SEPT 27 — LEC 09

Dynamic Programming — way of improving back tracking
problem w/ backtracking is too much scenarios to look at
- in backtracking we solve a bit of repeated subproblems
- in dynamic programming, don't solve a problem more than once just remember the answer (keep it in a lookup table)

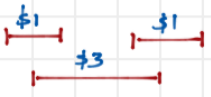
weighted interval scheduling

INPUT: a list of intervals with weights (start, finish, end)
- ex $I = (5, 10, \$7)$

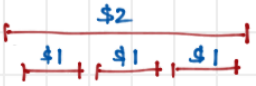
OUTPUT: a scheduling of the intervals which maximize total weight

note that a greedy algorithm would not work

counter example to: finishing time &
max weight per time



counter example to: max weight



IDEA

Given intervals $I_1, \ldots, I_n$

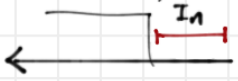
Let M_i = max profit obtainable by scheduling intervals $I_1, \ldots, I_i$

Then the solution to the problem is M_n

Let pred(I_i) = the last interval that finishes before I_i.start

$$m_n = \max \begin{cases} I_n.w + m_{pred(n)} & \leftarrow \text{take } I_n, \text{ then maximize the rest} \longrightarrow \text{ie } I_n \in \text{solution} \\ m_{n-1} & \leftarrow I_n \text{ is not in the solution} \end{cases}$$

Start from I_n so that problem continues as 1 piece not 2 recursive calls



WIS($I_1, \ldots, I_n$) —— with Backtracking

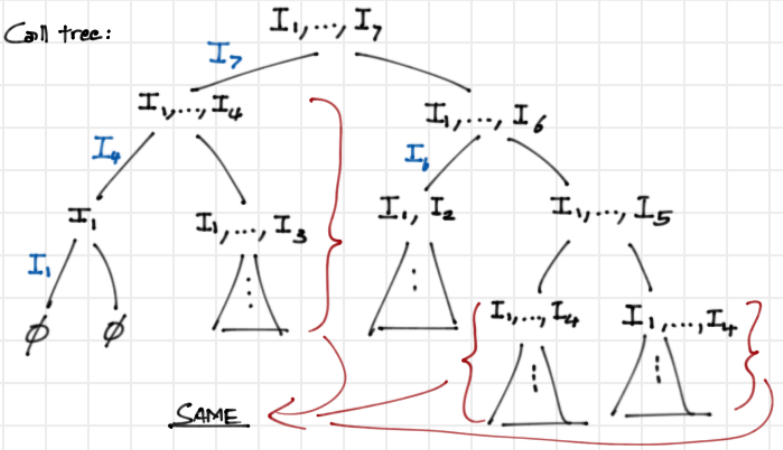
```
p = n
while (I_p.finish > I_n.start)
    p = p - 1
m = WIS(I_1, ..., I_{n-1})
m' = I_n.w + WIS(I_1, ..., I_p)
return max(m, m')
```

running time = $O(n^2)$

b/c may involve alot of repeated calls between calls for each interval

- ex.

Call tree:



WIS Algorithm —— with dynamic programming

$I_1, \ldots, I_n$

$T[1, \ldots, n]$

Sort intervals according to finish time

```
for i from 1 to n, do
    T[i] = -1   <- -1 means not solved yet
return WIS(n)
```

```
WIS(i)
    p = i
    while I_p.finish > I_i.start      } find last interval that does not intersect I_i
        p = p - 1
    if T[i-1] = -1, then
        T[i-1] = WIS(i-1)
    else if T[p] = -1, then
        T[p] = WIS(p)
    return max(T[i-1], T[p] + I_i.w)
```

running time = $O(n \log n) + O(n)$

- $O(n \log n)$: pre compute a pred table
- $O(n)$: WIS algorithm

SEPT 30 — LEC 10

Weighted Interval Scheduling (WIS) —— memonization

INPUT: $\{I_i | (s_i, f_i, w_i)\}$

OUTPUT: a schedule that maximizes the w

APPROACH —— consider the the optimal schedule S, 2 possibilities $I_n \in S$, or $I_n \notin S$; I_n is the last interval

if $I_n \in S$ then the rest of S must have an optimial way to schedule intervals $I_1, \dots, I_k$ where k is the largest index of intervals that does not overlap I_n
i.e $I_{k+1}, I_{k+2}, \dots, I_{n-1}$ all overlap I_n

if $I_n \notin S$ then S must consist of an optimial schedule of $I_1, \dots, I_n$

