2 Chapter 9: Sorting in Linear Time

2.1 Counting Sort

(a) Rank each element

i. Count how many times i occurs

ii. Prefix sum on counter array to find how many items $\leq i$

(b) Place at its location.

Start from right, place item in the output array, decrease its corresponding count.

$W(n) = O(n)$ if range is $O(n)$.

Stable sorts – counting sort, mergesort, Insertion sort, Bucket sort.

Nonstable sorts – Quicksort, heapsort.

2.2 Bucket Sort

k Buckets.

Algorithm:

1. hash items among buckets
2. sort the buckets
3. Combine buckets

Let there be k buckets, n items

1. distribution $O(n)$
2. sort the buckets
   $w(n) = O(n \log n)$
   $A(n) = O(k\frac{n}{k} \log \frac{n}{k}) = O(n \log(n/k))$
3. combine buckets $O(n)$.

Thus, bucket sort is

$$w(n) = O(n \log n)$$

$$A(n) = O(n \log \frac{n}{k})$$

If k is constant,

$$A(n) = O(n \log n - n \log k)$$
$$= O(n \log n)$$

$A(n) = O(n)$ if $k = n/20$, $A(n) = O(n)$

Good when item distribution is known so that bucket get equitable number of keys.

Space Usage

worst-case: each bucket should have space for n key (any allocation)

$\Rightarrow$ total $= O(nk)$

Thus, as k increases, average space increases but so does the space requirement.

If linked allocation is used

Space needed $= O(k) + O(n) = O(n + k) = O(n)$

However sorting each bucket using quicksort, mergesort, and heapsort will be difficult which require array representation.

If insertion sort is used to sort linked list, (buckets),

$$A(n) = O(\frac{n^2}{k^2}) * k = O(\frac{n^2}{k}) = O(n) \text{ for k} = \text{O(n)}.$$

Pseudocode

2.3 Radix sort

for $i \leftarrow 1$ to d

stable sort A on digit i.

(a) counting sort can be used

(b) bucket sort can also be used

Pseudocode

Finding largest: $n-1$ comparisons

Finding second largest: .

- Two scans: $(n-1) + (n-2) = 2n-3$ comparisons

- Divide & Conquer:
  $W(n) = 2W(n/2) + 2$
  $= 2 + 2(2 + 2W(n/2^2))$
  $= 2 + 2^2 + 2^2 W(n/2^2)$
  $= 2 + 2^2 + 2^2(2 + 2W(n/2^3))$

$= 2 + 2^2 + 2^3 + 2^3 W(n/2^3)$

Let $n = 2^k$,

We know, $W(2) = 1 \Rightarrow W(n/2^{k-1}) = 1$

$W(n) = 2 + 2^2 + 2^3 + \cdots + 2^{k-1} + 2^{k-1} W(n/2^{k-1})$

$= 2^1 + 2^2 + \cdots + 2^{k-1} + 2^{k-1}$

$= (2^k - 2) + 2^{k-1}$

$= n - 2 + n/2$

$= 3n/2 - 2$

- Using Tournament Tree

  To build a tournament tree requires $n - 1$ comparisons.

  $W(n) = n - 1$ for max

  $W(n) = n - 1 + (\log_2 n - 1)$ for 2max

2.1 Selection of kth smallest/largest

1. Sorting based

   $W(n) = O(n \log n)$

2. Tournament tree based

   $W(n) = O(n + k \log n) = O(n)$ for $k \leq \frac{n}{\log n}$ or $k \geq \frac{n}{\log n}$

2.2 A Good Av. Case Algorithm

Divide & conquer

Selection$(A[p,r],k)$:

$j =$ Partition2(A,p,r)

if $k < j$

return Selection$(A[p..(j-1)],k)$

else if k = j **then return** $L[j]$

else $\{k > j\}$

return Selection$(A[j+1..r],k-j)$

$W(n) = n - 1 + W(n-1) = O(n^2)$

$A(n) \approx n - 1 + A(n/2)$

$= \ (n-1) + (n/2 - 1) + (n/4 - 1) + \cdots + 1$

$< 2n \in O(n)$

(gross simplification)

2.3 Worst-case $O(n)$ algorithm

To improve $W(n)$, we must ensure a good split point.

Selection'$(A[p, r], k)$

1. Divide A in $\frac{n}{r}$ sublist ($r = 5, 7$, etc.), $n = r - p + 1$.
2. Find median of each of the $\frac{n}{r}$ sublists.
3. Recursively find median of these $\frac{n}{r}$ medians.
4. Use median of medians (MM) as the pivot in the previous algorithm for selection:

   Let MM be at index i. Swap$(A[p], A[i])$

$j = \text{Partition2}(A, p, r)$

5. Choose the appropriate partition for further search:

if $k < j$

return Selection'$(A[p..(j-1)], k)$

else if k=j **then return** $L[j]$

else $\{k > j\}$ **return** Selection'$(A[j+1..r], k-j)$

$T(n) \leq cn$ (Steps 1, 2, 4: for finding n/r medians and for partitioning the array based on pivot chosen as median of medians)

$+ T(n/5)$ (for step 3, recursively finding median of $n/5$ medians)

$+ T(3n/4)$ (for recursive call to larger partition)

To Prove: Let

$$T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right)$$

for $n \geq 5$. Then

$$T(n) \leq 20cn.$$

Choose c large enough such that $T(n) \leq cn$ for $n \leq 24$ (for list of size less than 5, there is no recursive call).

Basis: For $n \leq 24$, by choice of c, $T(n) \leq cn \leq 20cn$.

Hypothesis: Assume for $k \geq 24$, $T(k) \leq 20ck$.

Induction:

To show that $T(k+1) \leq 20c(k+1)$.

$$T(k+1) \leq c(k+1) + T\left(\frac{k+1}{5}\right)$$

$$
\begin{aligned}
T\left(\frac{3(k+1)}{4}\right) & \\
& \leq c(k+1) \ + \ 20\left(\frac{k+1}{5}\right)c \\
& \quad 20\left(\frac{3(k+1)}{4}\right)c \\
& = (k+1)(c+4c+15c) \\
& = 20(k+1)c
\end{aligned}
$$

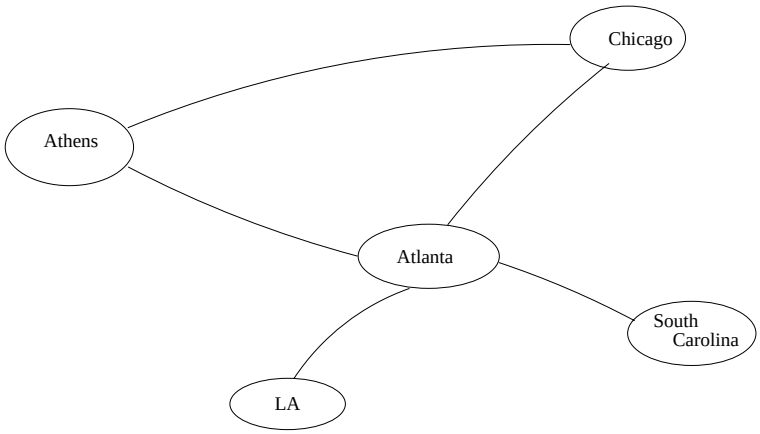
How to make quicksort $O(n \log n)$? Use selection algo to find the median.

Use median as partitioning element in Quicksort.

Complexity $T(n) = cn + 2T(n/2) = O(n \log n)$

2 GRAPHS

Consists of a set of nodes or point some of which are connected by edges or lines.



A (hypothetical) graph of nonstop airline flights.

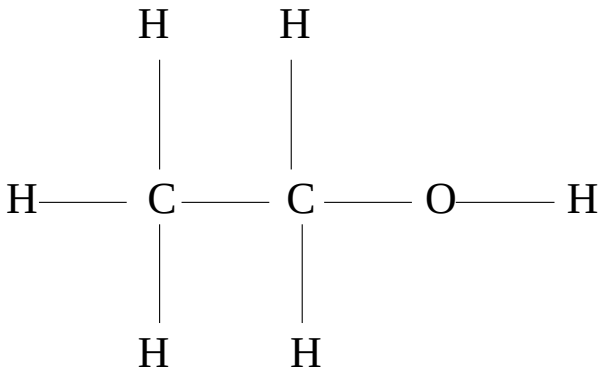
Molecule alcohol: CH_3CH_2OH

Def. A graph $G = (V, E)$ which V is the set of nodes,

$$V = \{v_1, v_2, \ldots, v_n\}$$

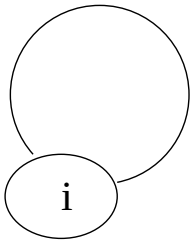
and E is the set of edges,

$$E = \{\{v_i, v_j\} | v_i \in V, v_j \in V\}$$



note:.

