll assume all n elements are distinct (no big ideas are needed for the general case)

- Fundamental Problem:
  Select the K^{th} smallest element in an unsorted sequence
- Definition: An element x is the K^{th} order statistic of a sequence S if x is the K^{th} smallest element of S.
- Selection Problem:
  - Given an array S of n elements and $K \in \{1, 2, \dots, n\}$.
  - Return the K^{th} order statistic of S
- Example: If n is odd and $K = (n+1)/2$, we get the median

A Naïve Solution

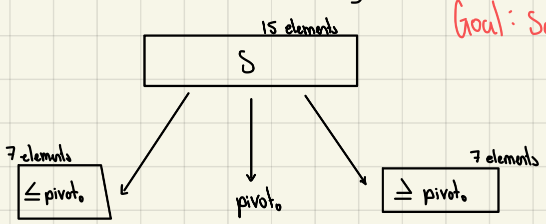
A sorting-based approach:

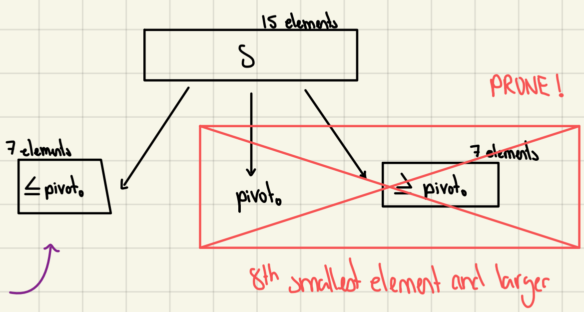
1. Sort S in increasing order
2. Output the K^{th} element of the sorted sequence

How long does this take? Is this the best possible?
$O(n \log n)$ No! → $O(n)$ is possible! → Time Bounds for Selection* — Stanford University (1972)

Quickselect: Quicksort with Pruning

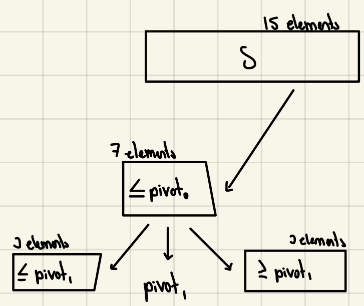
Goal: Select the 6^{th} smallest element

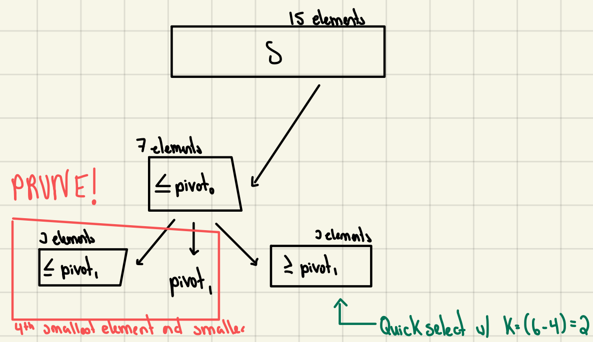




Quickselect w/ $K=6$

$$?\left(\sum_{j=0} (n)(1/2)^j = 2n\right)?$$





Oct. 28th 2021 (Lecture 12)

Selecting the k^{th} smallest element

Quickselect

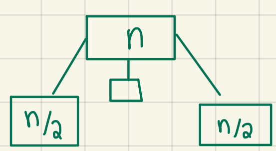
```
Quickselect(S,k):
  If S.length() == 1
     Return S[0]
  p = PickPivot(S)     // how to pick pivot? To be explained later!
  [L, G] = Partition(S, p)
  If k <= length(L)
     Return Quickselect(L, k)
  Else If  k == (length(L) + 1)
     Return p
  Else  // k > (length(L) + 1)
     Return Quickselect(G, k - length(L) - 1)
```

Quickselect — Optimistic Analysis

- Suppose we always take the pivot to be the first element in the sequence and are so lucky that it is always the median.
- Then PickPivot(S) just returns S[0] and so costs 1.
- Quickselect on sequence of length n either:
  (a) calls Quickselect on sequence on length at most $\lceil n/2 \rceil$
  or
  (b) returns the k^{th} order statistic itself

ex.

4	1	2	3	5	6	7



So: $T(n) \leq T(n/2) + cn$

↑ positive constant

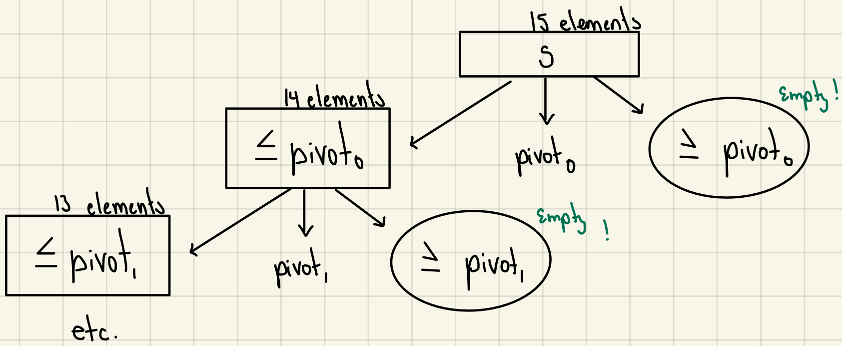
$$
\begin{aligned}
T(n) &\leq T(n/2) + cn \\
&\leq T(n/4) + cn/2 + cn \\
&\leq T(n/8) + cn/4 + cn/2 + cn \\
&\vdots \\
&\leq T(1) + (cn)\sum_{j=0}^{\infty} 2^{-j} \\
&= T(1) + 2cn \\
&= O(n)
\end{aligned}
$$

[missing notes 16:22 – 24:14]

This is great! But we cheated by assuming that taking the pivot as the first element always gives us the median.

QuickSelect — Worst-Case Analysis

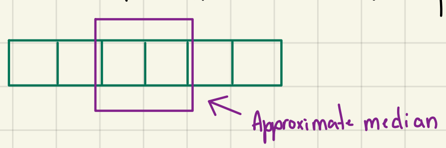
Goal: Select th 6^{th} smallest element!



$$
\begin{aligned}
T(n) &= T(n-1) + cn \\
&= T(n-2) + c(n-1) + cn \\
&= T(n-3) + c(n-2) + c(n-1) + cn \\
&\vdots \\
&= T(1) + c\sum_{j=2}^{n} j \\
&= O(1) + c\left(\frac{n(n+1)}{2} - 1\right) \\
&= \Omega(n^2)
\end{aligned}
$$

Picking a Good Pivot

- Median pivots are th best possible choice
- But if we knew how to get the median, we would be done!
- Idea: Try to find an "approximate median" using less work
  - Find th median of a well-chosen subset of th sequence



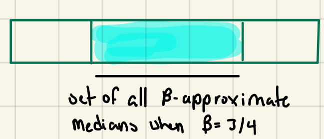
Definition

Let β satisfy $1/2 \leq \beta \leq 1$. We say that an element m of sequence S is a β-approximate median of S if:

At most β_n elements of S are less than m

and

At most β_n elements of S are greater than m



Is a β-Approximate median a Good Pivot?

- If the pivot is a β-approximate median, then calling Quickselect on a sequence of n points leads to both L and G that each are of size at most βn.
- If Quickselect always uses a β-approximate median, then at level j of Quickselect (i.e. inside the j^{th} recursive call), both L and G each can have size at most $\beta^j n$.

$$
\begin{aligned}
T(n) &\le T(\beta n) + cn \\
&\le T(\beta^2 n) + c\beta n + cn \\
&\le T(\beta^3 n) + c\beta^2 n + c\beta n + cn \\
&\vdots \\
&\le T(1) + cn \sum_{j=0}^{k'} \beta^j \quad \Big] = O(n) + \frac{cn}{1-\beta}
\end{aligned}
$$

[missing notes 37:00 – 38:00]

Quickselect with β-approximate median

So runtime of Quickselect using a β-approximate median is...

$$T(n) = O(1) + \frac{cn}{1-\beta} = O(n)$$

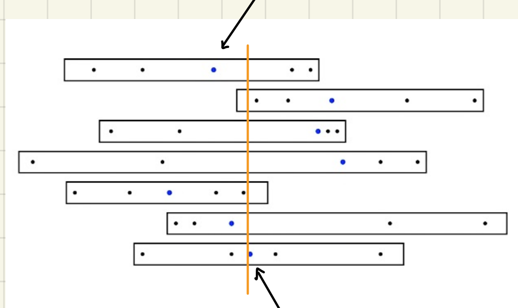
This is great, but we are still cheating... We need a way to find β-approximate median AND must account for the computational cost of doing so!

Computing a β-Approximate Median

- Partition sequence into $n/5$ segments, each of size 5.
  - For simplicity, we ignore the fact that the last segment might have size less than 5.
- Find the median of each segment (Sort to get median: $O(1)$) → Total cost: $O(n)$)
- Find the median of the $n/5$ medians (somehow).

ex.

Finds median of each segment



median of medians! (m^*)

- Two Outstanding problems:

(1) Is median of medians a good pivot, i.e. is it a β-approximate median?

- $n/5$ medians of which $n/10$ of are less than or equal to m^*

(2) How do you efficiently compute median of medians?

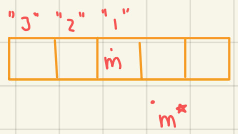
Quickselect with Median-of-Medians Pivot

- Is median of medians a good pivot,
  i.e. is it a β-approximate median?

  - $n/5$ medians of which $n/10$ of are less than or equal to m^*
  - For each such median, 2 more points are less than m^*

  Suppose $m < m^*$

  other median



  - At least $(3n/10) - 1$ elements less than m^*
  - Hence, at most $7n/10$ elements are greater than m^*
  - By symmetry, at most $7n/10$ elements are less than m^*

  $\Rightarrow m^*$ is a β-approximate median ($\text{for } \beta = 7/10$)

- How to compute median of medians?
- Ideal Recursively call Quickselect(medians, $(n/5)/2$)
- Can this really work?
  - Original sequence was of length n
  - Sequence of medians is of length only $n/5$
  - Smells like divide-and-conquer

Run-Time Analysis

median of medians!

$$T(n) = T(n/5) + T(7n/10) + cn$$

$\uparrow T(\beta_n)$

$$T(n) \le O(1) + cn \sum_{j=0}^{\infty} (\alpha+\beta)^j$$

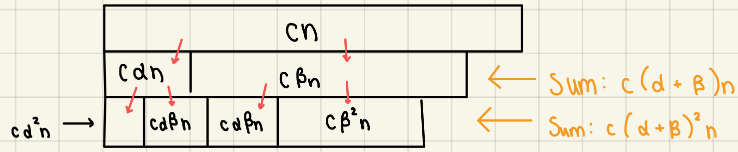
$$= \frac{cn}{1-(\alpha+\beta)}$$

$$= 10cn$$

$$= O(n)$$

Note: We could use substitution method to analyze complexity. Instead, let's use "stack of bricks" view of the recursion tree.

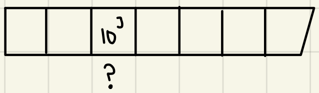
Set $\alpha = 1/5$ and $\beta = 7/10$



Nov. 1st 2021 (Lecture 13)

Last Lecture Wrap-Up

Ex. Bob: $o(n)$ selection algorithm



Find max : $k = n$

Probability

The sample space, or outcome space, is the set of all possible outcomes.
We denote it as Ω.

Suppose we flip a coin once. Then the sample space is: $\Omega = \{H, T\}$

If instead we flip a coin twice. Then the sample space is: $\Omega = \{H, T\}^2 = \{HH, HT, TH, TT\}$

Probability Distribution

First two of Kolmogorov's probability axioms:

1) For any outcome $a \in \Omega$, $P(a) \geq 0$

2) $P(\Omega) = 1$ (with probability 1, some outcome must happen)

Coin-Flipping Example

Suppose we flip a coin once, so $\Omega = \{H, T\}$.
Probability distribution of outcome is specified by the Bernoulli Distribution. (Bernoulli (p))

Let $P(H) = p$. We call p the success probability. A fair coin corresponds to $p = 1/2$.