RECURSIVE SOLUTION

```
for i from 1 to n; do          } O(n)
    T[i] = -1
Sort {I_i} according to their finishing time  O(n lg n)
return WIS(n)  O(n)
```

WIS(n) —— $O(n)\times$ since it is executed atmost once for each $1 \le i \le n$ b/c at most n sub-cases

```
if n < 1; then                          O(1)
    return 0
if T[n-1] == -1; then
    T[n-1] = WIS(n-1)
if T[pred(n)] == -1; then
    T[pred(n)] = WIS(pred(n))
return max(T[n-1], T[pred(n)] + I_n.w)
```

pred(i) = the last interval that does not intersect I_i ——$O(\lg n)$

RUN TIME = $O(n \lg n)$

NOTE: every recursive algorithm has an iterative version (by a stack)

ITERATIVE SOLUTION

```
for i from 1 to n; do ——— O(n) x
    T[i] = -1
Sort {I_i} according to their finishing time
T[0] = 0
for i from 1 to n; do ——— O(n) x
    T[i] = max(T[i-1], T[pred[i]] + I_i.w) —— O(lg n)
```

RUN TIME = $O(n \lg n)$

CORRECTNESS PROOF (itr.)

loop invariant: T contains optimal solution upto i

$P(i) \Rightarrow P(i+1)$

let S be an optimal solution for the first $i+1$ intervals

either I_{i+1} is in S,
in which case $S \in I_{i+1}.w$ + opt for pred[i+1]

or I_{i+1} is not in S,
in which case $S \in$ optimal for i

So $S \le \max(\_, \_) = T[i+1]$

When is more efficient for iterative / recursive?

iterative only solves for values needed for the problem

vs. recursive goes over all values

Dynamic Programming Paradigms

Optimization Problems likely satisfy these properties

subproblem optimality —— an optimal solution to the problem can always be obtained from optimal solutions to its subproblems

✔ Simple subproblems —— subproblems can be characterized precisely using a constant number of parameters (ex. indices)

Subproblem overlap —— Smaller subproblems are repeated many times as a part of the larger problem

Step 0 —— describe the recursive problem. how problem can be decomeposed into simple subproblems and how global optimal solution relates to these subproblems

Step 1 —— Define a table indexed by the parameters that define subproblems to store the optimal value for each subproblem ⟵ one of which is the actual prob.

Step 2 —— Based on the recursive structure of the problem describe a recursive relation satisfied by the table

Step 3 —— Write iterative / recursive algorithm based on the recursive relation

Step 4 —— use the computed table values to find the optimial solution

Dynamic programming

Main components:

1. optimal substructure property
2. DP table - define an entry
3. The recurrence relation
4. Algorithm: Iteration (bottom up)
   Recursive (top-down, memorization)
5. Construct solution

ex. Input - A list of items:

$\{(v_1, w_1), \ldots, (v_n, w_n)\}$ — unlimited supply of each item

- Capacity: W (v: value, w: weight)

Goal: A feasible multiset of items $S = \{i_1, \ldots, i_n\}$ of max value

$$\sum_{i \in S} w_i \leq W$$

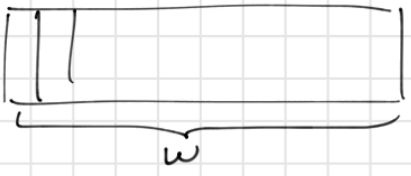
Optimal structure property

Say that $\{i_1, \ldots, i_K\}$ is the optimal solution.