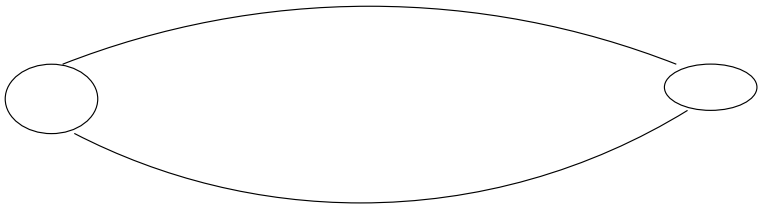
1. each edge is a set of pair of vertexes. No order implied.
2. $v\{v_i, v_j\}$, i could be equal to j.
   self-loop



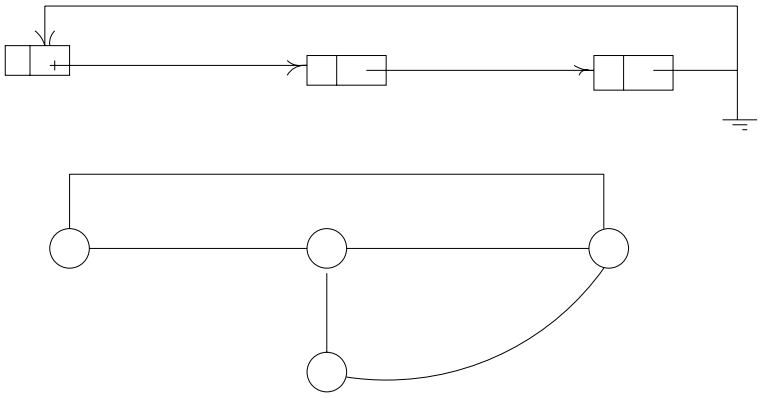
self loop

3. One also allows multiple edge between two node in a general graph

4. Cardinality of V, $|V| = n$, number of nodes, $|E| = m$, number of edges.



3 DIGRAPH



eg. A flow chart of a program

def. A graph whose edges are directed is a digraph, directed graph.
def. A digraph $G_2(V, E)$ when V is the set of vertex $V = \{v_1, v_2, \ldots, v_n\}$ and E is the set of directed edges on every $E = \{(v_i, v_j) such that v_i, v_j \in V\}$

Note

For the purpose of this chapter, we assume that V, the set of nodes, is nonempty & finite and that there no self loops or multiple edges in a graph or digraph.

Question

1. which route is cheapest?
2. which route is fastest?
3. if a node or a computer goes down in a computer network, does it get disconected?
4. is there a loop in a flowchart?

3.1 Subgraph

$V' \subseteq V, E' \subseteq E$
Induced Subgraph by V'
$G' = (V', E'), E' = \{u, v | u, v \in V'\}$

Complete graph
$E = \{\{v_i, v_j | 1 \leq j \leq n\}$ eg. if (v, w) is an edge then v & w are adjacent to each other and v and w are said to be incident with the edge (v, w).

def A path from v to w is a sequencs of vertex $v_0, V_1, v_2, \ldots,$ such that $v_0 = v, v_k = w$& $\{V_i, v_j\} \in E$.
(Books defenition is not good)
If $v_0 = v, v_k = w$& all $v_0, v_1, \ldots, v_k$ are distinct, then the length of the path is k. v_0 is a path of length 0.

Connected Graph:

Cycle is path $v_0v_1v_2 \ldots v_k$ such that $v_0 = v_k$
A graph without any cycle is called acylic.

Tree: Acyclic connected graph

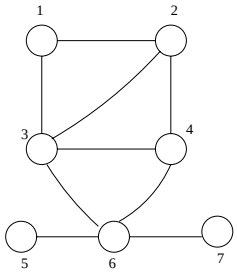
Rooted tree has a designated root vertex establishing parent & child relationship
number of edge in a tree is $n - 1$
Could be proved by induction
A connected component of a graph G is a maximal connected subgraph of G

Weighted Graph
$G = (V, E, w)$
$W : E \leftarrow 2+$
$w(e)$ weight of e (Capacity or distance)

3.2 Representation of a graph

3.2.1 Adjacency Matrix

$A = (a_{ij})_{n*n}$
$a_{ij} = 1$ if $v_i, v_j \in E$
0 else
for $1 \leq i, j \leq n$

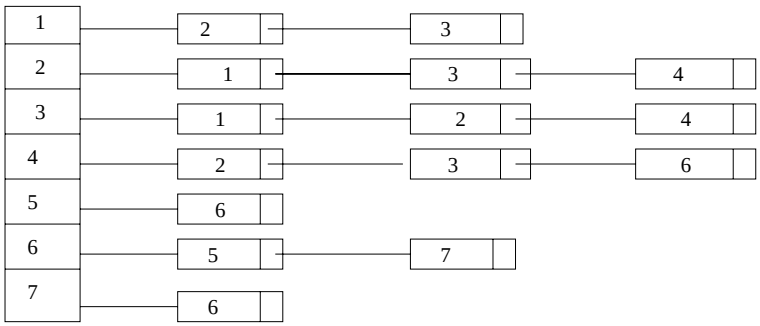


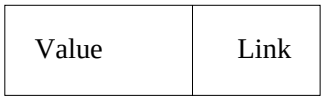
	1	2	3	4	5	6	7
1	0	1	0	1	0	0	0
2	1	0	1	1	0	0	0
3	1	1	0	1	0	0	0
4	0	1	1	0	0	1	0
5	0	0	0	0	0	1	0
6	0	0	1	1	0	1	1
7	0	0	0	0	0	1	0

Space usage $\Leftarrow O(n^2)$

3.2.2 Adjacency list

On any $A[1 \ldots n]$ of linked list, $A[j]$ is the linked list of all the vertices adjacent to v_j.



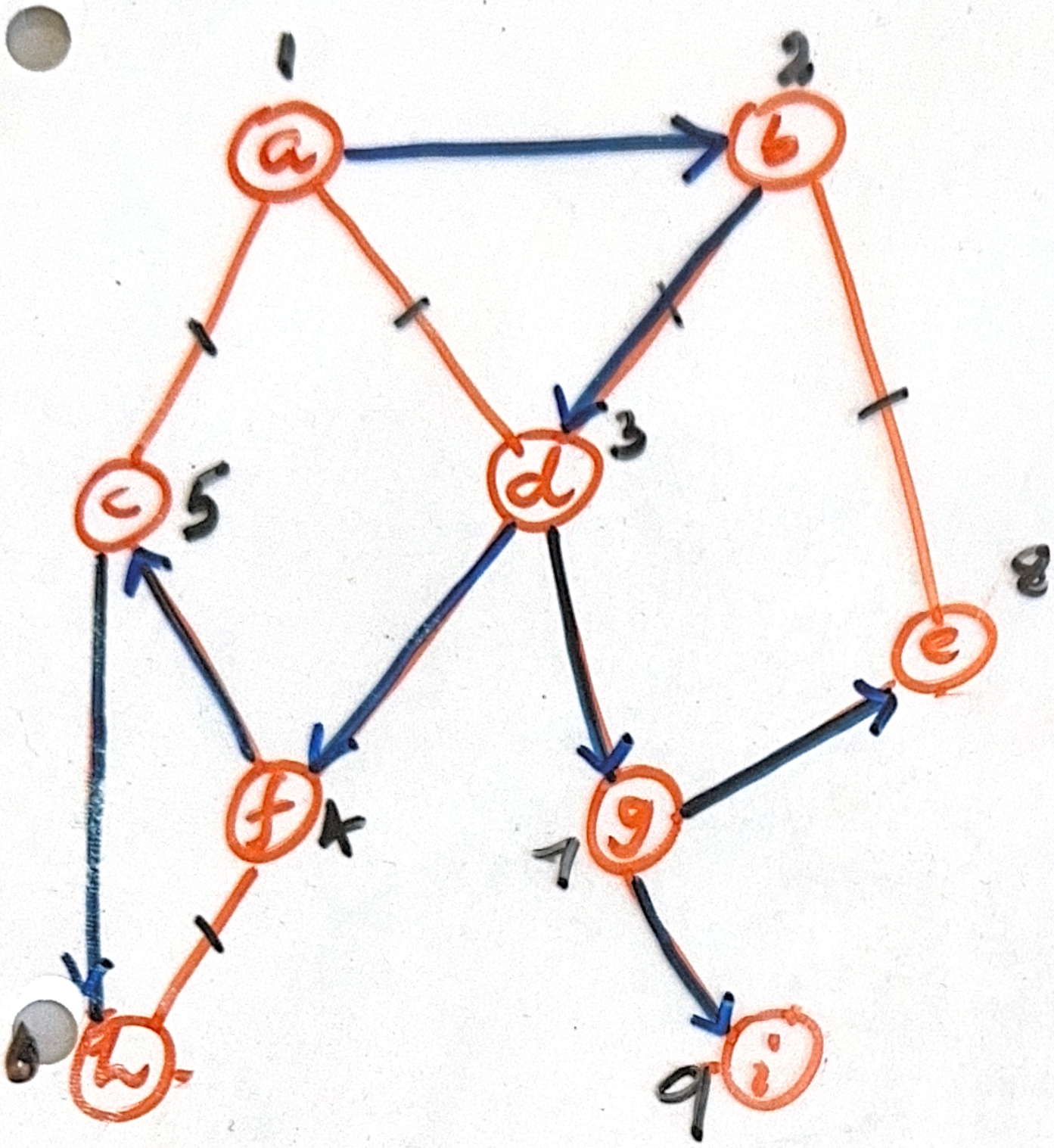


Space usage $\Leftarrow O(n) + O(m) = O(m + n)$
$= O(n^2)$ if m is $O(n^2)$

3.3 Weighted Graph Represatation

Adjacency Matrix $a_{ij} = W(v_i, v_j)$ if $\{v_i v_j\} \in E$
$= 0$ otherwise
Adjacent link