$\underbrace{P(H)}_{p} + \underbrace{P(T)}_{1-p} = 1$

Dice Example

Suppose we roll a pair of dice; then $\Omega = \{1, 2, \ldots, 6\}^2$.
Probability distribution for the outcome (a pair of numbers) is the Uniform Distribution.

The uniform distribution satisfies $P(a) = P(b)$ for all $a, b \in \Omega$

Therefore, we have...

$$P(a) = \frac{1}{|\Omega|} \quad \text{for all } a \in \Omega$$

In th dice example, $P(i,j) = 1/36$ for any $i,j \in \{1,2,\dots,6\}$

$P(1) = P(2) = \dots = P(6) = 1/6$ | $\Omega = \{1,2,\dots,6\}$

Formally: For $v \in V$, we can define th event $X = v$.

$$P(X=v) = P(\{a \in \Omega : X(a) = v\})$$

or $U \subseteq V$, $X \in U$

$$P(X \in U) = P(\{a \in \Omega : X(a) \in U\})$$

Events

An **event** $A \subseteq \Omega$ is a subset of th sample space.

Suppose we flip a coin twice. Then $\{HT, TH\}$ is an event.

The probability of an event A is $P(A) = \sum_{a \in A} P(A)$

Suppose we roll one die. What is th probability of rolling an even number? We can use shorthand:

$$P(a \text{ is even}) = P(\{a \in \Omega : a \text{ is even}\}) = P(\{2,4,6\})$$
$$= 1/6 + 1/6 + 1/6 = 1/2$$

Random Variables

A **random variable** X is a function from th sample space to $V \subseteq \mathbb{R}$.

$$X : \Omega \xrightarrow[\text{to}]{} V$$

Example 1: Suppose we roll a pair of dice and then win an amount of dollars equal to th sum of the rolls.

If the outcome is (a,b), then the amount we win is given by th random variable $X = a+b$.

Example: $V = [-100, 100]$
$X(H) = 100$
$X(T) = -90$

$(0.5)(100) + (0.5)(-90)$
$= 1/2(100-90)$
$= 5$

Example 2: Suppose that K horses are racing, and we bet money on horse j. If horse j wins th race, we win \$100; Otherwise we win \$0.

Formally, we have sample space $\Omega = \{1,2,\dots,k\}$, where th outcome is i if horse i wins.

The amount we win is given by th random variable: $X = 100 \cdot 1[\text{horse } j \text{ wins}] = 100 \cdot 1[a=j]$

Expected Value

For a random variable X, we defined the expected value as...

$$E[X] = \sum_{v \in V} v P(X=v)$$

$$= \sum_{a \in \Omega} X(a) P(a)$$

Linearity of Expectation

For random variable X and Y and constants a, b, c, we have:

- $E[aX] = aE[X]$
- $E[X+Y] = E[X] + E[Y]$ ex. $E[aX+bY+c] = aE[X] + bE[Y] + c$

Independence

Two events A and B are independant if $P(A \cap B) = P(A) \cdot P(B)$

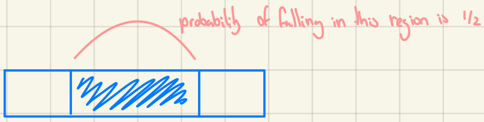
Two random variables X and Y are independant if, for any $u, v \in V$, the events $[X=u]$ and $[Y=v]$ are independant.

Randomized Quickselect and Randomized Quicksort

→ Recall Quickselect's "Recursion Path".
→ Recall Upper bound on runtime of Quickselect w/ median of medians pivot.
Theorem: Quickselect (S, k) using the median of medians pivot returns the k^{th} order statistic in time at most $O(n)$.

Randomized Quickselect

```
Quickselect(S,k):
    If S.length() == 1
        Return S[0]
    p = RandomPivot(S)   // p will be a random element from S
    [L,G] = Partition(S,p)
    If k <= length(L)
        Return Quickselect(L,k)
    Else If k == (length(L) + 1)
        Return p
    Else   // k > (length(L) + 1)
        Return Quickselect(G, k - length(L) - 1)
```



If the random pivot falls within blue middle region, the size of the next node in the recursion path will be at most 3/4 the size of the current node.

Using the language from last lecture, such a pivot is a β-approximate median for $\beta = 3/4$

Sketch of Bound on Expected Runtime

X_i = # of elements in node a

$\mathbb{E}[\text{Work}]\ \mathbb{E}\left[\sum_{i=1}^{n} cX_i\right] = c\sum_{i=1}^{n}\mathbb{E}[X_i]$

- Let's view the extension of the recursion path in rounds
- In each round, we draw a new pivot and consequently add one node in our recursion path (highlighted in blue)
- Because the chance of a random pivot falling in the region of good pivots is $1/2$, in roughly half the rounds we expect to decrease the node size to $3/4$ of its previous size
- After k rounds of good pivots, the node size is only $n(3/4)^k$
- After $\log_{4/3} n$ rounds of good pivots, the node size is at most 1 and so the algorithm has returned
- The expected runtime should therefore be at most double the runtime of an algorithm that always gets good pivots
  - Quickselect w/ the median pivot was precisely such an algorithm

$\Rightarrow$ The expected runtime of Randomized Quickselect should be $O(n)$

ex. $P(H) = q$, $P(T) = 1-q$

$P(H) + P(TH) + P(THH) + \ldots$

$= q + (1-q)q + (1-q)^2 q + \ldots$

$= q\left[(1-q)^0 + (1-q)^1 + (1-q)^2 + \ldots\right]$

$= q\left(\dfrac{1}{1-(1-q)}\right)$

$= q/q$

$= 1$

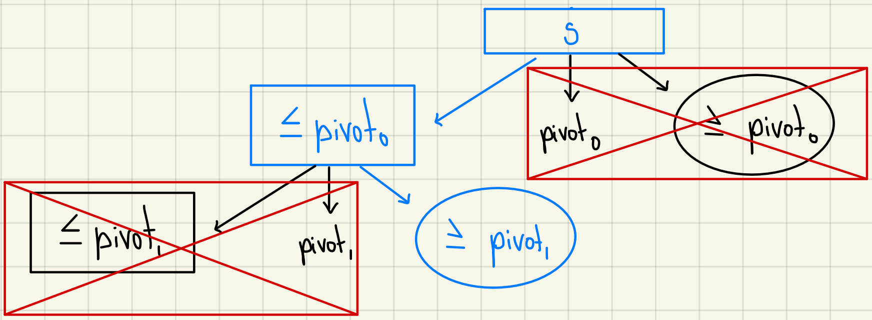
ex. [missing notes 1:13:00 – 1:16:00]

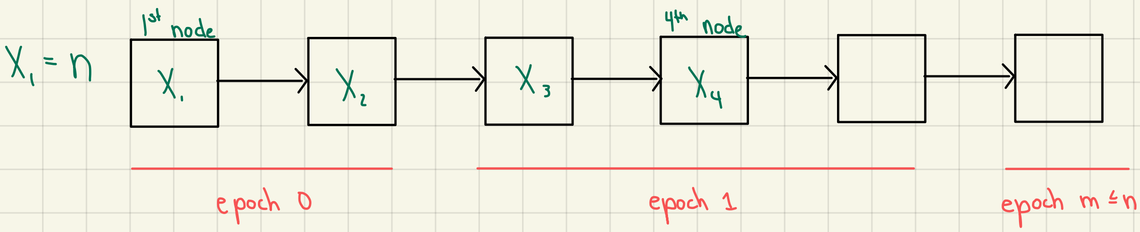
Note: expected runtime is not a worst case runtime!

Nov. 4th 2021 (Lecture 14)

Sketch of Bound on Expected Runtime

Recall





Step 1

X_i = # of elements in node i

runtime $\leq \sum_{i=1}^{n} c \cdot X_i$ ← random variable!

[missing notes : 10:40 – 16:59]

Step 2

node i belongs to epoch 0 if $(3/4)n < X_i \leq n$

node i belongs to epoch 1 if $(3/4)^{2} n < X_i \leq (3/4)n$

General case: if node i in epoch $(3/4)^{j+1} n < X_i = (3/4)^{j} n$

[missing notes : 18:00 – 27:00]

$\mathbb{E}\left[\text{\# times — first good pivot}\right]$

$$= \sum_{k=1}^{n} k\, P(\text{\# times} = k)$$
$$= \sum_{k=1}^{n} k\,(1-p)\,p^{k-1}$$
$$= (1-p) \sum_{k=0}^{n} k\, p^{k-1}$$
$$= (1-p) \sum_{k=0}^{n} \frac{d}{dp} p^{k}$$
$$= (1-p) \frac{d}{dp} \sum_{k=0}^{n} p^{k}$$

$$\leq (1-p) \frac{d}{dp} \frac{1}{1-p} = \frac{1-p}{(1-p)^2}$$

[missing notes : 34:00 — 39:00]

Random Quicksort

```
Quicksort (S, p, r)
    If p < r
        q = Partition (S, p, r)      ← majority of the work!
        Quicksort (S, p, q-1)
        Quicksort (S, q+1, r)
```

~~Randomized~~ Quicksort – Last Element as Pivot

```
Partition (S, p, r)
    x = S[r]
    i = p-1
    For j = p to r-1
        If S[j] <= x
            i = i+1
            Swap (S[i], S[j])
    Swap (S[i+1], S[r])
    Return i+1
```

Loop invariant

For any index k:
- If $p \le k \le i$, then $S[k] \le x$
- If $i+1 \le k \le j-1$, then $S[k] > x$
- If $k = r$, then $A[k] = x$

[missing notes : 45:00 — 50:00]

Randomized Quicksort - Random Pivot

```
Partition (S, p, r)
    Swap (S[Random (p, r)], S[r])
    x = S[r]    // the last element is now random
    i = p-1
    For j = p to r-1
        If S[j] <= x
            i = i+1
            Swap (S[i], S[j])
    Swap (S[i+1], S[r])
    Return i+1
```

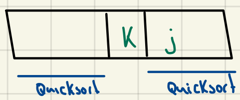
Loop invariant

For any index k:
- If $p \le k \le i$, then $S[k] \le x$
- If $i+1 \le k \le j-1$, then $S[k] > x$
- If $k = r$, then $A[k] = x$

Analysis

$\boxed{z_1 < z_2 < \ldots < z_n}$

Observation. Any 2 elements (j and k) can be only compared once



Let X_j be a random variable that = 1 if element j is compared to element k. ______ ?

$\mathbb{E}\left[\text{\# comparisons}\right]$ ______ can't read board!

[missing notes : 56:00 — 1:00:00]

[missing notes : 1:00.00 — 1:03:00]

Ex.

2	5	3	4	7	6	10

3	2	4

7	6	10

[missing notes : 1:06:00 — 1:10:00]

$P_r(Z_j \text{ is compared to } Z_K)$

$\leq P_r\left(\begin{matrix}\text{in some call to partition either} \\ z_j \text{ is pivot or } z_K \text{ is pivot}\end{matrix}\right)$

$\leq \dfrac{2}{k-j+1}$

[missing notes : 1:13:00 — 1:20:58]

Nov. 8th 2021 (Lecture 15)

Warm-Up Puzzle

We have a deck of n distinct cards (where n is large) and repeatedly sample a card uniformely at random, with replacement. On average, how many cards do we need to draw before we see some card twice (that is, before we have repeated a card)?

→ $n = 1\,000\,000 = 2^{20}$ $\qquad 1/10^6 \cdot 2/10^6 \cdot \ldots \cdot 20/10^6 \leq\ ?$

$\log n = 20$

Answer: If seen $\sqrt{n}$ distinct cards then chance next card is repeat ...

$$= \sqrt{n}/n = 1/\sqrt{n}$$

$$\underbrace{1/\sqrt{n} + \frac{\sqrt{n}+1}{n} + \ldots +}_{\sqrt{n}\text{ times}} \geq 1/\sqrt{n}\ \sqrt{n} = 1$$

$\therefore \Theta(\sqrt{n})$ (Note: Section 5.4.1 in CLRS (Birthday Paradox))

Dictionary

A dictionary is a data structure that contains key-value pairs.