Then $\{i_1, \ldots, i_{K-1}\}$ is opt solution for the knapsack problem w/ capacity $W - w_{i_K}$

Because we could obtain a better solution for the original problem

DP Table:



$K(w)$ = opt value corresponding to w

Recurrance relation

$K(w) = \max_{i \in \{1, \ldots, n\}: w_i \leq w} \{K(w - w_i) + v_i\}$

Base $K(0) = 0$

Algorithm

$K(0) = 0$ #inst

for $w = 1$ to W

$K(w) = \max_{i \in \{1, \ldots, n\}: w_i \leq w} \{K(w - w_i) - v_i\}$

$S(w) = \arg\max_{i \in \{1, \ldots, n\}: w_i \leq w} \{K(w - w_i) + v_i\}$

return $K(W)$ ← returns optimal value

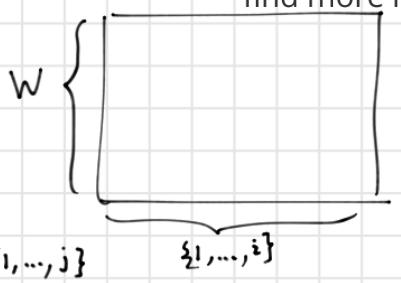
- in order to compute opt. solution
  - to "remember" what was the optimal choice in each step
  - S — solution table

running

- ex. Knapsack (w/o repetitions)

Input $\{(w_1, v_1), \ldots, (w_n, v_n)\}$

W - capacity



objective: find max value (feasible) subset

$K(w,j)$ — value opt. solution of weight $\leq w$, using items $\{1, \ldots, j\}$

recurrance relation

$$K(w,j) = \max\{K(w,j-1),\ \underbrace{K(w-w_j,\ j-1) + v_j}_{\text{if } w_j \leq w}\}$$

algorithm

Init (base)

- $K(w,0) = 0$, $K(0,j) = 0$, $\forall j \in \{0,\ldots,n\}$, $w \in [0, W]$

- for $j = 1$ to n:

  for $w = 1$ to W:

    if $w_j > w$:

      $K(w,j) = K(w,j-1)$

    else

      $K(w,j) = \max\{K(w,j-1),\ K(w-w_j,\ j-1) + v_j\}$

  return $K(W,n)$

Running time: $O(nW)$

The Knapsack problem
Input: We are given a set of n items $I_1, I_2, \cdots, I_n$ and a knapsack with an integer capacity C. Each item I_j has an integer weight w_j and a value v_j.
Output: Identify a subset of items S that (1) maximizes the sum of the values of items in S, and (2) the sum of the weight of items in S is at most C.

Note: This version of the problem is also called $\{0,1\}$-knapsack because each item can be picked at most once (two options for item I_j: (1) $I_j \in S$, and (2) $I_j \notin S$).

Semantic array:
$V[i, c]$ = the maximum value possible using only the first i items and a knapsack of capacity c.

These values can be calculated as follows:

$$V[i,c] = \begin{cases} 0 & \text{if } i = 0 \text{ or } c = 0 \\ \max\{V[i-1,c], V[i-1,c-w_i] + v_i\} & \text{otherwise} \end{cases}$$

Explanation: To find the optimal value $V[i, c]$, either item I_i is needed or it's not needed.

Note: In the calculation of $\max\{V[i-1,c], V[i-1,c-w_i] + v_i\}$, the second part is invoked only if $w_i \leqslant c$.

The solution of this problem is $V[n, C]$. To calculate this value we create a 2 dimensional table V. There are $n+1$ rows and $C+1$ columns. Each entry in this table takes constant time. Thus, the running time is $\mathcal{O}(nC)$.

The last step is to identify the items in the optimal solution. We use V values to decide if we should choose item I_i or not. Try to find out how you can make this decision. It's similar to other DP examples and is easy!

Problem 1: Consider the following "Longest Increasing Sublist" problem.

Input: A list of integers $L = [a_1, a_2, \ldots, a_n]$.

Output: A sublist $L' = [a_{i_1}, a_{i_2}, \ldots, a_{i_k}]$ such that $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$ and $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$ and k is maximum.