Traversing Graphs



→ Tree Edge

⊣ Back Edge

DFS (v) : Depth-First Search

mark v

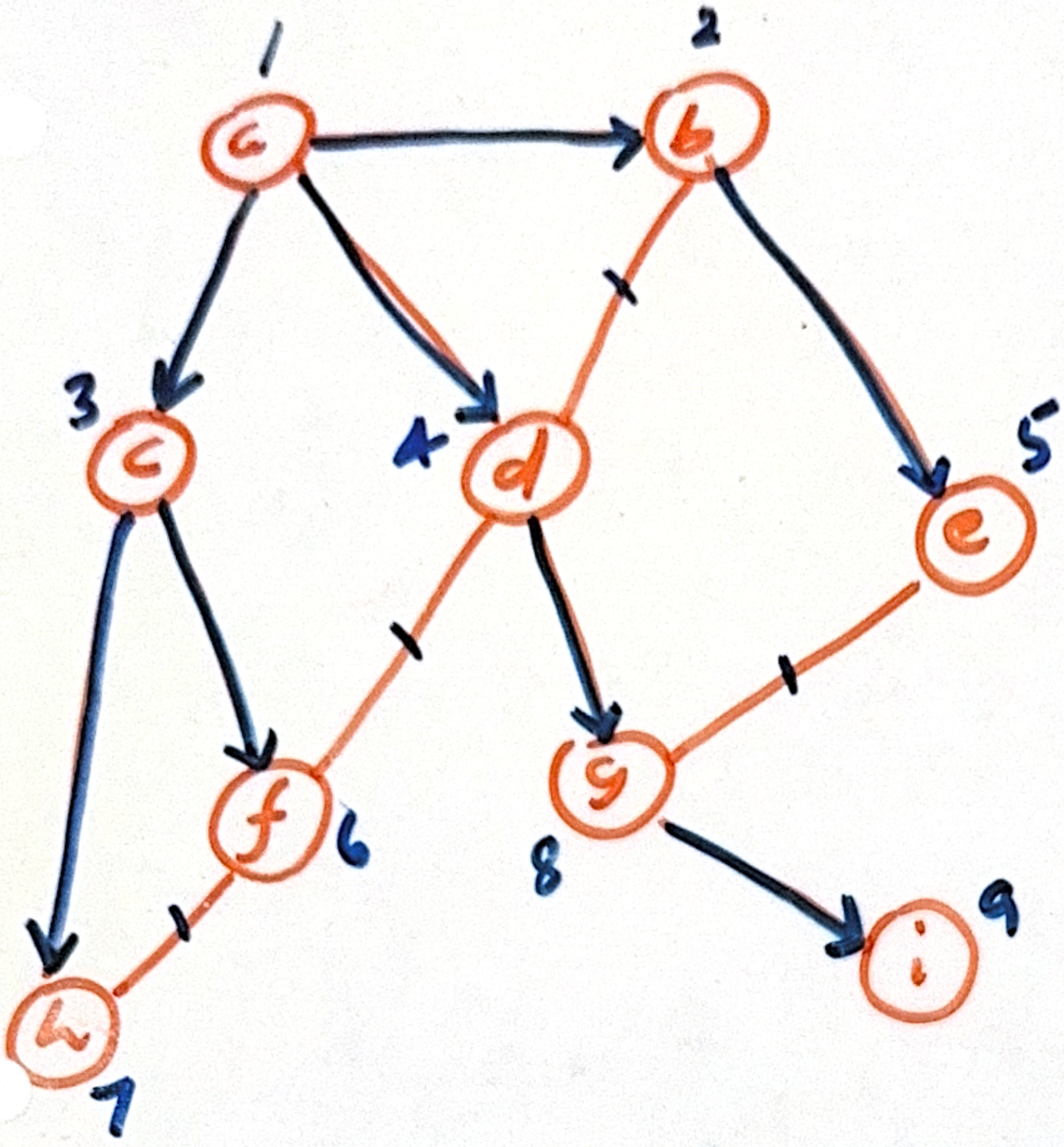
while there is an unmarked node

w adjacent to node v

DFS(w)

end

[Connected Components]



$O(m+n)$

+ cross edges

BFS (v) : Breadth-First SEARCH

mark v

Q = Qv

Q

v

while Q is nonempty do

x = deleteFront (Q)

x

for each unmarked node

w adjacent to x do

{ mark w

Q = Qw

end

4 TRAVERSING GRAPHS

DFS(v): Depth-First Search
mark v
while there is an unmarked node w adjacent to node v DFS(w)
end

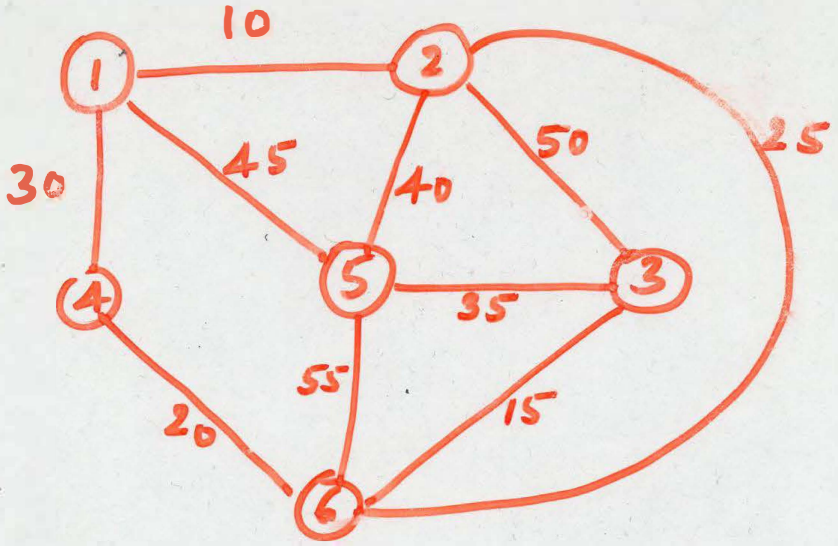
connected components
$O(m+n)$

BFS(v): Breadth-First Search
mark v
Q=Qv
while Q is nonempty do

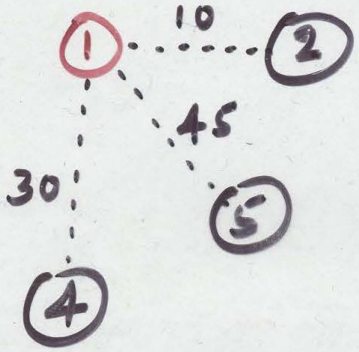
x = delete Front(Q)