→ Keys should be unique

→ Values can be anything and need not be unique

$\boxed{\text{key , data}}$

↑ ↑ "value"

Dictionary – Operations

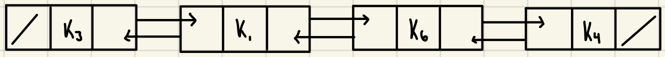
△ SEARCH – Need to specify key k

△ INSERT – Need to specify object x (obtain key via x.key)

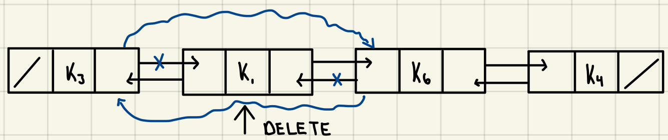
△ DELETE – Need to specify object x

(Note: we will see later why it is better to take as input x rather than x.key)

Unordered List

Double Linked List: Head → 

Operation	Worst-Case Running Time? (n Elements)
SEARCH(S,k)	O(n)
INSERT(S,x)	O(1)
DELETE(S,x)	O(1)

Head → 

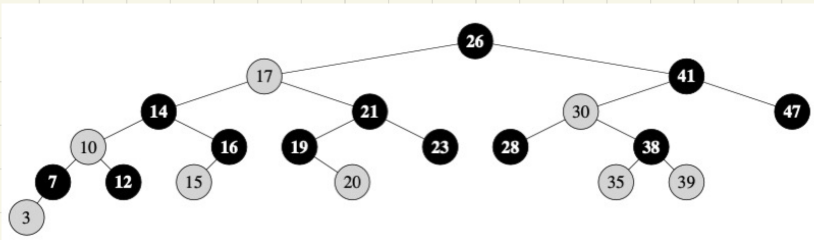
Ordered List

K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7
0	1	2	3	4	5	6	7

where $K_0 < K_1 < K_2 < \dots < K_7$

SEARCH(S, K) — Binary Search Enables $O(\log n)$
INSERT(S, x) — $O(n)$
DELETE(S, x) — $O(n)$

Balanced Binary Search Tree (red-black tree, AVL tree)



SEARCH(S, K) — $O(\log n)$
INSERT(S, x) — $O(\log n)$
DELETE(S, x) — $O(\log n)$

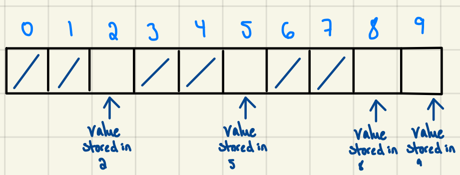
Direct-Address Table

Suppose the Keys are in a universe $U = \{0, 1, \dots, m-1\}$. In a direct-address table, we create an array T of size m (initialize all entries to NULL). Element with Key K is stored in $T[K]$

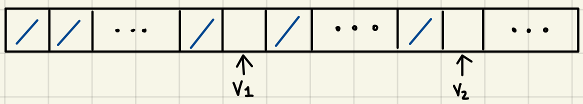
SEARCH(S, K) : return $T[K]$ — $O(1)$
INSERT(S, x) : $T[x.Key] = x$ — $O(1)$
DELETE(S, x) : $T[x.Key] = \text{NULL}$ — $O(1)$

Space complexity? : $O(m)$

ex. Universe $\{0, 1, \dots, 9\}$: $n=4$ w/ 4 Keys : 2, 5, 8, 9



ex. Universe $\{0, 1, \dots, 2^{n-1}\}$ where we only have n Keys



What fraction of space is being utilized?
Storing n Keys in 2^n space
Utilization: $n/2^n \approx 0$

Hash Tables

A data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index (hash code), into an array of buckets or slots, from which the desired value can be found.

Assume keys $\in U$ and $|U|$ is very large, but # slots in table $= m \ll |U|$.

[missing notes: 40:00 – 43:30]

Hash Function and Collisions

Let $h: U \longrightarrow \{0, 1, \ldots, m-1\}$ be hash function.

Given key k, we call $h(k)$ the hash value of key k. (not-so-smart example: $h(k) = k \bmod 10$)

If two keys hash to the same slot, then we have a collision.

If the key is a floating point number, rescale. If key is a string, interpret as number.

- Suppose any key $\in [0, 3.1428)$

  $g(k) = \lfloor (k/3.1428) 10000 \rfloor$

  $\tilde{h}\ \{0, 1, 2, \ldots, 9999\} \longrightarrow \{0, 1, 2, \ldots, m-1\}$

  $h(k) = \tilde{h}(g(k))$

[missing notes: 52:30 – 56:00]

Handling Collisions

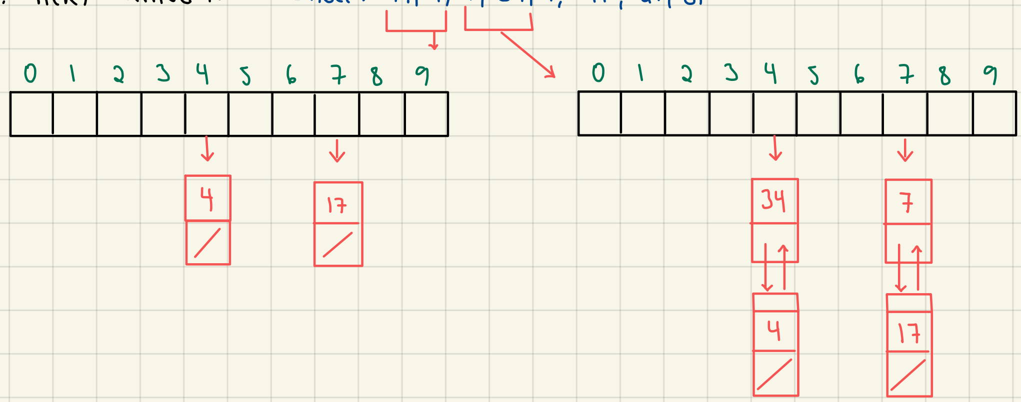
1) Design a hash function which makes collisions as unlikely as possible.

2) Chaining – Let each hash table slot store a linked list.
3) Open Addressing – If the desired entry is already full, then try some other slots (using some fixed order).

Chaining

In chaining, we store all elements that hash to the same slot j within a linked list $T[j]$.

ex. $h(k) = k \bmod 10$ Insert: 17, 4, 7, 34, 1, 41, 21, 31



SEARCH(S,k) : Search list at $T[h(k)]$ $O(\text{length of } T[h(k)]) = O(n)$

INSERT(S,x) : Insert x at the head of list $T[h(x.key)]$ $O(1)$

DELETE(S,x) : Delete x from list $T[h(x.key)]$ $O(1)$

Load Factor

Let T be a hash table of size m that stores n elements.

The load factor α of T is the average length of a chain. This is simply the ratio of number of elements stored to number of slots. Therefore, $\alpha = n/m$.

If we have a good hash function, the load is balanced (most chains have length α). In this case, the cost of each SEARCH operation is close to α.

It can be challenging to find a good hash function which deterministically keeps most chains at length α. Instead, we will consider situations where a hash function is randomly selected such that, on average, any chain $T[j]$ has length α.

[missing notes: 1:08:00 – 1:13:00]

Simple Uniform Hashing

For any key k, its hash value $h(k)$ is drawn uniformly at random from $\{0, 1, \ldots, m-1\}$.

Let n_j be the length of the chain $T[j]$.

Suppose we insert n elements and the simple uniform hashing assumption holds. For any j in $\{0, 1, \ldots, m-1\}$, what is $\mathbb{E}[n_j]$?

$\mathbb{E}[n_j] = ?$

Let random variable $z_{ij} = 1[h(k_i) = j]$

$n_j = \sum_{i=1}^{n} z_{ij}$ | $\mathbb{E}\left[\sum_{i=1}^{n} z_{ij}\right] = \sum_{i=1}^{n} \mathbb{E}[z_{ij}] = \sum_{i=1}^{n} \underbrace{P_r(z_{ij} = 1)}_{m} = n/m = \alpha$

November 15th 2021 (Lecture 16)

Simple Uniform Hashing

Slot j — n_1 be length of chain at slot j.

i^{th} inserted Key!

Let $Z_{ij} = 1[\text{Key } k_i \text{ hashes to slot } j]$

load factor

$$n_j = \sum_{i=1}^{n} Z_{ij} \quad \Big| \quad \mathbb{E}[n_j] = \sum_{i=1}^{n} \mathbb{E}[Z_{ij}] = \sum_{i=1}^{n} \underbrace{\Pr(h(k_i)=j)}_{=1/m} = n/m = \alpha$$

Expected Time for Unsuccessful Search (for a key k)

Proposition: The average-case cost of an unsuccessful search is $1+\alpha$

(Cost Model: Hash costs one, examining an element costs one)

Expected Cost (Runtime) : $(\text{I}) + (\text{II}) \longrightarrow 1+\alpha$

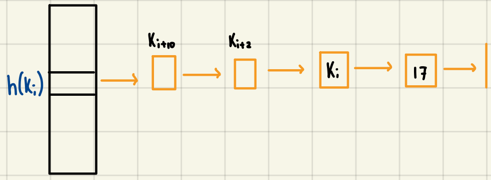
(I) cost to compute $h(k)$

(II) cost to traverse $T[r]$

Expected Time for Successful Search (for i^{th} inserted key)

Proposition: The average-case cost when searching for the i^{th} inserted Key (after all n Keys have been inserted) is:

$$2 + (n-i)/m \leq 2 + \alpha$$



$$\mathbb{E}[\cancel{Cost}] = \mathbb{E}\Big[\sum_{j=i+1}^{n} X_{ij}\Big] + 1 + 1 \quad \Big| \quad X_{ij} = 1[h(k_j) = \overbrace{h(k_i)}^{=r}]$$

$$= 1[h(k_j) = r]$$

$$= (n-i)/m + 2$$

$\cancel{n_{h(k_i)}}$

Corollary: The average-case cost when searching for an inserted Key (also chosen uniformly at random from the set of n inserted Keys) is:

$$2 + (n-1)/(2m) \leq 2 + \alpha/2$$

Suppose I is drawn from uniform distribution over $\{1, 2, \ldots, n\}$

$$\mathbb{E}[\text{ Cost for Searching for } k_I]$$

$$= \frac{1}{n} \sum_{i=1}^{n} \left((n-i)/m + 2 \right)$$

$$= 2 + \frac{1}{n} \sum_{i=1}^{n} (n-i)/m$$

$$= 2 + \frac{1}{nm} \sum_{j=0}^{n-1} j$$

[missing notes : 31:00 — 32:30]

How Can We Design A Good Hash Function?

Suppose floated point key $K \sim U([0,1))$ (uniform)

$$h(k) = \lfloor Km \rfloor \sim U(\{0, 1, 2, \ldots, m-1\})$$

Division Method

In the division method, we simply divide by m and take the remainder :

$$\boxed{h(k) = K \bmod m}$$

- $m = 10 \longrightarrow h(1017) = h(10^3 + 7) = h(27)$ BAD CHOICE !
- $m = 2^r$ (small positive integer)

  ↓

  Suppose $r = 3 \Rightarrow h(\underbrace{100101100}_{\text{Binary}}) = \underbrace{100}_{\text{Binary}}$ BAD CHOICE !

Multiplication Method

1) Select a constant A such that $0 < A < 1$
2) Take fractional part of KA
3) Multiply by m and truncate

$$h(k) = \lfloor m (KA \bmod 1) \rfloor$$

How to choose A?

→ $A = 1/\varphi = 0.6180339887 5\ldots$ tends to work well!

(distributes nearby integers roughly uniformly in [0,1])

Universal Hashing

The previous methods might work well in practice, but we do not have rigorous guarantees for them...

→ A universal hash family is a collection $\mathcal{H}$ of hash functions $h: U \to \{0,1,\dots,m-1\}$ such that, for any pair of keys j, k, at most $|\mathcal{H}|/m$ hash functions $h \in \mathcal{H}$ satisfy $h(j) = h(k)$.