For example, if $L = [4, 1, 7, 3, 10, 2, 5, 9]$, then $L_1 = [1, 3, 5, 9]$ and $L_2 = [1, 2, 5, 9]$ are two optimal solutions, but $[1, 2, 3, 4]$ is not a solution (it takes integers from L out of order), $[1, 7, 3, 10]$ is not a solution (it is not increasing), and $[4, 7, 10]$ is not an optimal solution (it is not as long as possible).

Give a dynamic programming algorithm to solve the Longest Increasing Sublist problem.

Step 0: *Describe the recursive structure of sub-problems.*

Any optimal solution for input $[a_1, a_2, \ldots, a_n]$ either contains a_n, or it does not. Consider sub-problems whose solutions have last element a_k, for various values of k.

Step 1: *Define an array ("semantic array") that stores optimal values for arbitrary sub-problems.*

Let $M[k]$ represent the length of a longest increasing sublist that ends with a_k, for $k = 1, 2, \ldots, n$.

Step 2: *Give a recurrence relation for the array values.*

$M[k] = \max\{M[i] + 1 : 0 < i < k \wedge a_i < a_k\}$, for $k = 1, 2, \ldots, n$.

$MAX = \max_{k=1}^{n} M[k]$ is the optimal value.

Step 3: *Write a bottom-up algorithm to compute the array values, following the recurrence.*

```
for k in [1,2,...,n]:
    M[k] := 1
    for i in [1,2,...,k-1]:
        if a_i < a_k and M[i] + 1 > M[k]:
            M[k] := M[i] + 1
```

Step 4: *Use the computed values to reconstruct an optimal solution.*

Use a second array $N[k]$ to store the index of the second-last element in the longest sub-list that ends with a_k.

```
# Complete algorithm.
for k in [1,2,...,n]:
    M[k] := 1
    N[k] := k
    for i in [1,2,...,k-1]:
        if a_i < a_k and M[i] + 1 > M[k]:
            M[k] := M[i] + 1
            N[k] := i

# Figure out the last element in the longest increasing sub-list.
b := 1
for k in [2,3,...,n]:
    if M[k] > M[b]:  b = k
```

```
# Generate the sub-list, working backwards.
S := [b]
while N[b] != b:
    S := N[b] + S
    b := N[b]
```

Problem 2: Consider the problem of creating a weekly schedule of TA office hours. You are given a list of TA's $t_1, t_2, \ldots, t_n$ and a list of time slots $s_1, s_2, \ldots, s_m$ for office hours. Each TA is available for some of the time slots and unavailable for others. Each time slot s_j must be assigned at most one TA, and every week, each TA t_i is responsible for some positive integer number of office hours h_i.
We want to know if there is a feasible schedule of office hours, *i.e.*, if it is possible to assign time slots to TA's to satisfy all of the problem constraints (each TA gets exactly h_i time slots and each time slot gets at most one TA — some time slots may remain unfilled).

(a) Describe precisely how to model this problem as a network flow problem. (Don't forget to specify all edge directions and capacities in your network.)

Solution: Create a network N with

- vertices $V = \{s, s_1, \ldots, s_m, t_1, \ldots, t_n, t\}$,
- edges $E = \{(s, s_i) : 1 \leqslant i \leqslant m\} \cup \{(s_i, t_j) : 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n$, and TA t_j is available at time $s_i\} \cup \{(t_j, t) : 1 \leqslant j \leqslant n\}$, where $c(s, s_i) = 1$ and $c(s_i, t_j) = 1$ and $c(t_j, t) = h_j$ for $1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n$.
  (Note: It is also correct to do this with all edges directed in the opposite direction.)

(b) Explain clearly the correspondence between valid assignments of TAs to office hour time slots and valid integer flows in your network above.