for each unmarked node w adjacent to x do mark w
Q=Qw

PRIM-DIJKSTRA'S MST ALGORITHM



PARTIAL MST

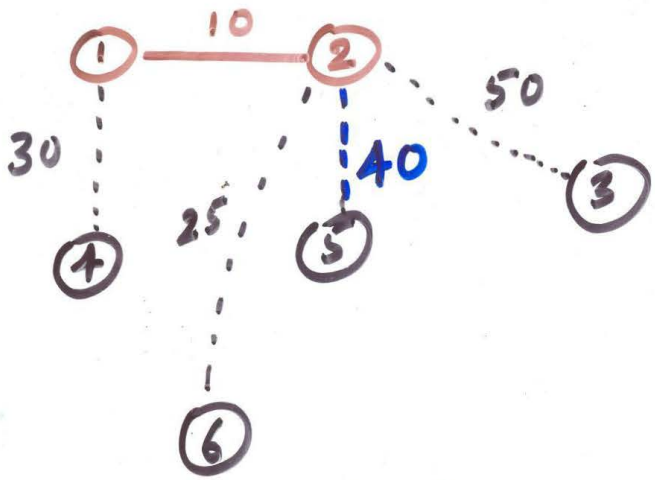


Fringe Vertices

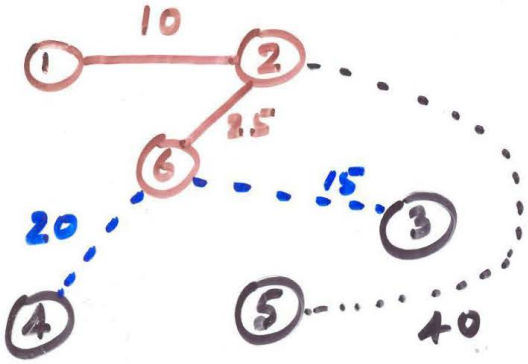
3 unseen vertices

6

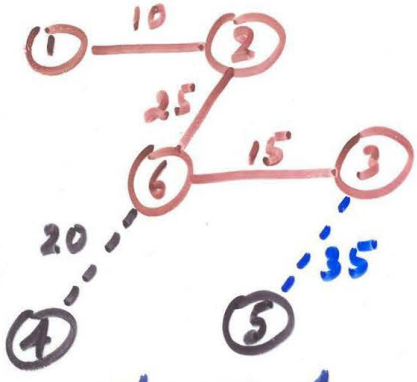
Choose node 2



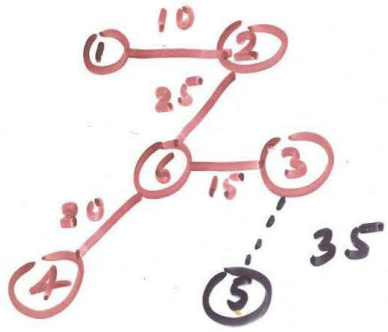
choose node 6



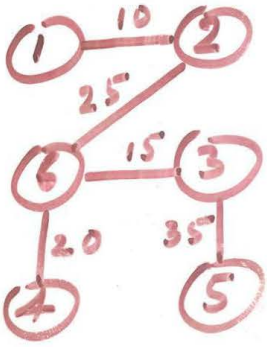
choose node 3



choose 4



choose 5



MST
W = 105

PRIM-DIJKSTRA MST ALGORITHM

Input: $G = (V, E, w)$

Output: $T = (V_T, E_T)$

$E_T = \emptyset$

Select a vertex $v \in V$ and move v to V_T: $V_T = \{v\}$, $V = V - \{v\}$

For $i := 1$ to $n-1$ do

$\Theta(n^2)$ Let $\{v, w\}$ be an edge such that $v \in V_T$, $w \in V$, and for all such edges, $\{v, w\}$ has the minimum weight.

$V_T = V_T \cup \{w\}$

$E_T = E_T \cup \{\{v, w\}\}$

$V = V - \{w\}$

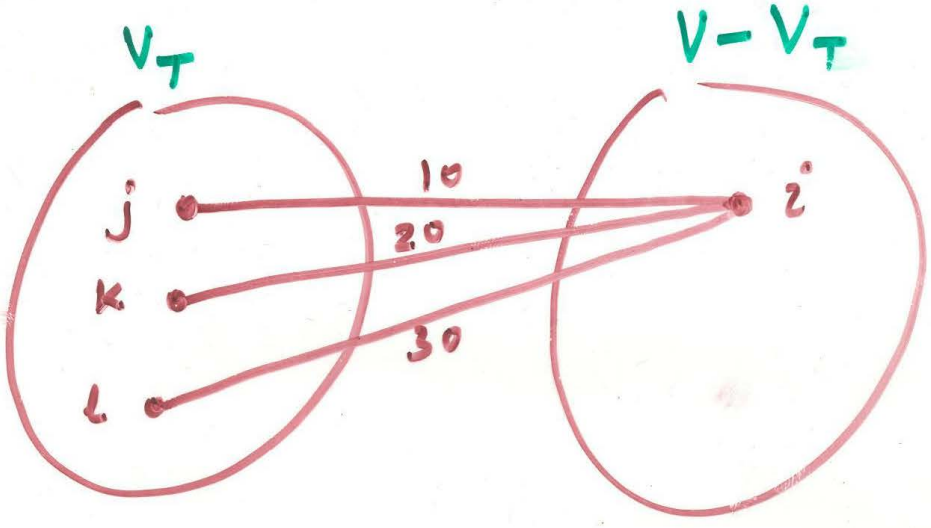
endfor

$W(n) = O(n^3)$

DATA STRUCTURE FOR PRIM'S ALGORITHM

NEAR[1..n]

$$NEAR[i] = \begin{cases} 0 & \text{if } i \in V_T \\ j & \text{if } w(i,j) \text{ is minimum among all } j \in V_T \end{cases}$$



NEAR[i] = j

NEAR[j] = 0

NEAR[l] = 0

MODIFIED PRIM'S ALGORITHM

Input $G = (V, E, w)$, a connected weighted graph

Output $T = (V_T, E_T)$, a MST.

1. $E_T = \emptyset$
2. NEAR[1] = 0 /* $V_T = \{1\}$ */
   NEAR[2..n] = 1 /* $V = V - \{1\}$ */
3. FOR i = 1 to n-1 do

   $O(n)$ $O(\log n)$ FIND j such that NEAR[j] ≠ 0 AND w(j, NEAR[j]) is min.

   NEAR[j] = 0 /* $V_T = V_T \cup \{j\}$ */

   $E_T = E_T \cup \{\{j, NEAR[j]\}\}$.

   /* update NEAR[1..n] */

   for k = 1 to n do

   $O(n \log n)$ DECREASE-KEY if NEAR[k] ≠ 0 AND w(k, NEAR(k)) > w(k, j) then NEAR[k] = j ←

   endfor

endfor.

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

Thm Let $G = (V, E, w)$ be a weighted connected graph and $T = (V, E_T)$ be a MST of G. Let $T' = (V', E')$ is a subtree of T. If $\{x, y\}$ is the minimum weight edge such that

$x \in V'$ and $y \in V - V'$

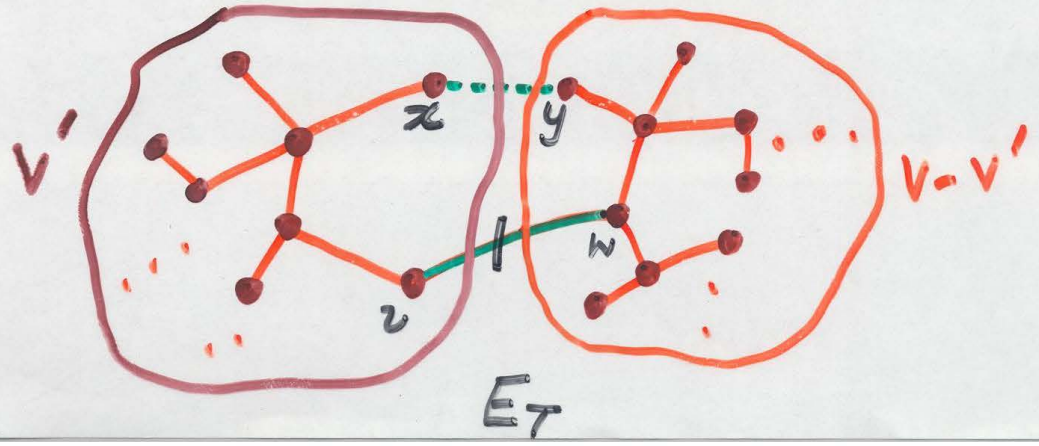
then $T'' = (V' \cup \{y\}, E' \cup \{\{x, y\}\})$

is a subtree of a MST of G.

Proof

1. If $\{x, y\} \in E_T$, done.
2. Let $\{x, y\} \notin E_T$.

   Then $E_T \cup \{\{x, y\}\}$ has a cycle:



By the choice of $\{x, y\}$

$$w(\{x, y\}) \leq w(\{v, w\})$$

Consider $E_T \cup \{\{x, y\}\} - \{\{v, w\}\}$

Its weight is no more than the weight of T and it is a spanning tree.

$\Rightarrow E' \cup \{\{x, y\}\}$ is a subtree of a MST of G.

□

Kruskal's Algorithm

input: $G = (V, E, W)$, a connected graph

output: $T = (V, E_T)$, a MST of G.

$T \leftarrow \emptyset$

while $|T| < n-1$ do

Let $\{v, w\}$ be the least cost edge in E.

$E \leftarrow E - \{\{v, w\}\}$

if $\{v, w\}$ does not create a cycle in T

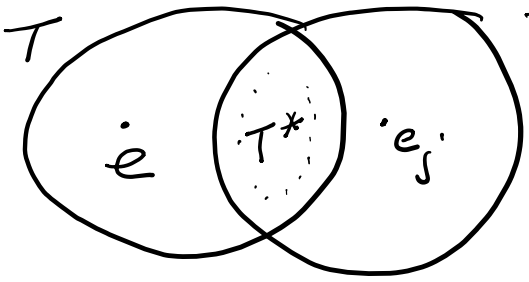
then add $\{v, w\}$ to T

end while

Kruskal's MST Algorithm
Proof of Correctness

Th: Let T be the spanning tree for G generated by kruskal's algorithm. Let T' be a minimum cost spanning tree for G. Show that $W(T) \leq W(T')$. and T is a MST.

Proof:



Let $e \in T - T'$ of min weight.

$\{e\} \cup T'$ has a cycle.

Let $e_j \in T' - T$ in this cycle.

$W(e_j) \geq W(e)$ else $e_j \in T$ by kruskal's

{Why? e_j would have been considered before e for inclusion into a subset $T^* \subseteq T \cap T'$

$\Rightarrow \{e_j\} \cup T^*$ would not form cycle because $T^* \subset T'$.