(for any keys $k \neq j$ # of elements of $\mathcal{H}$ such that $h(j) = h(k) = |\mathcal{H}|/m$)

"for any keys $j \neq k$ at most $1/m$ fraction of our hash functions lead to a collision."

How can we use this?

If we select h uniformly at random from $\mathcal{H}$, then for each pair of keys j, k, we have:

$$Pr(h(j) = h(k)) \leq 1/m$$

Average-Case Analysis for Universal Hashing

Proposition: Let h be drawn uniformly at random from a universal family of hash functions. Consider an arbitrary key k.

If key k is not in the table, then the expected length of the list $T[k]$ is at most α.
Otherwise, the expected length of the list is at most $1+\alpha$.

Search for key k

(i) Suppose key $k \notin T$: $\mathbb{E}[\text{length of } T[h(k)]]$

$$= \mathbb{E}\left[\sum_{\ell \in T} X_{\ell k}\right]$$

$$\leq n \cdot 1/m = n/m = \alpha$$

Let $X_{jk} = 1[h(j) = h(k)]$

$$\mathbb{E}[X_{jk}] \leq 1/m$$

(ii) Suppose $k \in T$: $\mathbb{E}[\text{length of } T[h(k)]]$

$$= 1 + \mathbb{E}\left[\sum_{\ell \in T, \ell \neq k} X_{\ell k}\right]$$

$$\leq 1 + (n-1)/m$$

$$\leq n/m + 1 = 1 + \alpha$$

Bonus: Constructing a universal family of hash functions (lecture14.pdf — Slide 29)

Open Addressing

Open addressing is another method for handling collisions. Unlike chaining, each slot stores at most one Key.

If we try to store a Key in a slot but find that it is already occupied, we instead try some other slot, and if that slot is full, we try yet another slot, and so on...

This sequence of slots that we try when we are probing for an unoccupied slot is called a PROBE SEQUENCE.

A first probe sequence:

$$\underbrace{h(k), h(k)+1, h(k)+2, \ldots, h(k)+(m-1)}_{\text{all mod } m}$$

Linear probing uses this probe sequence.

[missing notes: 1:14:00 — 1:15:20]

[missing notes: 1:16:00 — 1:20:52]

November 18th 2021 (Lecture 17)

Linear Probing

Using the hash function $h(k) = k \bmod 10$.

Search: Use probe sequence and stop when we have either found the key or arrived at an unoccupied slot.

Delete: It can cause trouble for Search.
↳ Upon deletion, mark slot w/ special value DELETED.

→ Insert: 35, 21, 16, 45, 31, 8
$h(k) =$ 5, 1, 6, 5, 1, 8

0	1	2	3	4	5	6	7	8	9
	21	31			35	16	45	8	

Note: It doesn't work well in pratice ◠̈ → primary clustering problem.
Limited number of probe sequence (only m of them).

Quadratic Probing

In quadratic probing, we use a somewhat more sophisticated probe sequence. For carefully selected positive constants c_1 and c_2, the probe sequence is..

$$(h(k) + c_1 i + c_2 i^2) \bmod m \quad : \text{for } i = 0, 1, \ldots, m-1.$$

Advantages: Avoids primary clustering problem. Δ̈

Disadvantages: Experiences secondary clustering problem. Still only m probe sequences.

- 2 keys k, k' such that $h(k) = h(k')$ will have some probe sequence ◠̈

Double Hashing

Let h_1 and h_2 be auxiliary hash functions.

$i = 0 : h(k,i) = h_1(k)$
$i = 1 : h(k,i) = h_1(k) + h_2(k)$
$i = 2 : h(k,i) = h_1(k) + 2h_2(k)$

Double hashing uses the probe sequence:

$$h(k,i) = h(h_1(k) + i \cdot h_2(k)) \bmod m$$

↑ General way of specifying elements in probe sequence.

$\overbrace{h_2(k) \text{ must be relatively prime to } m}^{\text{no common factors}}$ in order for the whole table to be searched. (How can this be achieved?)

If m = power of 2 and $h_2(k)$ is odd keys k.

Average-Case Analysis of Open Addressing

Δ Can we provide average-case guarantees for open addressing? YES! Under a certain assumption.

Assumption of uniform hashing – for each key, the probe sequence $h(k,i)$ is chosen uniformly at random from the set of all possible permutations of $(0,1,\ldots,m-1)$. (This is not realistic, but we might approximate it in practice using, e.g., double hashing).

Proposition: Given a hash table with load factor $\alpha = n/m < 1$, under uniform hashing the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.

Ex. $1/2 \leq 2/3 \leq 4/5 \leq \ldots$

If $\alpha = 1/2$, $1/(1-\frac{1}{2}) = 2$.

Proof: Suppose probe sequence is drawn from uniform distribution over set of all permutations of $(0,1,2,\ldots,m-1)$

- Cost for sure ≥ 1
- Expected cost :

prob. that 1st slot is occupied!

prob that 2nd slot is occupied!

$$1 + (n/m)\left(1 + \frac{n-1}{m-1}\left(1 + \frac{n-2}{m-2}\left(1 + \frac{n-3}{m-3}\left(1 + \ldots\right)\right)\right)\right)$$
$$\leq 1 + (n/m)(1 + n/m(1 + n/m(1 + n/m(1 + \ldots)$$
$$= 1 + \alpha(1 + \alpha(1 + \alpha(1 + \ldots)$$
$$= 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$$
$$= 1/1-\alpha$$

Note: Theorem 11.8 of CLRS!

Amortized Analysis

Amortized analysis is a way of doing worst-case analysis by bounding the average cost to perform each operation (averaged over the sequence of operations).

Motivation: Some operations might be very expensive, but they happen infrequently, so on average each operation might have low cost.

Amortized Analysis is NOT RELATED to Average-Case Analysis; amortized analysis doesn't use probability or expected value.

The Peril of Per-Operation Worst-Case Analysis

Ex. Stack S

Suppose that in addition to having the usual PUSH $O(1)$ and POP $O(1)$ operation, we also have an operation KPOP $O(k)$.

KPOP(S,k): Pop top k elements on stack (or all elements if less than k elements are on stack).

Worst-case cost of KPOP operation? $\longrightarrow O(k)$

Worst-case cost of sequence of n PUSH, POP, and KPOP operations? $\longrightarrow O(nk)$

Aggregate Analysis

Aggregate analysis bounds worst-case runtime in aggregate (over whole sequence of operations), rather than giving worst-case bounds for each operation separately (without consideration of previous operations).

Ex. Stack with KPoP

- Cost of each KPoP is simply th number of actual pops that happen within it.
- Total number of pops is at most total number of pushes (at most n).
- So, any sequence of n pushes, pops, and KPoPs takes at most $O(n)$ time.

Accounting Method

In th accounting method, for each operation we charge an amortized cost.

- The amoritized cost of an operation can be greater than (or less than!) th actual cost.

For th i^{th} operation:

c_i is actual cost, $\hat{c}_i$ is amortized cost. $\longrightarrow$ Interpretation of $\hat{c}_i > c_i$?

We change th actual cost c_i plus credit $(\hat{c}_i - c_i)$.

Goal: Select amoritized costs such that we have:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$

actual cost — upper bound on runtime!

Accounting Method — Stack with KPoP Example

If i^{th} operation is PUSH: $\hat{c}_i = 2$ (pay 1 for actual cost $c_i = 1$ and prepay 1 because eventually th pushed element might be popped).

If i^{th} operation is POP : $\hat{c}_i = 0$ (already paid for by some PUSH! $[c_i = 1]$).

If i^{th} operation is KPoP : $\hat{c}_i = 0$ (all pops are already paid for).

$$\rightarrow \sum_{i=1}^{n} \hat{c}_i \leq \sum_{i=1}^{n} 2 = 2n$$

Even though some actual costs c_i are large, all amoritized costs $\hat{c}_i$ are small.

Incrementing Binary Counter Example

K-bit counter with INCREMENT

```
INCREMENT:
  i = 0
  while i < k and A[i] == 1
      A[i] = 0
      i = i + 1
  if i < k
      A[i] = 1
```

naive analysis:
worst-case operation is increment when
A = 1111 1111
$O(k) \rightarrow$ 0000 0000
$\rightarrow O(nk)$ - worst-case of n increments!

Worst-case cost of n INCREMENT operations?

Aggregate Analysis:

A[0] changes n times
A[1] changes n/2 times
A[2] changes n/4 times

$$\text{Total cost} \leq \sum_{i=0}^{k-1} n\left(\tfrac{1}{2}\right)^i$$

$$\leq n \sum_{i=0}^{\infty} \left(\tfrac{1}{2}\right)^i$$

$$= 2n$$

$k = 8$

A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	decimal
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	2
0	0	0	0	0	0	1	1	3
0	0	0	0	0	1	0	0	4
0	0	0	0	0	1	0	1	5
0	0	0	0	0	1	1	0	6
0	0	0	0	0	1	1	1	7
0	0	0	0	1	0	0	0	8

Accounting Method:

In one operation, at most a single 0 will be set to 1.

When a 0 is set to 1, charge \$2
(\$1 for actual $0 \rightarrow 1$, \$1 to prepay for $1 \rightarrow 0$)

When a 1 is set to 0, charge \$0.

$$\text{Total cost } \sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$
$$\leq \sum_{i=1}^{n} 2$$
$$= 2n$$

c_i	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	
–	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	1	-\$1
2	0	0	0	0	0	0	1	0	
1	0	0	0	0	0	0	1	1	-\$1
3	0	0	0	0	0	1	0	0	
1	0	0	0	0	0	1	0	1	-\$1
2	0	0	0	0	0	1	1	0	
1	0	0	0	0	0	1	1	1	-\$1
4	0	0	0	0	1	0	0	0	

cumulative

		$\hat{c}_i$	c_i	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	
		–	–	0	0	0	0	0	0	0	0	
1	2	2	1	0	0	0	0	0	0	0	1	-\$1
3	4	2	2	0	0	0	0	0	0	1	0	
4	6	2	1	0	0	0	0	0	0	1	1	-\$1
7	8	2	3	0	0	0	0	0	1	0	0	
8	10	2	1	0	0	0	0	0	1	0	1	-\$1
10	12	2	2	0	0	0	0	0	1	1	0	
11	14	2	1	0	0	0	0	0	1	1	1	-\$1
15	16	2	4	0	0	0	0	1	0	0	0	

November 22nd 2021 (Lecture 18)

Dynamic Tables

We want a table which can support a stream of INSERT and DELETE operations.

Like with hash tables, we define the load factor of table T to be:

$$\alpha(T) = \frac{[\text{\# of elements stored in } T]}{[\text{size of } T]} = \frac{n}{m}$$

When the load factor is 1 and a new INSERT operation arrives, we need to increase the size of the table.

We also want the load factor to be lower bounded by a positive constant (let's use 1/2 ↓ can be any fraction!) to ensure that we are using at least a constant fraction of the allocated space.

How can we insert an element x when the load factor is 1 (so the table T is full)? We need to resize the table. For simplicity let's suppose the table is of size at least 1.