Solution:

- Every valid assignment of TAs to time slots generates a valid flow in N by setting $f(s_i, t_j) = 1$ iff t_j is assigned to s_i, $f(s, s_i) = 1$ iff someone is assigned to time s_i, $f(t_j, t) =$ the number of hours assigned to t_j.
- Every valid integer flow in N corresponds to a valid assignment of TAs to time slots by assigning t_j to s_i for all edges with $f(s_i, t_j) = 1$, because no time can have more than one TA assigned and no TA t_i can be assigned to more than h_i times, by the capacity and conservation constraints.

Problem 3 [If you have time]:
Consider the following "teaching assignment" problem: We are given a set of profs $p_1, ..., p_n$ with teaching loads $L_1, ..., L_n$, and a set of courses $c_1, ..., c_m$ with number of sections $S_1, ..., S_m$, along with subsets of courses that each prof is available to teach. The goal is to assign profs to courses so that: (1) each prof p_i assigned exactly L_i courses, and (2) each course c_j assigned exactly S_j profs.

Show how to represent this problem as a network flow, and how to solve it using network flow algorithms. Justify carefully that your solution is correct and can be obtained in polytime.

Solution: Given input, create network with vertices $p_1, ..., p_n, c_1, ..., c_m$, source s, sink t, and edges (s, p_i) of capacity L_i for each p_i, edges (c_j, t) of capacity S_j for each c_j, edges (p_i, c_j) of capacity 1 for each p_i, c_j such that p_i is available to teach c_j.

- Any assignment of profs to courses yields flow in network: set $f(p_i, c_j) = 1$ if p_i assigned c_j, 0 otherwise; set $f(s, p_i) =$ number of courses assigned to p_i; set $f(c_j, t) =$ number of profs assigned to c_j. Value of this flow

Oct 02 — LEC 11

Single Source Shortest Path (Greedy)

INPUT: connected graph $G = (V, E)$
with none negative weights $w(e)$ for $e \in E$
Source vertex $s \in V$

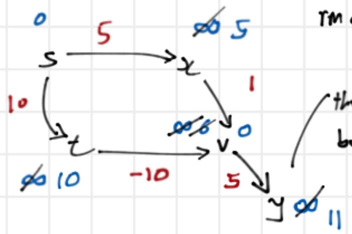
OUTPUT: a path from s to all $v \in V$ with minimum cost (shortest)

DIJKSTRA — modified BFS, store unvisited vertices in a priority queue with key = min distance so far each $v \in V$

```
for i = 1 to n; do                  — O(n) x
    dist[i] = +∞                     } initialization step
    prev [i] = nil
dist[s] = 0
H = BuildHeap(dist)  ← <dist from s, v>

while H is not empty ; do           }
    V = H.ExtractMin()              } O(n+m) x
    for edge (v,u) ∈ E ; do         }
        if dist[u] > dist[v] + w(v,u); then        main loop
            dist[u] = dist[v] + w(v,u)
            prev[u] = v
            H.decreasekey(u)   O(lg n)
```

must not have negative edges; ex.



next min
TM order: s, x, v, t, y

this is actually 5 but is set to 11 b/c visiting order b/c we're not picking the vertex with min dist. each time

running time — $O(n \lg n)$

PROOF

after step i have the shortest distance to all vertices
for step $i+1$ prove still have shortest distance to all vertices

Oct 04 — LEC 12

Single Source Shortest Path (Dynamic Programming)

INPUT: connected graph $G = (V, E)$ with no negative cycles
with weights $w(e)$ for $e \in E$
Source vertex $s \in V$

OUTPUT: a path from s to all $v \in V$ with minimum cost (shortest)

BELLMAN-FORD — restrict number of edges in a path
can also restrict number of vertices in a path

Step 0 — consider the shortest path P from s to v
since G contains no negative cycles, P must be simple (no cycles) $\Rightarrow$ at most $n-1$ edges
let u be the last vertex in P before v
claim the path P' $s \to u$ must also be the shortest path in G, $P' \in P$. Else P is not the shortest path from $s \to v$

Step 1 — define a table
$A_s[v, k]$ = smallest weights of paths from $s \to v$, $v \in V$ with at most k edges, $0 \le k \le n-1$