$\Rightarrow W(T') \geq W(\underbrace{T' - \{e_j\} \cup \{e\}}_{T''}) \geq W(T)$

$\Rightarrow T$ is a MST. $\square$

Repeat for each $e \in T - T'$ $\Rightarrow$ T' becomes T

Kruskal's Algorithm (Refined)

OPERATIONS

1. UNION(i,j), of sets i & j, contains elements of sets i and j.
2. FIND(v) = i iff v ∈ set i.

a) Construct a min-heap E of edges in E.

b) Each $v \in V$ forms singleton set by itself, such that FIND(v) = v.

c) $E_T = \phi$ {Tree is empty}

d) while $|E_T| < n-1$ do

log m Delete the root edge {v,w} from min-heap E and restore heap E.

O(1) O(log n) if FIND(v) ≠ FIND(w)
{$E_T \cup \{\{v,w\}\}$ has no cycle}

then

O(1) $E_T = E_T \cup \{\{v,w\}\}$

/* now, combine the components of E_T joined by {v,w} into one */

O(1) UNION (FIND(v), FIND(w))

end if

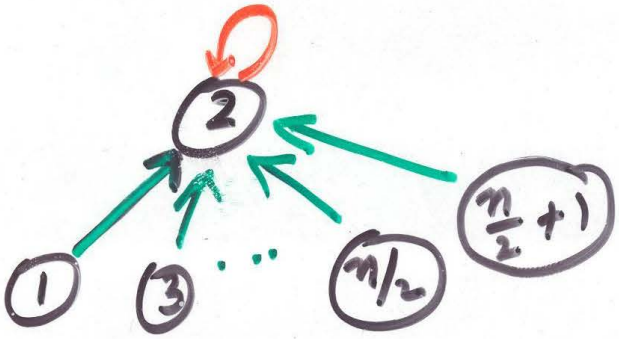
end while O(m log m)

1 2 ... n

1. UNION (1, 2)
2. UNION (2, 3)
⋮
n/2. UNION (n/2, n/2+1)

n/2+1 FIND (1)
FIND (1)
⋮
n FIND (1)

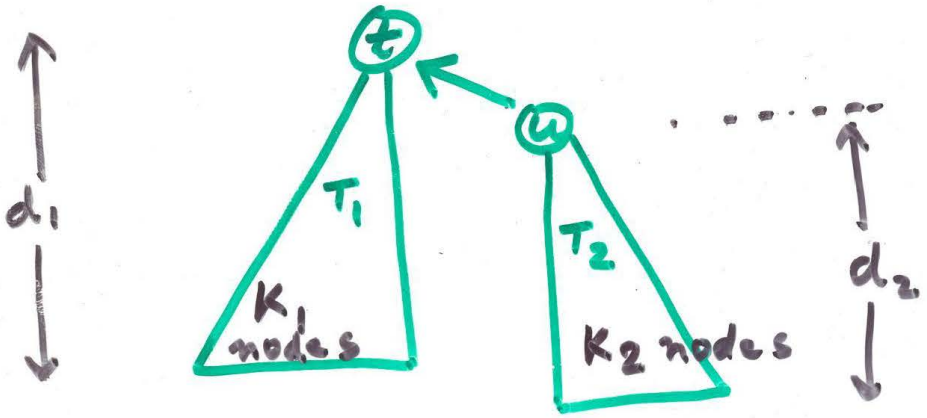
TREE WITH UNWEIGTED UNION



TREE WITH WEIGTED UNION

Lemma With W-UNION, ANY TREE THAT HAS K NODES HAS DEPTH AT MOST $\lfloor \lg K \rfloor$.

W-UNION (t, u)



Let $k_1 \geq k_2$, $K = k_1 + k_2$

$d = d_1$ if $d_1 > d_2$

$= d_2 + 1$ if $d_1 \leq d_2$

BASIS: $K = 1$ $\lfloor \lg 1 \rfloor = 0$

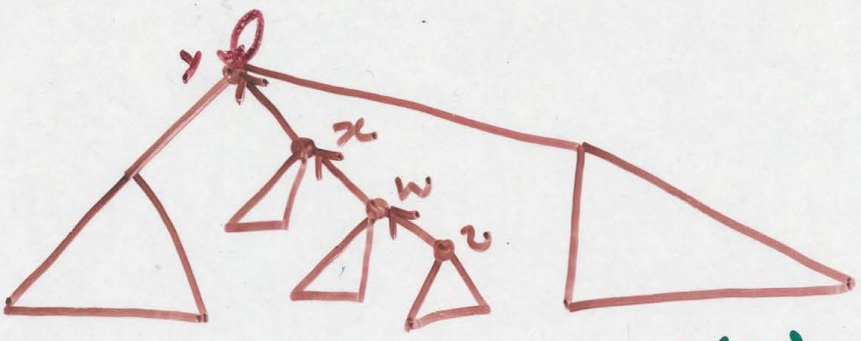
HYPOTHESIS: for $k' < k$, depth $\leq \lfloor \lg k' \rfloor$

INDUCTION

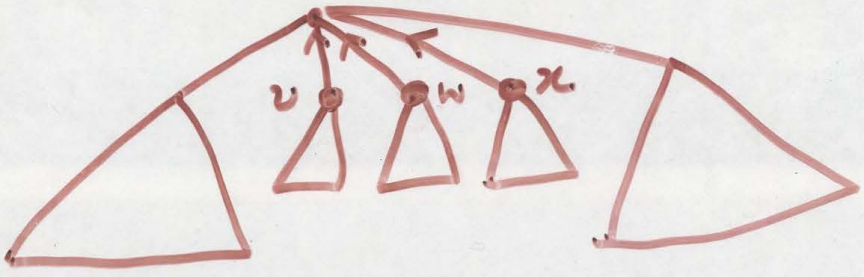
I. $d = d_1 \leq \lfloor \lg k_1 \rfloor \leq \lfloor \lg (k_1 + k_2) \rfloor = \lfloor \lg K \rfloor$

II. $d = d_2 + 1 \leq \lfloor \lg k_2 \rfloor + 1 = \lfloor \lg 2k_2 \rfloor \leq \lfloor \lg (k_1 + k_2) \rfloor$

Lemma A UNION-FIND PROGRAM OF SIZE n DOES $\Theta(n \lg n)$ link OPERATIONS IN THE WORST CASE IF THE WEIGTED UNION IS USED.



BEFORE C-FIND(v)
COMPRESSING-FIND(v)



AFTER C-FIND(v)

Lemma THE NUMBER OF LINK OPERATIONS DONE BY A UNION-FIND PROGRAM OF LENGTH n IMPLEMENTED WITH W-UNION AND C-FIND IS $O(n G(n))$.

Tarjan & Hopcroft

$G(n) = \log^* n$

$= \text{SMALLEST } i \text{ SUCH THAT}$

$\log^{(i)} n \leq 1$

where $\log^{(i)} n = \log(\log^{(i-1)} n)$

and $\log^{(0)} n = n$

$G(65536) = \log^*(2^{16}) = 4$

$\log 2^{16} = 16, \quad \log 16 = 4, \quad \log 4 = 2, \quad \log 2 = 1$

$G(2^{65536}) = 5. \Rightarrow G(n) \leq 5$ for all reasonable n.

[pp. 498-508 ch 22]

COMPARISON

m	$O(n)$	$O(\frac{n^2}{\log n})$	$O(n^2$
Kruskal's	$O(n \log n)$	$O(n^2)$	$O(n^2 \lg n)$
Prim's	$O(n^2)$	$O(n^2)$	$O(n^2$

Kruskal's
$O(m \log m)$
$= O(m \log n)$

PRIM's
$O(n^2)$

Time Complexity of Prim's

- with heap $O(n \log n + m \log n)$
- with Fibonacci Heap
  $O(n \log n + m)$

(MATROID)

STRATEGY GREEDY

```
Solution ← ∅
for i ← 1 to n do
    SELECT the next input x.
    If {x} ∪ Solution is
    FEASIBLE then
        Solution ← COMBINE (Solution, x)
```

Select appropriately finds the next input to be considered.

A feasible solution satisfies the constraints required for the output.

Combine enlarges the current solution to include a new input.

Strategy Divide-and-Conquer.

Divide problem P into smaller problems $P_1, P_2, \ldots, P_k$.

Solve problems $P_1, \ldots, P_k$ to obtain solutions $S_1, S_2, \ldots, S_k$

Combine solution $S_1, S_2, \ldots, S_k$ to get the final solution.

Subproblems $P_1, P_2, \ldots, P_k$ are solved **recursively** using divide-and-conquer.

2 STRATEGY DIVIDE-AND-CONQUER

- Divide Problem P into smaller problem $P_1, P_2, \ldots, P_k$.
- Solve problems $P_1, P_2, \ldots, P_k$ to obtain solutions $S_1, S_2, \ldots, S_k$
- Combine solution $S_1, S_2, \ldots, S_k$ to get the final solution.

Subproblems $P_1, P_2, \ldots, P_k$ are solved recursively using divide-and-conquer.

Examples: Quicksort and mergesort.