T.n = # of elements stored in table T.

```
INSERT(T, x):
    if T.n == T.size          # of slots
        Allocate new table T_new of size 2·T.size     ] cost is of
        Insert all items in T into T_new              ] order T.n
        T = T_new
    Insert x into T
    T.n = T.n + 1
```

Accounting Method (credit – pre paying)

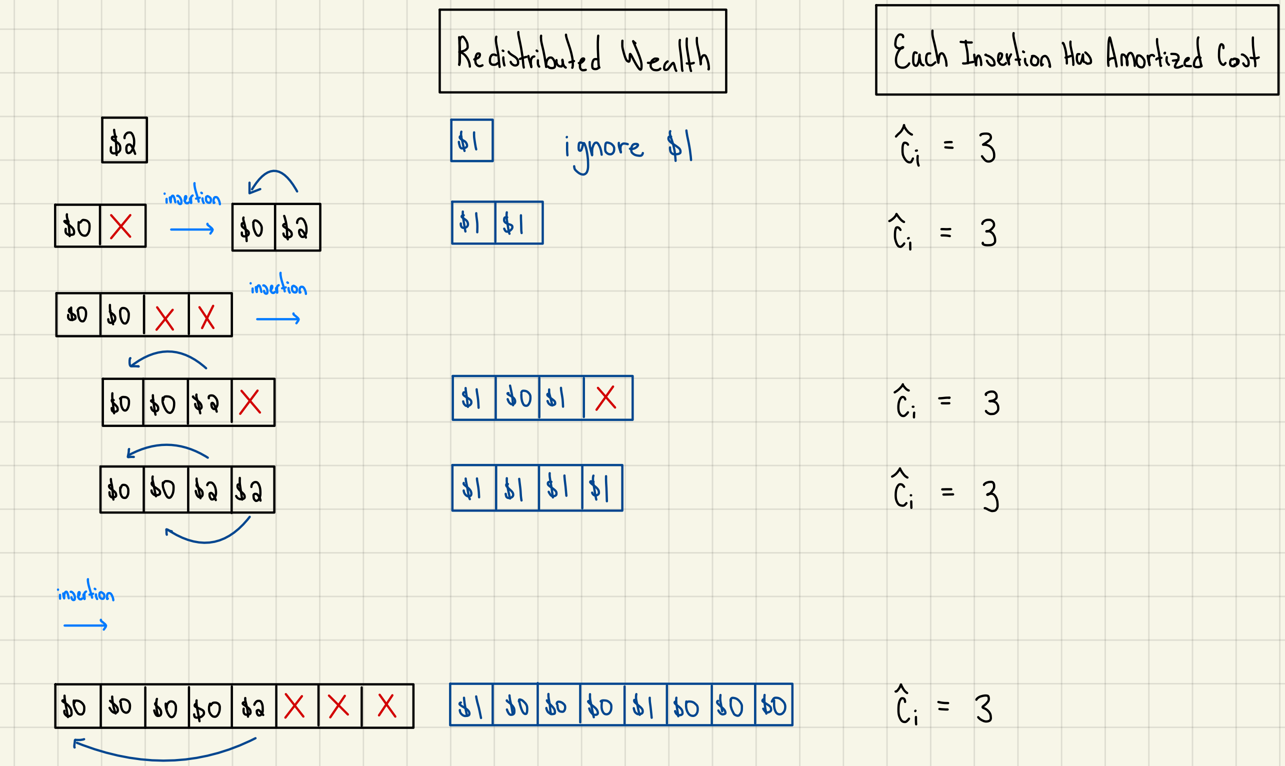
Each insertion – charge \$3, broken down as:

- \$1 for the insertion itself (this is the actual cost)
- \$2 credit for use upon resize operation:
  - \$1 credit for moving this item
  - \$1 credit for moving an item that has already been moved (such an item has no credit anymore as it used up its credit when it was moved the first time).

Example:

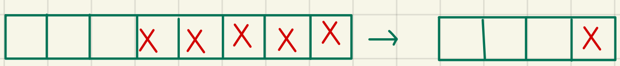
↓

Insertion



What about shrinking the table once the table is too empty? By symmetry, you might think "let's halve the table when it's less than half empty," but this is problematic.

Two ways to see why:



1) Consider what happens when the load factor is right around 1/2. A deletion triggers a halving, at which point the table is now nearly full. Two insertions trigger a doubling, and the table is right around 1/2 again. Two deletions trigger a halving, etc. This is too expensive!

2) A halving can happen soon after a doubling (since a doubling brings the load factor close to 1/2). But, after a doubling, nearly all elements have no credit on them. So, we don't have enough to pay for a halving.

Ah, why don't we postpone halving until after we have built up enough credit. Let's charge $2 for each deletion:

- $1 for the actual deletion itself (this is the actual cost)
- $1 credit to pay for moving an item upon a halving operation

When have we earned enough credit to do a halving? When the table has load $\leq 1/4$. Why?

Since the most recent doubling operation, we have deleted at least a quarter of the table. Thus, we have enough credit to move a quarter of the table upon deletion.

Deletion

Delete — generates 1 extra \$

\$0	\$0	\$0	\$0	\$2	X	X	X

↑ ignoring

$\hat{c}_i = 2$

Delete

\$1	\$0	\$0	\$0	X	X	X	X

$\hat{c}_i = 2$

Delete

\$1	\$1	\$0	X	X	X	X	X

\$0	\$0	X	X

$\alpha = 1/4$

For any sequence of n Insert / Delete operations:

$$\text{runtime} = \sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i$$

$$\le \sum_{i=1}^{n} \max\{3, 2\}$$

$$= \sum_{i=1}^{n} 3$$

$$= 3n$$

Substring Search

Goal : Find pattern of length M in text of length N. ← Typically N >> M (In some cases: M ≈ N/2)

pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A

match !

ex. Searching pdf, memory or disks, identify patterns indicative of spam, electronic surveillance.

→ SPAM: PROFITS, LOSE WEIGHT, herbal Viagra, There is no catch., This is a one-time mailing., This message is sent in compliance with spam regulations.

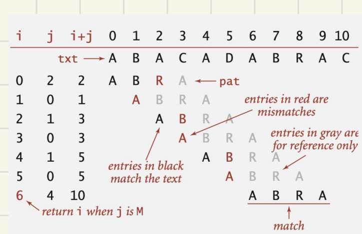
Screen scraping : Extract relevant data from web page.

ex. Find string delimited by and after first occurence of pattern "Last Trade:".

Brute - Force Substring Search

Check for pattern starting at each text position.

N = length of text ; M = length of pattern



Worst Case (cost)

i	j	i+j	0	1	2	3	4	5	6	7	8	9
		txt →	A	A	A	A	A	A	A	A	A	B
0	4	4	A	A	A	A	B	← pat				
1	4	5		A	A	A	A	B				
2	4	6			A	A	A	A	B			
3	4	7				A	A	A	A	B		
4	4	8					A	A	A	A	B	
5	5	10						A	A	A	A	B

↑ *match*

$M = 5$; $N = 10$

$O(NM) \longrightarrow$ Precise

$M \cdot (N - M + 1)$

$N - M + 1$

$\rightarrow \sim MN$ char compares.

Imagine $M \approx \sqrt{N}$ <u>or</u> $M \approx N/2$ and $N = 10^9$

November 25th 2021 (Lecture 19)

Backup

In many applications, we want to avoid backup in text stream.
- Treat input as stream of data.
- Abstract model: standard input.

Brute-force algorithm needs backup for every mismatch.

Approach 1. Maintain buffer of last M characters.

Rabin-Karp Fingerprint Search

Basic idea = modular hashing. (Division Method)
- Compute a hash of pat[0.. M-1].

$h(k) = k \bmod Q$ ⟵ large prime number!

modular hashing with R = 10 and hash(s) = s(mod 997)

; (% = "mod")

i	pat.charAt(i)					
	0	1	2	3	4	
	2	6	5	3	5	% 997 = 613 (Q = 997)

txt.charAt(i)

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
0	3	1	4	1	5	% 997 = 508										
1		1	4	1	5	9	% 997 = 201									
2			4	1	5	9	2	% 997 = 715								
3				1	5	9	2	6	% 997 = 971							
4					5	9	2	6	5	% 997 = 442						
5						9	2	6	5	3	% 997 = 929					
6 ⟵ *return* i = 6							2	6	5	3	5	% 997 = 613 *match*				

Modular Hashing of Strings with General Alphabet (Division Method)

R = size of alphabet (# of distinct characters that can appear in text)
M = length of pattern

1st char of text (represented in range {0,1,..., R-1})

$$x_0 = \left(t_0 \cdot R^{M-1} + t_1 \cdot R^{M-2} + \ldots + t_{M-1} \cdot \underbrace{R^0}_{=1}\right) \bmod Q$$

Hash value of the initial M characters of text

Key Challenges

Challenge 1: If M is large, might have numerical overflow

Challenge 2: Hashing one substring (of length m): costs M
Hashing N-M+1 : cost order NM

Challenge 1

If M is large, then the number will overflow.

R = size of alphabet ; M = length of pattern

$$X_0 = \left(t_0 \cdot R^{m-1} + t_1 \cdot R^{m-2} + \dots + t_{m-1} \cdot \underbrace{R^0}_{=1}\right) \bmod Q$$

Two Modular Arithmetic Identities

1) $(a+b) \bmod Q = ((a \bmod Q) + (b \bmod Q)) \bmod Q$
2) $(a \cdot b) \bmod Q = ((a \bmod Q) \cdot (b \bmod Q)) \bmod Q$

$$\begin{aligned} X_0 &= (R^2 \cdot t_0 + R^1 \cdot t_1 + R^0 t_2) \bmod Q \\ &= ((t_0 R + t_1) \cdot R + t_2) \bmod Q \\ &= ((((t_0 \bmod Q) \cdot R + t_1) \bmod Q) \cdot R + t_2) \bmod Q \end{aligned}$$

Horner's Method (runtime NM)

```
h = 0
for i = 0 → M-1
    h = (h·R + t_i) mod Q
return h
```

Example (M=3)

$t_0 \bmod Q$
$((t_0 \bmod Q)R + t_1) \bmod Q$
$((((t_0 \bmod Q)R + t_1) \bmod Q)R + t_2) \bmod Q$

Challenge 2:

Avoiding total cost of MN.

X_i is hash value for $t_i\, t_{i+1} \dots t_{i+m-1}$
X_{i+1} is hash value for $t_{i+1}\, t_{i+2} \dots t_{i+m}$

$$X_i = (t_i\, R^{M-1} + t_{i+1} \cdot R^{m-2} + \dots + t_{i+m-1} \cdot R^0) \bmod Q$$

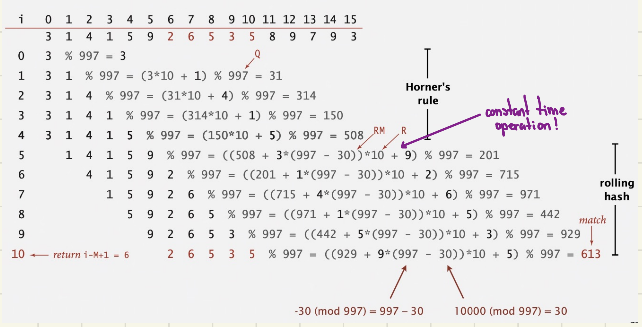
$$X_{i+1} = (t_{i+1} \cdot R^{M-1} + \dots + t_{i+m-1} \cdot R^1 + t_{i+m} \cdot R^0) \bmod Q$$

$$\begin{aligned} X_{i+1} &= ((X_i - t_i \cdot R^{m-1}) R + t_{i+m}) \bmod Q \\ &= ((X_i - t_i \cdot \bar{R}) R + t_{i+m}) \bmod Q \end{aligned}$$

$\nwarrow$ $(R^{m-1} \bmod Q)$ (precompute)

Rabin-Karp Substring Search Example

First R entries: Use Horner's rule.
Remaining entries: Use rolling hash (and % to avoid overflow)



$R^{M-1} = 10000$

$\overline{R}M = R^{M-1} \bmod Q = 30$

Rabin-Karp Analysis

Theory. If Q is a sufficiently large random prime (about MN^2), then the probability of a false collision is about $1/N$. → over entire course of algorithm

Practice. Choose Q to be a large prime (but not so large to cause overflow). Under reasonable assumptions, probability of a collisions is about $1/Q$.
↖ single hash

Las-Vegas Algorithm

Use Rabin-Karp to find hash matches, and upon each hash match, check if substrings of text actually matches pattern. ⟶ Cost: M

Expected Cost of Algorithm.

$$O(N + N\cdot(1/Q)\cdot M) \longrightarrow Q = \cancel{(1/M)} = M$$

$$O(N + (N/M)\cdot M)$$

$$= O(N)$$

$\boxed{\text{Suppose } Q \doteq M}$

Note: Always returns correct answer
Extremely likely to run in linear time (but worst case is MN)

Monte Carlo Algorithm

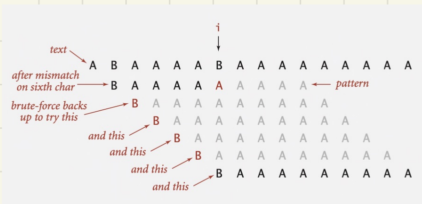
Always runs in linear time.
Extremely likely to return correct answer (but not always!).