Step 2 — $A_s[s, 0] = 0$; initially $A_s[v, 0] = \infty$, $v \in V$
then $A_s[v,k] = \min \begin{cases} A_s[v, k-1] \\ A_s[u, k-1] + weight(u,v), & (u,v) \in E \end{cases}$

Step 3 — compute values to A

```
for v ∈ V; do                          — O(n) x
    As[v,0] = ∞,   prev[v] = nil
As[s,0] = 0
for k = 1 to n-1; do                   — O(n) x
    for v ∈ V; do                      }
        As[v,k] = As[v, k-1]           } O(n+m) x
        for edge (u, v) in E; do       }
            if As[u, k-1] + w(u,v) < As[v, k]; then
                As[v,k] = As[u, k-1] + w(u,v)
                prev[v] = S                          O(n^2 + m)
```

Step 4 — find optimal answer. work back from the prev table

```
Path(s,v):
    if v==s; then
        return [ ]
    return [Path(s, prev[v]) , (prev[v], v)]
```

Oct 07 Lec 13

Bellman-Ford by restricting number of vertices

Let $s \to t$ be the shortest path

$D_s[t,k]$ = the shortest path from s to t where the vertices in the path except first and last are vertices with # $1, 2, \ldots, k$

Structure of the recurance:

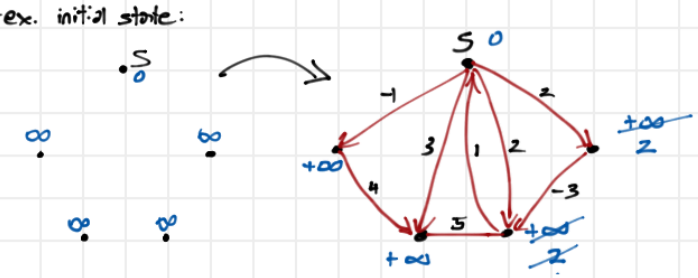
$$D_s[t,k] = \begin{cases} D_s[t,k-1] & \text{—— don't use vertex } k \\ D_s[k,k-1] + D_k[t,k-1] & \text{—— use vertex } k \end{cases}$$

Algorithm

```
for v∈V; do                                   } O(n²)
    for i = 1 to n; do
        D_s[v,i] = ∞
D_s[s,0] = 0
for i = 1 to n-1; do        ← considers all possible vertices from s→v   —— O(n) ×
    for edge (u,v)∈E; do                                                  —— O(m) ×
        if D_s[v,i] > D_s[u,i-1] + weight(u,v); then
            D_s[v,i] = D_s[u,i-1] + weight(u,v)
```

Running time = $O(n^2 + nm)$

- ex. initial state:

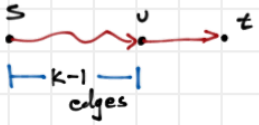


Comparsion with an edge restricting implementation:

$D_s[t,k]$ = the shortest path from s to t with atmost k edges

either the shortest path is of length k or it is of length $k-1$

$$D_s[t,k] = \begin{cases} D_s[t,k-1] \\ D_s[u,k-1] + \text{weight}(u,t) \end{cases}$$



All Pairs Shortest Path

INPUT: A connected graph $G=(V,E)$ with weights $w(e)$ for $e \in E$

OUTPUT: For each $u,v \in V$ a shortest path $u \to v$

IDEA: let $V = \{1, 2, \ldots, n\}$

let $D[u,v,k]$ = the weight of the shortest path from $u \to v$ such that the path only contains vertices $\{1,\ldots,k\}$

Then the initial possible values

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

For each $D[u,v,k]$ there are 2 possible cases:

1. The shortest path does not go through node k
   ⇒ the shortest path is the shortest path using $\{1,\ldots,k-1\}$
   $= D[u,v,k-1]$
2. The shortest path goes through node k.
   ⇒ this path can be segmented into two
   $P_1: s \to k$ using $\{1,\ldots,k-1\}$, $P_2: k \to t$ using $\{1,\ldots,k-1\}$
   the shortest path is the shortest path for P_1 and P_2