3 STRATEGY GREEDY

Solution $\leftarrow \Phi$

for $i \leftarrow 1$ to n do

SELECT the next input x.
If $\{x\} \cup$ Solution is **FEASIBLE** then
solution $\leftarrow$ **COMBINE**(Solution, x)

- **SELECT** appropriately finds the next input to be considered.
- A **FEASIBLE** solution satisfies the constraints required for the output.
- **COMBINE** enlarges the current solution to include a new input.

Examples: Max finding, Selection Sort, and Kruskal's Smallest Edge First algorithm for Minimum Spanning Tree.

4 STRATEGY DYNAMIC PROGRAMMING

- Fibonacci Numbers:

$$F_n = F_{n-1} + F_{n-2}$$

$F_1 = F_0 = 1$.

— Recursive solution requires exponential time: has **overlapping subproblems.**

— **Bottom-up** iterative solution is linear – **compute once, store, and use many times.**

4.1 Matrix Sequence Multiplication

- **eg. 1:**

  $A_{30x1}\ X\ B_{1x40}\ X\ C_{40x10}\ X\ D_{10x25}\ X\ E_{25x1}$

  — Left to right evaluation requires more than 12K multiplications.

  — $(A\ X\ (\ (B\ X\ C)\ X\ (D\ X\ E)\ )\ )$ needs only 690 multiplications (minimum needed).

  — **Greedy Algorithm: Largest Common Dimension First**

- **eg. 2:**

  $A_{1x2}\ X\ B_{2x3}\ X\ C_{3x4}\ X\ D_{4x5}\ X\ E_{5x6}$

  — Largest Common Dimension First imposes following order:

  $(A\ X\ (\ B\ X\ (C\ X\ (D\ X\ E)\ )\ )\ )$

  which needs 240 multiplications.

— Best order:

$$((((A\ X\ B)\ X\ C)\ X\ D)\ X\ E)$$

which needs 68 multiplications.

— **Another Greedy Algorithm: Smallest Common Dimension First** but did not work for eg. 1.

4.2 Divide and Conquer Solution

Input: $\underset{d_0 * d_1}{A_1} * \underset{d_1 * d_2}{A_2} \ldots * \underset{d_{n-1} * d_n}{A_n}$

Output: A paranthesization of the input sequence resulting in minimum number of multiplications needed to multiply the n matrices.

- **Subgoal:** Ignore Structure of Output (order of parenthesization), focus on obtaining a numerical solution (minimum number of multiplications)
- Define **M[i,j]** = the minimum number of multiplications needed to compute

$$A_i * A_{i+1} * \cdots * A_j$$

for $i \leq j \leq n$

- Subgoal is to obtain $M[1, n]$.

e.g. For,

$A1_{30x1}\ X\ A2_{1x40}\ X\ A3_{40x10}\ X\ A4_{10x25}\ X\ A5_{25x1}$

$M[1,1] = 0, M[1,2] = 1200, M[1,5] = 690$

4.3 Recursive Formulation of $M[i, j]$

- A1 X A2 = (A1) X (A2)

  — Partition at k=1: Subproblems (A1) and (A2)

  — cost of (A1) is $M[1, 1]$ and that of (A2) is $M[2, 2]$

  — cost of combining (A1) and (A2) into one is $d_0 * d_1 * d2$.

  — $M[1, 2] = M[1, 1] + M[2, 2] + d_0 * d_1 * d2$.

  — $M[1, 2] = 0 + 0 + 1200 = 1200$.

- A2 X A3 X A4 = (A2) X (A3 X A4) *(k=2)*
  Or, = (A2 X A3) X A4 *(k=3)*.

  — $k = 2$: cost= $M[2, 2] + M[3, 4] + d1 * d2 * d4$

  — $k = 3$: cost= $M[2, 3] + M[4, 4] + d1 * d3 * d4$

  — $M[2, 4] = min(M[2, 2] + M[3, 4] + d1 * d2 * d4, M[2, 3] + M[4, 4] + d1 * d3 * d4)$

In short, $M[2,4] =$
$\min_{2 \leq k \leq 3} (M[2,k] + M[k,4] + d_1 d_k d_4)$

- A2 X A3 X A4 X A5
  = (A2) X (A3 X A4 X A5) *(k=2)*
  Or, = (A2 X A3) X (A4 X A5) *(k=3)*
  Or, = (A2 X A3 X A4) X (A5) *(k=4)*

  $M[2,5] = (M[2,2] + M[3,5] + d_1 d_2 d_5, M[2,3] +$
  $M[4,5] + d_1 d_3 d_5, M[2,4] + M[5,5] + d_1 d_4 d_5)$

- In general, by factoring $(A_i * A_{i+1} * \cdots * A_j)$ at kth index position into $(A_i * A_{i+1} * \cdots * A_k)$ and $(A_{k+1} * \cdots * A_j)$ need $M[i,k] + M[k+1,j]$ multiplications and creates matrices of dimensions $d_{i-1} * d_k$ and $d_k * d_j$. These two matrices need additional $d_{i-1} * d_k * d_j$ multiplications to combine.

4.4 Recursive Formula and Time Taken

- Recursively,
  $M[i, j] =$

$$\min_{i \le k \le j-1} \left(M[i, k] + M[k+1, j] + d_{i-1} d_k d_j \right)$$

$$M(i, i) = 0$$

- **Optimal Substructure**

- We can recursively solve for
  $M[1,n] =$
  $\min_{1 \le k \le n-1} (M[1,k] + M[k+1,n] + d_0 d_k d_n)$
  $= min[M[1,1] + M[2,n] + d_0 d_1 d_n,$
  $M[1,2] + M[3,n] + d_0 d_2 d_n,$
  $M[1,3] + M[4,n] + d_0 d_3 d_n,$
  $\vdots$
  $M[1,n-1] + M[n,n] + d_0 d_{n-1} d_n$

- Time Complexity:

$$\begin{aligned} T_n = n \; & + T_1 + T_{n-1} \\ & + T_2 + T_{n-2} \\ & + T_3 + T_{n-3} \\ & \vdots \\ & + T_{n-2} + T_2 \\ & + T_{n-1} + T_1 \end{aligned}$$

$$T_n = n + 2T_1 + 2T_2 + \cdots + 2T_{n-1} \quad (1)$$
$$T_{n-1} = n - 1 + 2T_1 + 2T_2 + \cdots + 2T_{n-2} \quad (2)$$

Subtracting (I)-(II) yields

$$\begin{aligned} T_n - T_{n-1} &= 1 + 2T_{n-1} \\ T_n &= 1 + 3T_{n-1} \\ &= 1 + 3(1 + 3T_{n-2}) \\ T_n &= 1 + 3 + 3^2 + 3^3 + \cdots + 3^{n-1}T_1 \\ &= 1 + 3 + 3^2 + 3^3 + \cdots + 3^{n-2} \end{aligned}$$

- Recursive Solution is exponential time $\Omega(3^{n-2})$
- Space $O(n)$ stack depth.
- **Overlapping subproblems:** e.g. Recursion tree for $M[1,4]$.
  26 recursive calls for just 10 subproblems
  $M[1,1], M[2,2], M[3,3], M4,4], M[1,2], M[2,3], M$

So we turn to dynamic Programming,

- the same formulation
- approach the problem bottom-to-top
- find a suitable table to store the sub-solutions.

How many sub-solutions do we have?

$$
\begin{array}{ll}
M[1,1], M[2,2], M[3,3] \cdots, M[n,n] & n \\
M[1,2], M[2,3], \cdots, M[n-1,n] & n-1 \\
M[1,3], M[2,4], \cdots, M[n-2,n] & n-2 \\
\vdots & \\
M[1,n-1], M[2,n] & 2 \\
M[1,n] & 1
\end{array}
$$

$$\frac{n(n-1)}{2}$$

$\Rightarrow$ we need $O(n^2)$ space

MATRIX ORDER

```
M, FACTOR : MATRIX

for i ← 1 to n do M[i,i] ← 0
                  {* main diagonal *}
for diagonal ← 1 to n-1 do
    for i ← 1 to n-diagonal do
        j ← i + diagonal
        M[i,j] = min_{i≤k≤j-1} [M[i,k] +
                 M[k+1,j] + d_{i-1} d_k d_j]
        factor[i,j] = k that
            gave the min value for
            M[i,j].
    endfor
endfor
```

$A_1 \times A_2 \times A_3 \times A_4 \times A_5$

$30 \times 1 \quad 1 \times 40 \quad 40 \times 10 \quad 10 \times 25 \quad 25 \times 1$