Advantages: Extends to 2D patterns. Extends to finding multiple patterns.
Disadvantages: Arithmetic ops slower than char compares. Las Vegas version requires backup. Poor worst-case guarantee.

Knuth-Morris-Pratt Substring Search

Intuition. Suppose we are searching in text for pattern BAAA AAA AAA.

- Suppose we match 5 chars in pattern, with mismatch on 6^{th} char.
- We know previous 6 chars in text are BAAAAB. ⟵ assuming {A, B} alphabet.
- Don't need to back up text pointer!



but no backup is needed! ⟶ BAAAAAAAAA

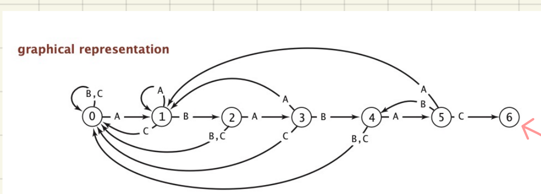
Knuth-Morris-Pratt Algorithm. Clever method to always avoid backup. (!)

Deterministic Finite State Automation (DFA)

⟵ (CSC 320 – Turing Machines)

DFA is abstract string-searching machine.

- Finite number of states (including start and halt).
- Exactly one transition for each char in alphabet.
- Accept if sequence of transitions leads to halt state



Alphabet = {A, B, C}

Arrives here if and only if we have a match!

internal representation

	j	0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

variable c

If in state j reading char c:
  if j is 6 halt and accept
  else move to state dfa[c][j]

November 29th 2021 (Lecture 20)

Interpretation of Knuth-Morris-Pratt DFA

Question. What is interpretation of DFA state after reading in txt[i]?

Answer. State = number of characters in pattern that have been matched.

length of longest prefix pat[] that is a suffix of txt[0..i]

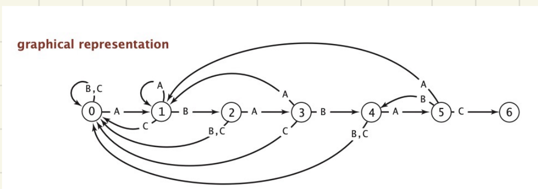
Example. DFA is in state 3 after reading in txt[0..6].

txt →

0	1	2	3	4	5	6 (i)	7	8
B	C	B	A	A	B	A	C	A

pat →

0	1	2	3	4	5
A	B	A	B	A	C



txt →

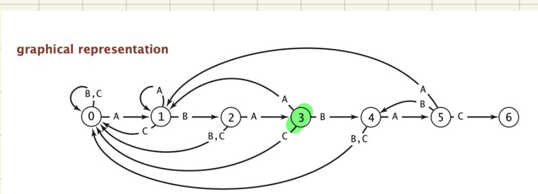
0	1	2	3	4	5	6 (i)	7	8
B	C	B	A	A	B	A	C	A

suffix of txt[0..6]

pat →

0	1	2	3	4	5
A	B	A	B	A	C

prefix of pat[]



Knuth-Morris-Pratt Substring Search: Java Implementation

Key differences from brute-force implementation.

- Need to precompute dfa[][] from pattern
- Text pointer i never decrements.

```
public int search(String txt) {
    int i, j, N = txt.length();
    for (i = 0, j = 0; i < N && j < M; i++) {
        j = dfa[txt.charAt(i)][j];        ← NO BACKUP!
    }
    if (j == M) return i - M;             ← found pattern!
    else        return N;
}
```

where i indicates what character
j indicates the state

Running time.

- Simulate DFA on text: at most N character accesses.
- Build DFA: how to do efficiently? [Warning: tricky algorithm ahead]

Knuth-Morris-Pratt Demo: DFA Construction

Include one state for each character in pattern (plus accept state)

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
	A						
dfa[][j]	B						
	C						

Constructing the DFA for KMP substring search for ABABAC

$0 \xrightarrow{A} 1 \xrightarrow{B} 2 \quad 3 \quad 4 \quad 5 \quad 6$

Match Transition. If in state j and next char c == pat.charAt(j), go to j+1.

first j characters of pattern have already been matched

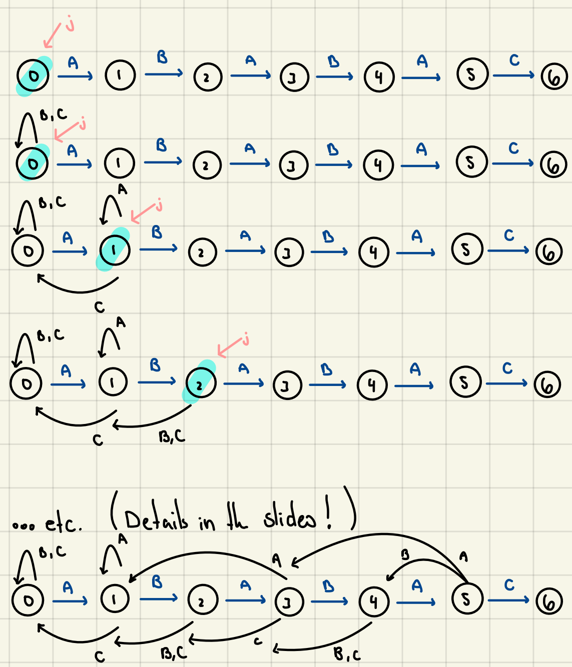
next char matches

now first j+1 characters of pattern have been matched

$0 \xrightarrow{A} 1 \xrightarrow{B} 2 \xrightarrow{A} 3 \xrightarrow{B} 4 \xrightarrow{A} 5 \xrightarrow{C} 6$

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
	A	1		3		5	
dfa[][j]	B		2		4		
	C						6

Mismatch Transition. Back up if c != pat.charAt(j).



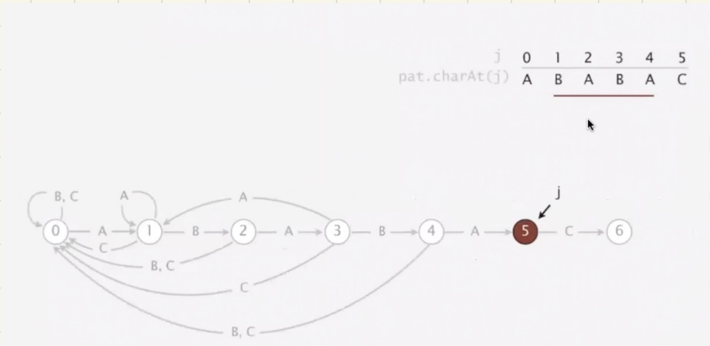
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
	A	1	1	3	1	5	1
dfa[][j]	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

Suppose mismatch: AA , AC
→ shift i by 1 in text → shift i by 1
→ A → C

ABB
↓ BB Ø match at all
↓ B Ø match at all

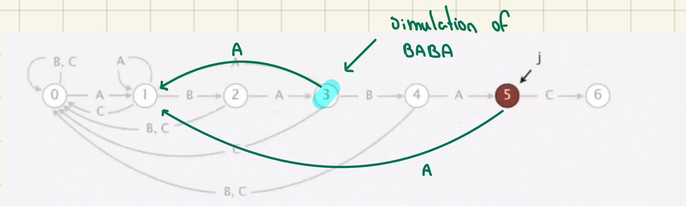
<u>Mismatch Transition.</u> If in state j and next char c != pat.charAt(j),
then the last $j-1$ characters of input are pat[1..j-1], followed by c.

<u>To compute dfa[c][j]:</u> Simulate pat[1..j-1] on DFA and take transition c.
state X
still under construction!

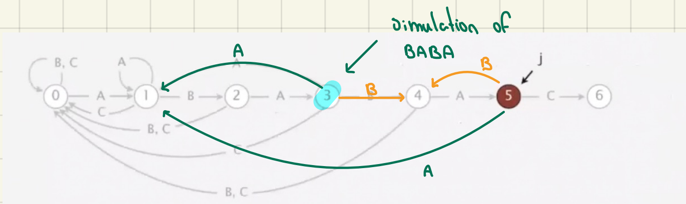


Ex. dfa['A'][5] = 1;
Simulate BABA;
take transition 'A' = dfa['A'][3]

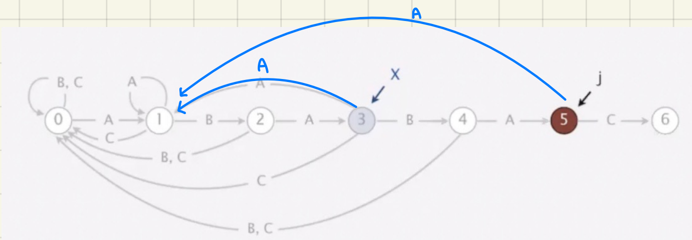
<u>Note:</u> Memorization!



Ex. dfa['B'][5] = 4;
Simulate BABA;
take transition 'B' = dfa['B'][3]



<u>Running Time.</u> Seems to require j steps.
Takes only constant time if we maintain state X.

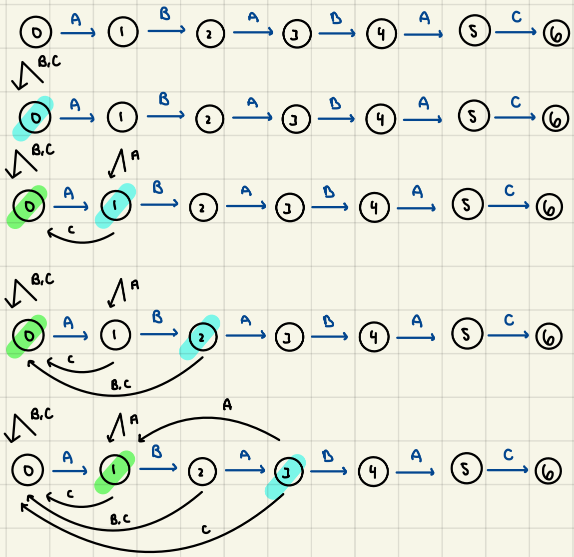


Ex. dfa['A'][5] = 1;
from state X,
take transition 'A'
= dfa['A'][X]

Knuth-Morris-Pratt Demo: DFA Construction in Linear Time

<u>Match Transition.</u> For each state j, dfa[pat.charAt(j)][j] = j+1.
first j characters of pattern have already been matched.
Now first j+1 characters of pattern have been matched.

<u>Mismatch Transition.</u> For state 0 and char c != pat.charAt(j), set dfa[c][0] = 0.



= j, = X

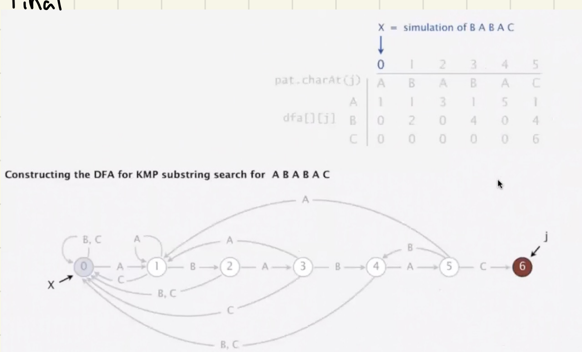
X = simulation of empty string

	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][j] A	1	1	3		5	
B	0	2	0	4		
C	0	0	0			6

X = simulation of BA

	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][j] A	1	1	3	1	5	
B	0	2	0	4		
C	0	0	0	0		6

Final


X = simulation of B A B A C

	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][j] A	1	1	3	1	5	1
B	0	2	0	4	0	4
C	0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C

<u>Running Time</u>. M characters accesses (but space / time proportional to RM).