⇒ Recurance Structure

$$D[u,v,k] = \begin{cases} D[u,v,k-1] \\ D[u,k,k-1] + D[k,v,k-1] \end{cases}$$

Algorithm — Floyd-Warshall

```
for u = 1 to n; do                                  —— O(n³) ×
    for v = 1 to n; do
        for k = 1 to n; do
            if u == v; then
                D[u,v,k] = 0
            else if (u,v) ∈ E; then
                D[u,v,k] = w(u,v)
            else
                D[u,v,k] = ∞
for u = 1 to n; do                                  —— O(n³) ×
    for v = 1 to n; do
        for k = 1 to n; do
            D[s,t,k] = D[s,t,k-1]
            if D[s,t,k] > D[s,k,k-1] + D[k,t,k-1]; then
                D[s,t,k] = D[s,k,k-1] + D[k,t,k-1]
```

Running time = $O(n^3)$

Chain Matrix Multiplication

Suppose we want to multiply three matrices A, B, C. To calculate the result of this multiplication, we should iteratively multiply two matrices each time. Note that matrix multiplication is not commutative (i.e., $A \times B \neq B \times A$); however, it is associative (i.e., $A \times (B \times C) = (A \times B) \times C$). So to do this multiplication we can parenthesize in different ways (e.g., A(BC) or (AB)C). All will yield same answer but not same running time. Using classical matrix multiplication, multiplying a matrix with dimensionality $p \times q$ and a matrix with dimensionality $q \times r$ costs pqr operations.

Example: Consider three matrices A, B, C with dimensions (respectively) 5×10, 10×100, 100×50. Using $(A \times B) \times C$ costs $5000 + 25000 = 30000$ operations. Using $A \times (B \times C)$ costs $50000 + 2500 = 52500$ operations. As you can see the order of multiplications can significantly change the total run time.

The goal is to identify the optimal order.

Input: There are n matrices $M_1, \cdots, M_n$. Matrix M_i has dimension $d_{i-1} \times d_i$.
Output: Identify the optimal parenthesized product with smallest total cost.

Note: The input in this problem is the dimensions of the matrices not the actual matrix entries.

The optimal parenthesization (the optimal order of pairwise products) can be represented by an optimal parse tree. The leaves are the matrices. Each internal node represents a pairwise matrix multiplication. The root is the last pairwise multiplication. The subtrees are the subproblems that must be computed optimally.

Semantic array:
$\boldsymbol{C[i,j]}$ **= the cost of an optimal product ordering of** $\boldsymbol{M_i \times \cdots \times M_j}$**.**

These values can be calculated as follows:

$$C[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\{C[i,k] + C[k+1,j] + d_{i-1}d_k d_j : i \leqslant k < j\} & \text{if } i < j \end{cases}$$

Identifying the optimal way to parenthesize the product: store the optimal k (the break point) for each $C[i,j]$ in another array $B[i,j]$.

$$B[i,j] = \begin{cases} -1 & \text{if } i = j \\ \arg\min\{C[i,k] + C[k+1,j] + d_{i-1}d_k d_j : i \leqslant k < j\} & \text{if } i < j \end{cases}$$

The following function identifies the best way to parenthesize the matrices.

Function Parenthesize(i, j)

```
1 if i = j then
2 |  print M_i
3 print "(" + Parenthesize(i, B[i, j]) + "×" + (B[i, j] + 1, j) + ")"
```

The optimal way to parenthesize $M_1 M_2 \cdots M_n$ is Parenthesize$(1, n)$.

↳ $O(n^3)$ path to every vertex v from s.

☆ • SHORTEST PATHS

• Floyd: All pairs $\in O(n^3)$

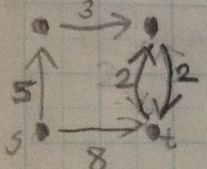
Bellman-Ford: Single Source $\in O(ne+n^2)$ } Dynamic Programming

Dijkstra's: $O(e \lg e + n \lg n) = O(e \lg n + n \lg n)$ Single-Source → Greedy

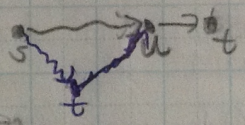
↳ Works w/ - edges

• Find the longest (simple) path from s to t → CANNOT USE DYNAMIC PROG.