	1	2	3	4	5
1	0	1200 (1)	700 (1)	1400 (1)	690 (1)
2		0	400 (2)	650 (3)	660 (3)
3			0	10000 (3)	650 (3)
4				0	250 (4)
5					0

$$M[1,2] = \min_{1 \le k \le 1} \left[M[i,k] + M[k+1,j] + d_{i-1} d_k d_j \right]$$

$$= \min \left[M[1,1] + M[2,2] + d_0 d_1 d_2 \right]$$

$$= 0 + 0 + 30 \times 1 \times 40$$

$$= 1200$$

$$M[1,3] = \min_{1 \le k \le 3-1} \left[M[i,k] + M[k+1,j] + d_{i-1} d_k d_j \right]$$

$$= \min \left[M[1,1] + M[2,3] + d_0 d_1 d_3, \; M[1,2] + M[3,3] + d_0 d_2 d_3 \right]$$

$$= \min \left[0 + 400 + 30 \times 1 \times 10, \; 1200 + 0 + 30 \times 40 \times 10 \right]$$

$$= \min [700, 12000 + 1200] = 700$$

4.5 Matrix Parenthesization Order

M,Factor: Matrix

for $i \leftarrow 1$ to n do $M[i,i] \leftarrow 0$
  /* main diagonal*/

for diagonal $\leftarrow 1$ to $n-1$ do

  for $i \leftarrow 1$ to $n-$diagonal do

    $j = i +$ diagonal

    $M[i,j] = \min_{i \leq k \leq j-1}(M[i,k] + M[k+1,j] + d_{i-1} d_k d_j)$

    Factor$[i,j] = k$ that gave the minimum value for $M[i,j]$.

  endfor

endfor

4.6 Work out

$A1_{30x1}\ X\ A2_{1x40}\ X\ A3_{40x10}\ X\ A4_{10x25}\ X\ A5_{25x1}$

$M[1,2] = \min_{1 \le k \le 1}[M[i,k] + M[k+1,j] + d_{i-1}d_k d_j]$

$= min[M[1,1] + M[2,2] + d_0 d_1 d_2]$

$= 0 + 0 + 30 * 1 * 40$

$= 1200$

$M[1,3] = \min_{1 \le k \le 3-1}[M[i,k] + M[k+1,j] + d_{i-1}d_k d_j]$

$= min[M[1,1] + M[2,3] + d_0 d_1 d_3,$

$M[1,2] + M[3,3] + d_0 d_2 d_3]$

$= min[0 + 400 + 30 * 1 * 10,$

$1200 + 0 + 30 * 40 * 10]$

$= min[700, 12000 + 1200]$

$= 700$

$M[2,4] = min[M[i,k] + M[k+1,j] + d_{i-1}d_k d_j]$
$2 \leq k \leq 3$
$= min[M[2,2] + M[3,4] + d_1 d_2 d_4,$
$M[2,3] + M[4,4] + d_1 d_3 d_4]$
$= min[0 + 10000 + 1 * 40 * 25, 400 + 0 + 1 * 10 * 25]$
$= min[10100, 650] = 650$

$M[3,5] = min[M[i,k] + M[k+1,j] + d_{i-1}d_k d_j]$
$3 \leq k \leq 4$
$= min[M[3,3] + M[4,5] + d_2 d_3 d_5,$
$M[3,4] + M[5,5] + d_2 d_4 d_5]$
$= min[0 + 250 + 40 * 10 * 1, 10000 + 0 + 40 * 25 * 1]$
$= min[650, -] = 650$

$M[1,4] = min[M[1,1] + M[2,4] + d_0 d_1 d_4,$
$M[1,2] + M[3,4] + d_0 d_2 d_4,$
$M[1,3] + M[4,4] + d_0 d_3 d_4]$
$= min[0 + 650 + 30 * 1 * 25,$
$1200 + 10000 + 30 * 40 * 25,$
$700 + 0 + 30 * 10 * 25]$
$= min[1400, -, -] = 1400$

5 DYNAMIC PROGRAMMING REQUIREMENTS

Requirements: **a)** Optimal Substructure

b) Overlapping subproblem

Steps: **1)** Characterize the structure of an optimal solution

2) Formulate a recursive solution

3) Compute the value of *an* opt. solution bottom-up. (get value rather than the structure)

4) Construct an optimal solution (structure) from computed infomation.

Memoization: Top-down, compute and store first time, reuse subsequent times.

Advanced Questions on Dynamic Programming and Matrix Chain Multiplication

1. Matrix Chain Multiplication with Restricted Splits

Given a sequence of matrices with dimensions $d_0 \times d_1$, $d_1 \times d_2$, ..., $d_{n-1} \times d_n$, suppose you are only allowed to split the multiplication chain at **even positions** (i.e., valid k for splitting must be even).

- Design a modified dynamic programming algorithm to compute the **minimum number of scalar multiplications**.
- Analyze the time and space complexity.
- Does the optimal substructure property still hold?

Answer:

Problem Statement Recap

Given:

A sequence of matrices $A_1 \times A_2 \times ... \times A_n$, with dimensions $d_0 \times d_1$, $d_1 \times d_2$, ..., $d_{n-1} \times d_n$

You are only allowed to split the multiplication chain at even values of k (i.e., split $A_i..A_j$ at A_k only if k is even and $i \leq k < j$)

✅ Part 1: Standard MCM Recap

Standard recursive relation for Matrix Chain Multiplication:

M[i][j]=mini≤k<j(M[i][k]+M[k+1][j]+d[i−1]∗d[k]∗d[j])M[i][j] = min_{i ≤ k < j} (M[i][k] + M[k+1][j] + d[i-1] * d[k] * d[j]) M[i][j]=mini≤k<j (M[i][k]+M[k+1][j]+d[i−1]∗d[k]∗d[j])

✅ Part 2: Modified DP with Restricted Splits

🎯 Goal:

Only allow even values of k when computing:

M[i][j]=mini≤k<j,keven(M[i][k]+M[k+1][j]+d[i−1]∗d[k]∗d[j])M[i][j] = min_{i ≤ k < j, k even} (M[i][k] + M[k+1][j] + d[i-1] * d[k] * d[j]) M[i][j]=mini≤k<j,keven (M[i][k]+M[k+1][j]+d[i−1]∗d[k]∗d[j])

🧠 Algorithm:

```
for length in range(2, n + 1):  # length of chain
    for i in range(1, n - length + 2):
        j = i + length - 1
        M[i][j] = ∞
        for k in range(i, j):
            if k % 2 == 0:  # only allow even k
                q = M[i][k] + M[k+1][j] + d[i-1]*d[k]*d[j]
                M[i][j] = min(M[i][j], q)
```

🧮 Initialization:

```
for i in range(1, n+1):
    M[i][i] = 0  # cost of multiplying one matrix is zero
```

📦 Space Complexity:

Still uses $O(n^2)$ for storing M[i][j] values

⏱️ Time Complexity:

Outer loop: O(n) (chain lengths)

Middle loop: O(n) (starting indices)

Inner loop: originally O(n) → now O(n/2) due to even k

✅ Total Time Complexity = $O(n^3)$
(Same asymptotic complexity, but 50% fewer computations in inner loop)

✅ Part 3: Optimal Substructure Property

Yes, optimal substructure still holds — but within a restricted solution space.

We still build M[i][j] using optimal solutions of subproblems M[i][k] and M[k+1][j]

However, the solution might not be globally optimal compared to unrestricted DP because some optimal splits (odd k) are not allowed

👉 Substructure is still optimal within the restricted search space

🍼 Example:

Let's say:

dims = [10, 30, 5, 60] → Matrices: A1 (10×30), A2 (30×5), A3 (5×60)

Valid split: Only k = 2 (since it's the only even index in [1,2))

Standard DP would allow:

k = 1, cost = 10×30×5 + 10×5×60 = 1500 + 3000 = 4500

k = 2, cost = 30×5×60 + 10×30×60 = 9000 + 18000 = 27000

🔺 So under restricted DP (only k = 2), it picks suboptimal 27000

✅ Hence:

Optimal substructure is preserved, but

Solution quality can degrade due to restricted choices

2. Fault-Tolerant Matrix Multiplication Order

In a system where any matrix A_i may randomly fail to compute with **5% probability**, propose a modified dynamic programming formulation that **minimizes expected cost** instead of strict scalar multiplications.

- Incorporate failure probabilities into your recurrence relation.
- Discuss trade-offs between accuracy and robustness.

Answer: 🔷 **Problem Summary**

You're given a chain of matrices:

css
CopyEdit
$A_1 \times A_2 \times ... \times A_n$ with dimensions $d_0 \times d_1$, $d_1 \times d_2$, ..., $d_{n-1} \times d_n$

And:

- Each matrix A_i has a **5% probability of failure (i.e., 0.05)**
- The **goal** is to **minimize the expected cost** of multiplication, accounting for potential failures

☑ Step 1: Understanding Failure Impact

In standard Matrix Chain Multiplication:

pgsql
CopyEdit
Cost to compute $A_i..A_j$ = M[i][k] + M[k+1][j] + d[i-1]*d[k]*d[j]

Now, if any matrix within $A_i..A_j$ fails, the **entire computation fails** and needs to be **recomputed**.