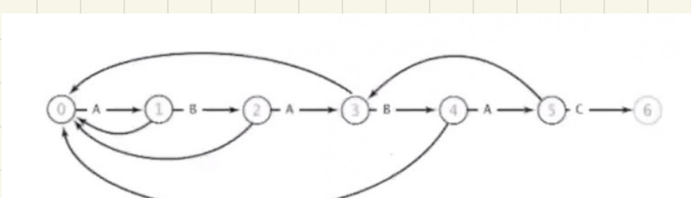
KMP Substring Search Analysis

<u>Proposition</u>. KMP substring search accesses no more than M+N chars to search for a pattern of length M in a text of length N.

<u>Proof</u>. Each pattern char accessed once when constructing the DFA; each text char accessed once (in the worst case) when simulating the DFA.

<u>Proposition</u>. KMP constructs dfa[][] in time and space proportional to RM.

<u>Larger Alphabets</u>. Improved version of KMP constructs nfa[][] in time and space proportional to M.


KMP NFA for ABABAC

November 29th 2021 (Lecture 20)

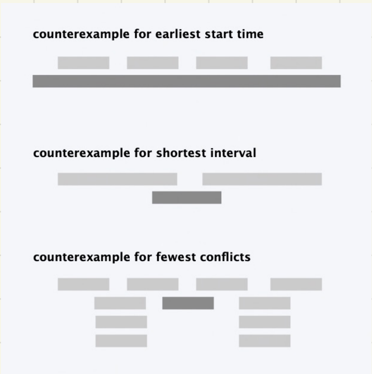
Greedy Algorithms – Interval Scheduling

- Job j starts at s_j and finishes at f_j. Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.

Greedy Template. Consider jobs in some natural order. Take each job provided it's compatible with the ones already taken.

- [Earliest Start Time] Consider jobs in ascending order of s_j.
- [Earliest finish Time] Consider jobs in ascending order of f_j.
- [Shortest Interval] Consider jobs in ascending order of $f_j - s_j$.
- [fewest Conflicts] For each job j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

Examples and Counter Examples.



None of these provide a consistent optimal solution.

Earliest – Finish – Time – First Algorithm

```
EARLIEST-FINISH-TIME-FIRST (n, s1, s2, ..., sn, f1, f2, ..., fn)
    SORT jobs by finish time so that f1 ≤ f2 ≤ ... ≤ fn
    A ← ϕ   ← set of jobs selected
    FOR j = 1 TO n
        IF job j is compatible with A
            A ← A ∪ {j}
    RETURN A
```

This job w/ earliest finish time will always be run.

Proposition. Can implement earliest-finish-time first in $O(n \log n)$ time.

- Keep track of job j^* that was added last to A.
- Job j is compatible with A iff $s_j \geq f_{j^*}$.
- Sorting by finish time takes $O(n \log n)$ time.

Theorem. Earliest-First-Time first is optimal (the schedule A that algorithm returns maximizes the number of jobs that we can run on a single computer, among all schedules.)

Let O be optimal schedule. $O = (o_1, o_2, \dots, o_m)$
Let A be algorithms schedule. $A = (a_1, a_2, \dots, a_K)$

Proof $(K = m)$

Lemma "Greedy Stays Ahead"

For all $r \le K$, $f(a_r) \le f(o_r)$

Proof by Induction

1) Base Case: $f(a_1) \le f(o_1)$ ✓ Because $f(a_1) \le f(i)$ $\forall i = 1, \dots, n$
It is true that $f(a_1)$ is the min finish time amoungst all jobs, becase of ascending order!

2) I.H (Assume) $f(a_{r-1}) \le f(o_{r-1})$

3) I.S (Show) $f(a_r) \le f(o_r)$

Proof $\longrightarrow$ $s(o_r) \ge f(o_{r-1}) \overset{(I.H)}{\ge} f(a_{r-1})$

When A considers the r^{th} job to add, it will be able run job o_r.

Algorithm considered o_r and a_r and it chose a_r $\Rightarrow f(a_r) \le f(o_r)$

Proof of Theorem by Contradiction

Suppose $K < m$ (suppose greedy is suboptimal)

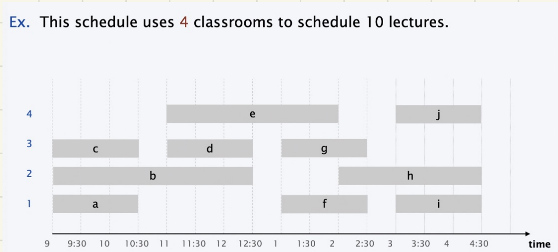
From Lemma, well, $f(a_K) \le f(o_K)$

Therefore, Algorithm could add job o_{K+1} ↯

But then, job o_{K+1} is compatible with $(a_1, a_2, \dots, a_K)$
↳ A could also added job o_{K+1}.

Interval Partitioning

- △ Lecture j starts at s_j and finishes at f_j.
- △ Goal: Find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.



Earliest Start Time First Algorithm

EARLIEST-START-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT lectures by start time so that $s_1 \leq s_2 \leq \ldots \leq s_n$.
$d \leftarrow 0$ ⟵ number of allocated classrooms
FOR $j = 1$ TO n
  IF lecture j is compatible with some classroom
    Schedule lecture j in any such classroom k.
  ELSE
    Allocate a new classroom $d+1$.
    Schedule lecture j in classroom $d+1$.
    $d \leftarrow d+1$
RETURN schedule.

Proposition. The earliest-start-time-first algorithm can be implemented in $O(n \log n)$ time.

Proof. Store classrooms in a priority queue (key = finish time of its last lecture).
- To determine wether lecture j is compatible with some classroom, compare s_j to key of min classroom k in priority queue.
- To add lecture j to classroom k, increase key of classroom k to f_j.
- Total number of priority queue operations is $O(n)$.
- Sorting by start time takes $O(n \log n)$ time. ■

Remark. This implementation chooses the classroom k whose finish time of its last lecture is the earliest.

Lower Bound On Optimal Solution

Definition. The depth d^* of a set of open intervals is the maximum number that contain any given time.

Key Observation. Number of classrooms needed $\geq$ depth.

Question. Does number of classrooms needed always equal depth?
Answer. YES! Moreover, earliest-start-time-first algorithm finds one

At end of our algorithm, $\underbrace{d = d^*}$ ⟶ to be proved!

Proof that $d = d^*$ ← depth
↑ at end of algorithm

Suppose we are considering scheduling the j^{th} lecture. We open new classroom if all currently open classrooms are running lectures (↔ intervals) that intersect lecture j (↔ interval j).

At most $d^* - 1$ such lectures intersect, (by definition of depth).

So, one classroom must be available for j.
So, we will never open more than d^* classrooms.

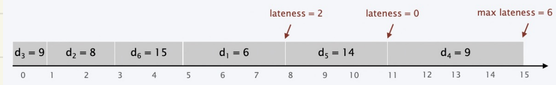
Scheduling to Minimizing Lateness

Minimizing Lateness Problem (input : $n, t_1, t_2, \ldots, t_n, d_1, d_2, \ldots d_n$)

- Single resource processes one job at a time.
- Job j requires t_j units of processing time and is due at time d_j.
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
- Lateness : $\ell_j = \max\{0, f_j - d_j\}$.

- Goal: Schedule all jobs to minimize maximum lateness $L = \max_j \ell_j$.

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Example : [Shortest Processing Time first] Schedule jobs in ascending order of processing time t_j.

	1	2
t_j	1	10
d_j	100	10

Counterexample!

Job 1 — not late — $\ell_1 = \max\{0, 1-100\} = 0$

Job 2 — late — $\ell_2 = \max\{0, 11-10\} = 1$

Earliest Deadline First

```
EARLIEST-DEADLINE-FIRST (n, t1, t2, .. tn, d1, d2, ... dn)
  SORT n jobs so that d1 ≤ d2 ≤ ... ≤ dn
  t ← 0
  FOR j = 1 TO n
     Assign job j to interval [t, t + tj].
     sj ← t ; fj ← t + tj
     t ← t + tj
  RETURN intervals [s1, f1], [s2, f2], ..., [sn, fn].
```

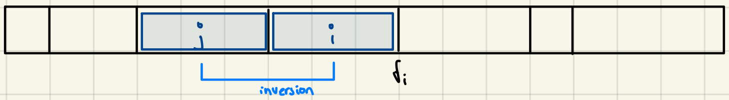
No Idle Chat

Observation 1. There exists an optimal schedule with no idle time

Observation 2. The earliest-deadline-first schedule has no idle time.

Inversions

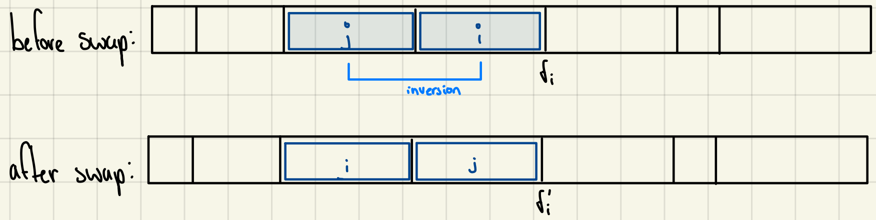
Definition Given a schedule S, an inversion is a pair of jobs i and j such that: $i < j$ but j scheduled before i. $d_i < d_j$



[as before, we assume jobs are numbered so that $d_1 \le d_2 \le \ldots \le d_n$]

Observation 3. The earliest-deadline-first schedule has no inversions.

Observation 4. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.



Claim. Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

Proof. Let ℓ be the lateness before the swap, and let ℓ' be it afterwards.

- $\ell'_k = \ell_k$ for all $k \neq i, j$.
- $\ell'_i \leq \ell_i$.
- If job j is late, $\ell'_j = f'_j - d_j$ (definition)

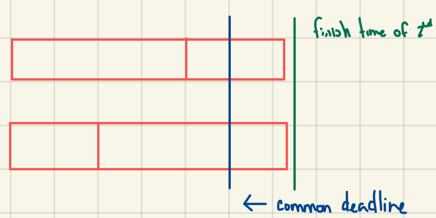
$= f_i - d_j$ (j now finishes at time f_i)

$\leq f_i - d_i$ (since i and j inverted)

$\leq \ell_i$. (definition)

Proof of Optimality of Our Greedy Algorithm

First. All schedules with no inversions and no idle time have same maximum lateness.



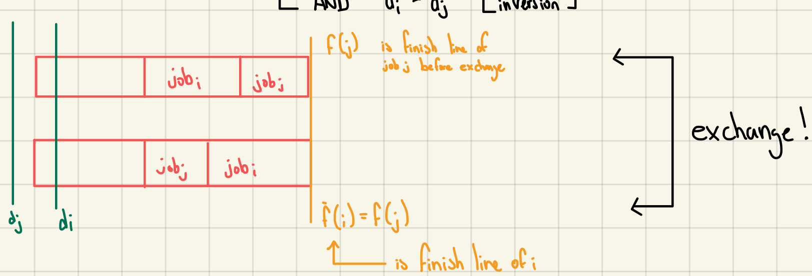
Max-lateness will be same among the 2 jobs.

Claim. There is an optimal schedule with no inversions and no idle time.

Proof (by "Exchange Argument")

Suppose O is an optimal schedule, and suppose O has an inversion.

Then, $\exists\, i, j$ such that [job i is immediately followed by job j AND $d_i > d_j$ [inversion]]



$\bar{\ell}_i = \max\{0, \bar{f}(i) - d_i\}$

$= \max\{0, f(j) - d_i\}$

$\leq \max\{0, f(j) - d_j\}$

$= \ell_j$

$\leq$ max lateness before exchange!

Greedy Analysis Strategies

- Greedy Algorithm Stays Ahead: Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

- Structural: Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

- Exchange Argument: Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

- Other Greedy Algorithms: Gale-Shapley, Kruskal, Prim, Dijkstra, Huffman, ...