No repeated vertices



↳ Break to subproblems:



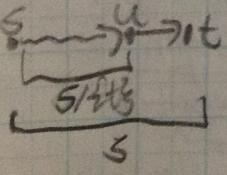
Find the longest (simple) path to u where there is an edge ut.

↳ Diff ij:

* ↳ Q: Is there a (simple) path of length n-1? (Hamiltonian Path) ∈NP-Complete.

↳ Longest (simple) path from s to t using $\{1, \ldots, n\} = S$.

Find the longest (simple) path from s to u using $S \setminus \{t\}$ s.t. there is an edge $ut \in E$.



$\in O(n^3)$ *.

# of multiplication ops := $\ell m n$

$M_1 \quad M_2 \quad M_3$

$d_1 \times d_2 \quad d_2 \times d_3 \quad d_3 \times d_4$

$m_{i,n} =$

$m_{i,n} = \min_{1 \le k \le n-1} \{ m_{i,k} + m_{k+1,n} \}$

$m_{ij}: 1 \le i \le j$

ALGO

for i from 1

for j from

$m_{ij} := +\infty$

if $i = j$ then

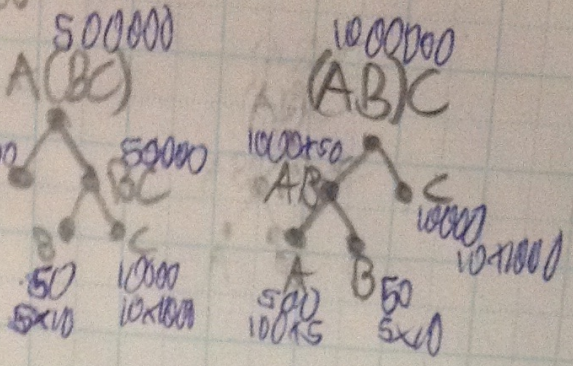
for k from

if

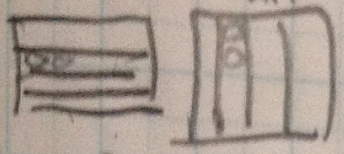
m_{ij}

- $A_{100\times5}$ $B_{5\times10}$ $C_{10\times1000}$

ABC. Can do:



$A_{l\times m}$ $B_{m\times n}$



# of multiplication ops := lmn

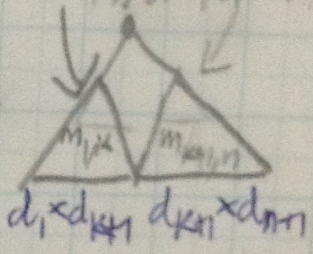
M_1 M_2 M_3 ... M_n

$d_1\times d_2$ $d_2\times d_3$ $d_3\times d_4$ $d_n\times d_{n+1}$

$m_{i,n}$:= min # of multiplications using $M_i, \dots, M_n$

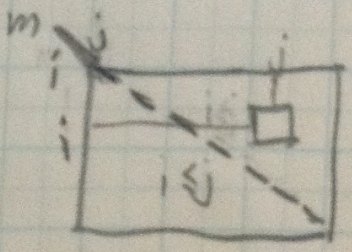
$m_{1,n}$ = ?

At K: $(M_1 \dots M_K)(M_{K+1} \dots M_n)$



$$m_{1,n} = \min_{1\le k\le n-1}\{m_{1,k} + m_{k+1,n} + d_1 d_{k+1} d_{n+1}\}$$

$m_{i,j}$: $1 \le i \le j \le n$



ALGO

```
for i from 1 to n-1 do
    for j from 1 to n do
        m_ij := +∞
        if i = j then m_ij = 0   ## BC
        for k from i to j-1 do
            if m_ij = m_ik + m_(k+1)j + d_i d_(k+1) d_(j+1) < m_ij then
```