So we compute **Expected Cost (E[i][j])** of multiplying matrices $A_i...A_j$

☑ Step 2: Incorporate Probability

Let:

- p = 0.95 (probability that a matrix **does not** fail)
- Then, for A_i to A_j:
  - Probability that **no matrix fails** = p^(j - i + 1)
  - Expected number of trials until success = 1 / p^(j - i + 1)

So the **expected cost** of computing $A_i..A_j$ is:

E[i][j]=(1/p(j−i+1))×(E[i][k]+E[k+1][j]+d[i−1]×d[k]×d[j])E[i][j] = (1 / p^(j - i + 1)) × (E[i][k] + E[k+1][j] + d[i-1] × d[k] × d[j])
E[i][j]=(1/p(j−i+1))×(E[i][k]+E[k+1][j]+d[i−1]×d[k]×d[j])

☑ Step 3: Modified DP Formulation

E[i][j]=minoveri≤k<jof:(1/p(j−i+1))×(E[i][k]+E[k+1][j]+d[i−1]×d[k]×d[j])E[i][j] = min over i ≤ k < j of: (1 / p^(j - i + 1)) × (E[i][k] + E[k+1][j] + d[i-1] × d[k] × d[j])
E[i][j]=minoveri≤k<jof:(1/p(j−i+1))×(E[i][k]+E[k+1][j]+d[i−1]×d[k]×d[j])

☑ Initialization:

E[i][i]=0foralliE[i][i] = 0 for all i E[i][i]=0foralli

☑ Algorithm Structure:

- Outer loop: chain length l = 2 to n
- Middle loop: start index i
- Inner loop: split point k
- Store E[i][j] = expected cost from i to j

⏱ Time & Space Complexity

Metric	Value
Time	$O(n^3)$
Space	$O(n^2)$

3. Mixed Strategy Optimization

You are given a sequence of matrices and two strategies:

- Strategy A: Use a greedy method (e.g., Largest Common Dimension First).
- Strategy B: Use full dynamic programming.

Propose an **adaptive algorithm** that selects between A and B at each subproblem level based on an estimated threshold (e.g., matrix size or dimension ratio).

- Justify when greedy should be favored.
- Compare output to pure DP in practice.

Problem Overview

You're given a sequence of matrices $A_1 \times A_2 \times ... \times A_n$ with dimensions:

less
CopyEdit
A_1: $d_0 \times d_1$, A_2: $d_1 \times d_2$, ..., A_n: $d_{n-1} \times d_n$

You have **two strategies**:

- **Strategy A (Greedy)**: Use a simple rule like "Largest Common Dimension First"
- **Strategy B (DP)**: Use full dynamic programming to get the optimal solution

You are to **adaptively choose** between A and B at each level based on some **heuristics**.

☑ 1. Design of the Adaptive Algorithm

📌 Key Heuristic:

Use **DP** only when the problem size is "large" or has **complex dimension variation**.
Use **Greedy** when subproblem is:

- Small (few matrices)
- Dimensions are "balanced" or similar

☑ Parameters to Consider:

Parameter	Use Greedy When…
length = j - i + 1	≤ Threshold (e.g., 3 or 4 matrices)
dimension_ratio = max(d)/min(d)	≤ RatioThreshold (e.g., ≤ 5)

🧠 Algorithm Sketch:

python
CopyEdit

```
def adaptive_mcm(i, j, dims):
    if j - i + 1 <= GREEDY_THRESHOLD and is_balanced_dims(i, j, dims):
        return greedy_cost(i, j, dims)
    else:
        return dp_cost(i, j, dims)
```

Where:

- is_balanced_dims() checks whether the ratio max(d)/min(d) in subchain is below a threshold
- greedy_cost() applies greedy rule (e.g., merge pair with largest shared dimension)
- dp_cost() applies standard DP recursively

☑️ 2. When is Greedy Preferred?

Greedy works well when:

- All matrices are similar in size
- The common dimensions are consistently large
- You need **fast approximation**, and exact optimality isn't critical

Examples:

python
CopyEdit
A1: 100×100, A2: 100×100, A3: 100×100 → Greedy = DP

Fails when:

- One matrix has drastically different size
- Unbalanced dimensions create opportunities for optimization

☑️ 3. Performance Comparison

🧪 Example Comparison:

Let:

python
CopyEdit
dims = [10, 30, 5, 60]

- Greedy might choose A2 × A3 first → 30×5×60 = 9000
- But optimal (DP): A1 × A2 first → 10×30×5 = 1500, then (A1A2) × A3 = 10×5×60 = 3000 → total = 4500

👉 So in **unbalanced scenarios**, greedy can do **2× worse**.

⏱️ Efficiency Comparison:

Strategy	Time Complexity
DP	$O(n^3)$
Greedy	$O(n^2)$ or better
Mixed Strategy	Adaptive (best of both)

In practice, Mixed Strategy:

- Performs **near-DP quality** with **better average time**
- Can **skip full DP** on easy subproblems

4. Matrix Chain Split Reuse Optimization

You are multiplying a **repeated pattern of matrices**:
e.g., A, B, C, A, B, C, A, B, C (with same dimensions)

- Show how memoization can exploit this structure.
- Design a DP that reduces redundant recomputations by **recognizing identical subchains**.
- Estimate how much time/space is saved compared to standard DP.

Problem:

You're multiplying a repeating matrix pattern like:

```
css
CopyEdit
A, B, C, A, B, C, A, B, C
```

with all A, B, and C having **identical dimensions**.

☑️ Key Idea:

Even though the input length n is large (say 9 matrices), the subchains repeat patterns like ABC, BC, AB, etc.

So instead of treating each M[i][j] independently, we can **hash subchains by pattern**, and **memoize** solutions for subchains with **identical structure**.

🗃️ Optimization:

- Create a **cache**: memo[(pattern)] = cost
- For each subproblem M[i][j], if the **pattern of matrices** $A_i...A_j$ matches a known one (e.g., "ABC"), reuse the result.

🔄 Implementation Strategy:

```
python
CopyEdit
def compute(i, j, matrix_labels):
    key = tuple(matrix_labels[i:j+1])
    if key in memo:
        return memo[key]
    # Otherwise compute normally
    min_cost = inf
    for k in range(i, j):
        cost = compute(i, k) + compute(k+1, j) + cost_of(i, k, j)
        min_cost = min(min_cost, cost)
    memo[key] = min_cost
    return min_cost
```

📉 Time/Space Savings Estimate:

Standard DP:

- Time: $O(n^3)$ for n matrices
- Space: $O(n^2)$

With reuse:

- Time reduced to: $O(u^3)$ where u = #unique patterns (often much smaller than n)
- Huge **speedup** if pattern count is constant (like "ABC" repeating ⇒ just 6 unique chains)

☑ Result:

Memoization of **repeated matrix patterns** yields **dramatic efficiency** for structured input.

5. Space-Optimized Bottom-Up Implementation

Given the classic bottom-up DP for matrix chain multiplication uses $O(n^2)$ space.

- Prove that only a **triangular slice** of the DP matrix is used.
- Design an **in-place optimization** or memory-reduced version of the algorithm using only O(n) space (if possible).
- Discuss the limitations and impact on backtracking the actual parenthesization.

◇ 5. Space-Optimized Bottom-Up Implementation

☑ Observation:

- In bottom-up DP, we compute only the **upper triangle** of the matrix M[i][j], where i < j.

So, only:

python
CopyEdit
j = i + l - 1 (for l = 2 to n)

is relevant.

☑ Can We Reduce to O(n) Space?

We can **reduce from $O(n^2)$ to O(n)** in **special cases**:

📌 *If only the minimum cost value is needed (not the split order), we can:*

- Use 1D array curr[j] and prev[j] for current and previous diagonals
- Only keep O(n) elements at a time

🔧 Limitation:

- If we need to **reconstruct the parenthesization**, we must store **split points (k)** — which needs $O(n^2)$ space.

So:

Goal	Space Usage
Cost only	$O(n)$
Cost + Parentheses Path	$O(n^2)$

6. Randomized Matrix Chain Sampling

Suppose you don't need the exact minimum multiplication cost, but an **expected near-optimal solution**.

- Design a **randomized sampling approach** (Monte Carlo-style) that tries a set of k random parenthesizations.
- Compare its average performance to full DP.
- Suggest use-cases where this is useful (e.g., very large matrix chains).

6. Randomized Matrix Chain Sampling (Monte Carlo)

🧠 Problem:

You don’t need the exact optimal cost — just a **good enough** estimate, **fast**.

Use **random sampling** of parenthesizations.

☑ Algorithm Steps:

1. **Randomly generate k different full parenthesizations**

    a. Use recursive random splitting
2. **Evaluate scalar cost** of each
3. **Return the best (minimum) among them**

python
CopyEdit

```
def sample_cost(i, j, dims):
    if i == j:
        return 0
    k = random.randint(i, j-1)
    return sample_cost(i, k) + sample_cost(k+1, j) + dims[i-1]*dims[k]*dims[j]
```

Repeat this k times and take the best.

📊 Performance Comparison

Metric	Monte Carlo	Full DP
Accuracy	Approximate	Optimal
Time Complexity	O(k * n)	$O(n^3)$
Space	O(1) (if stateless)	$O(n^2)$

☑️ When Monte Carlo is Useful:

- n > 500 → DP becomes infeasible
- Systems with **memory or time constraints**
- Real-time or interactive systems
- Distributed/online algorithms