December 2nd 2021 (Lecture 21)

Interval Partitioning

d = # classrooms used by our algorithm

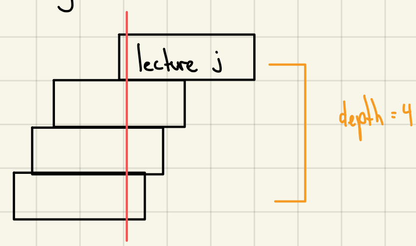
Claim. $d = d^*$ ← depth = best possible

Proof: by Contradiction

Suppose scheduling j^{th} lecture (in order of increasing start time) and we already opened d^* classrooms, and they are all occupied (each open classroom currently has running lecture which intersects j^{th} lecture).

if this happen!
then depth $\geq d^* + 1$ ↯

$\Rightarrow$ open $(d^*+1)^{th}$ classroom



suppose $d^* = 3$

Dynamic Programming

Algorithmic Paradigms

- ☐ Greedy. Build up a solution incrementally, myopically optimizing some local criterion.
- ☐ Divide-And-Conquer. Break up a problem into independent subproblems, solve each subproblem, and combine solution to subproblems to form solution to original problem.
- ☐ Dynamic Programming. Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.

  ↑ Fancy name for caching away intermediate results in a table for later reuse.

History: Bellman pioneered the systematic study of dynamic programming in 1950s.

Application: Bioinformatics, Control Theory, Information Theory, Operations Research, Theory, Graphics

Algorithms. Unix diff, Bellman-Ford

Weighted Interval Scheduling

Weighted Interval Scheduling Problem.

- ☐ Job j starts at s_j, finishes at f_j, and has weight or value v_j
- ☐ Two jobs compatible if they don't overlap.
- ☐ Goal. Find maximum weight subset of mutually compatible jobs.

Earliest-finish-time first

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previous chosen jobs.

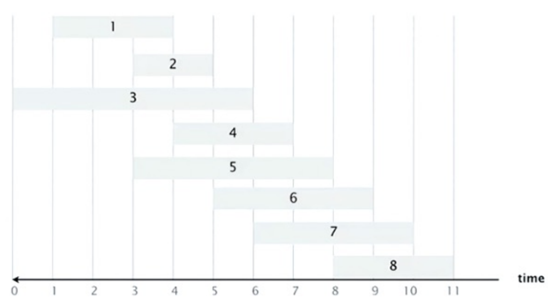
Recall. Greedy algorithm is correct if all weights are 1.

Observation. Greedy algorithm fails spectacularly for weighted version.

Notation. Label jobs by finishing time: $f_1 \le f_2 \le \ldots \le f_n$.

Definition. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Ex. $p(8) = 5, p(7) = 3, p(2) = 0$.



December 6th 2021 (Lecture 22)   (Note: Lecture had no audio "/)

Dynamic Programming: Binary Choice   (OPT (0) = 0)

Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, . . ., j.

Case 1. OPT Selects job j.   j = 0, 1, 2, . . ., n ← # jobs

- Collect profit v_j.
- Can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }. V_j + OPT (p[j])
- Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j).

Case 2. OPT Does Not select job j.   0 + OPT (j - 1)   optimal substructure property (proof via exchange argument)

- Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j - 1.

$$\text{OPT}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{ v_j + \text{OPT}(p(j)),\ \text{OPT}(j-1) \} & \text{otherwise} \end{cases}$$

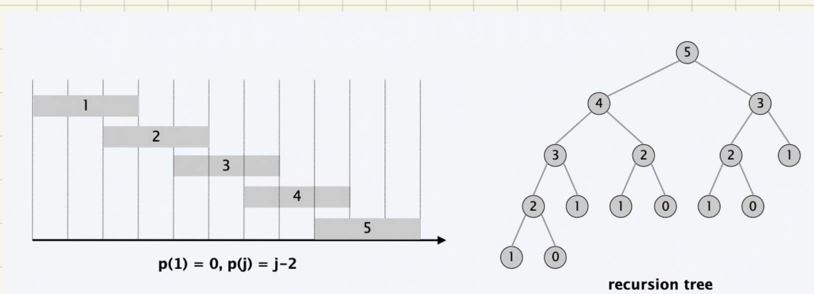
Weighted Interval Scheduling: Brute Force

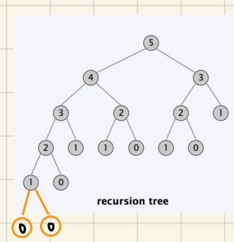
Input: n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ ... ≤ f[n].
Compute p[1], p[2], ..., p[n]. ← exercise: How to do this efficiently?
cost: O(n log(n))

```
Compute-Opt (j)
   if j = 0
      return 0.
   else
      return max(v[j] + Compute-Opt (p[j]), Compute-Opt (j-1)).
```

Observation. Recursive algorithm fails spectacularly because of redundant subproblems
=> Exponential Algorithms

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci Sequence.





Proof ?

recursive calls made by compute-opt(j) = T(j)

$T(0) = 1$
$T(1) = T(0) + T(0) = 2$
$T(2) = T(1) + T(0) = 2 + 1 = 3$
$T(3) = T(2) + T(1) = 3 + 1 = 5$
$\vdots$
$T(j) = T(j-1) + T(j-2)$

$T(j) = T(j+2)^{th}$ Fibonacci #

As $j \to \infty$ $\quad T(j) \to \varphi^j / \sqrt{5}$ (φ = 1.6...)

Weighted Interval Scheduling: Memoization

Memoization. Cache results of each subproblem; lookup as needed.

Input: n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ . . ≤ f[n].
Compute p[1], p[2], ..., p[n].

```
For j=1 to n
    M[j] ← Empty.
M[0] ← 0.
```

M-Compute-Opt(j)

```
if M[j] is empty
    M[j] ← max(v[j] + M-Compute-Opt(p[j]), M-Compute-Opt(j-1)).
return M[j].
```

Weighted Interval Scheduling: Running Time

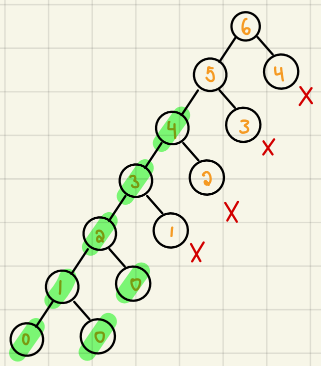
Claim. Memoized version of algorithm takes O(n logn) time.

- Sort by finish time: O(n logn)
- Computing p(·): O(n logn) via sorting by start time.
- M-COMPUTE-OPT(j): each invocation takes O(1) time and either...
  i) returns an existing value M[j]
  ii) fills in one new entry M[j] and makes two recursive calls

- Progress measure Φ = # nonempty entries of M[ ].
  - initially $\Phi = 0$, throughout $\Phi \le n$.
  - (ii) increases Φ by 1 $\Rightarrow$ at most $2n$ recursive calls.

- Overall running time of M-COMPUTE-OPT(n) is O(n). ∎

<u>Remark.</u> O(n) if jobs are presorted by start and finish time.



Weighted Interval Scheduling: Finding a Solution

Q. DP algorithm computes optimal value. How to find solution itself?
A. Make a second pass.

```
Find-Solution
    if j=0
        return ∅.
    else if (v[j] + M[p[j]] > M[j-1])
        return {j} ∪ Find-Solution(p[j]).
    else
        return Find-Solution(j-1)
```

ex.

j	0	1	2	3	4	5	6	7	8
	...	...	...	...	...	M(5)	m(6)	m(7)	m(8)

$\emptyset$ 8

let p[j] = j-2
if v[8] + M[6] > M[7]
if v[6] + M[4] ≤ M[5]

<u>Analysis.</u> # of recursive calls $\le n \Rightarrow O(n)$.

Weighted Interval Scheduling: Bottom-Up

<u>Bottom-Up Dynamic Programming.</u> Unwind Recursion.

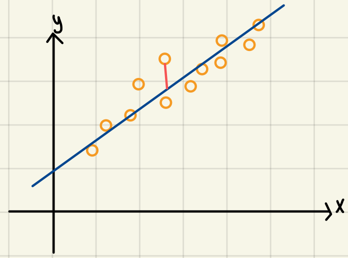
```
Bottom-Up (n, s_1, ..., s_n, f_1, ..., f_n, v_1, ..., v_n)
    Sort jobs by finish time so that f_1 ≤ f_2 ≤ ... ≤ f_n.
    Compute p(1), p(2), ..., p(n).
    M[0] ← 0.
    For j=1 to n
        M[j] ← max { v_j + M[p(j)], M[j-1] }.
```

<u>Exercise:</u> Can you solve problem using jobs sorted by start time?

Least Squares (Foundational Problem in Statistics)

- △ Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- △ Find a line $y = ax + b$ that minimizes the sum of the squared error:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$



Solution. Calculus => min error is achieved when

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Knapsack Problem

- △ Given n objects and a "Knapsack".
- △ Item i weighs $w_i > 0$ and has value $v_i > 0$.
- △ Knapsack has capacity of W.
- △ Goal. Fill Knapsack so as to maximize total value.

$n = 5$

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack instance (weight limit $W = 11$)

Greedy by Value. Repeatedly add item with maximum v_i.
Greedy by Weight. Repeatedly add item with minimum w_i.
Greedy by Ratio. Repeatedly add item with maximum ratio v_i / w_i.

ex. $\{1, 2, 5\}$ has value 35.
$\{3, 4\}$ has value 40.
$\{3, 5\}$ has value 46 (but exceeds weight limit)

Observation. None of greedy algorithms is optimal.

Dynamic Programming: False Start

Def. $OPT(i)$ = max profit subset of items $1, \ldots, i$.

Case 1. OPT Does Not Select Item i.
- OPT selects best of $\{1, 2, \ldots, i-1\}$. ← optimal substructure property (proof via exchange argument)

Case 2. OPT selects item i.
- Selecting item i does not immediately imply that we will have to reject other items.
- Without knowing what other items were selected before i, we don't even know if we have enough room for i.

Conclusion. Need more subproblems!

Dynamic Programming: Adding a New Variable

(Full Problem: $OPT(n, W)$)

Def. $OPT(i, w)$ = max profit subset of items $1, \ldots, i$ with limit w.

Case 1. OPT Does Not Select Item i. $OPT(i-1, w)$
- OPT selects best of $\{1, 2, \ldots, i-1\}$ using weight limit w. ← optimal substructure property (proof via exchange argument)

Case 2. OPT selects item i.

- New weight limit $= w - w_i$. $OPT(i-1, w-w_i)$ ← optimal substructure property (proof via exchange argument)
- OPT selects best of $\{1, 2, \dots, i-1\}$ using this new weight limit.

$$OPT(i,w) = \begin{cases} 0 & \text{if } i=0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ v_i + OPT(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

$M[i, w]$; where i = item and w = weight limit

KNAPSACK $(n, W, w_1, \dots, w_n, v_1, \dots, v_n)$

```
FOR w=0 TO W
    M[0,w] ← 0.

For i=1 TO n
    FOR w=0 TO W
    IF (w_i > w) M[i,w] ← M[i-1,w].
    ELSE         M[i,w] ← max {M[i-1,w], v_i + M[i-1, w-w_i]}.
                              exclude i    include i

RETURN M[n,W].
```

Demo

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i=0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ v_i + OPT(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

exclude

include 3

18 + 22 = 40

weight limit w

subset of items 1, …, i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

★ include 4 , ▲ exclude 5

Knapsack Problem: Running Time

Theorem There exists an algorithm to solve the Knapsack problem with n items and maximum weight W in $\Theta(nW)$ time and $\Theta(nW)$ space.

↑ weights are integers between 1 and W

Proof.

- Takes $O(1)$ time per table entry.
- There are $\Theta(nW)$ table entries.
- After computing optimal values, can trace back to find solution: take item i in $OPT(i,w)$ iff $M[i,w] > M[i-1,w]$. ∎

Remarks.

- Not polynomial in input size! ← "pseudo-polynomial"
- Decision version of Knapsack problem is NP-COMPLETE. [CHAPTER 8]
- There exists a poly-time algorithm that produces a feasible solution that has value within 1% of optimum. [SECTION 11.8]