CSE 431/531: Algorithm Analysis and Design (Spring 2021)

Dynamic Programming

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo

Paradigms for Designing Algorithms

Greedy algorithm

- Make a greedy choice
- Prove that the greedy choice is safe
- Reduce the problem to a sub-problem and solve it iteratively
- Usually for optimization problems

Divide-and-conquer

- Break a problem into many independent sub-problems
- Solve each sub-problem separately
- Combine solutions for sub-problems to form a solution for the original one
- Usually used to design more efficient algorithms

Paradigms for Designing Algorithms

Dynamic Programming

- Break up a problem into many overlapping sub-problems
- Build solutions for larger and larger sub-problems
- Use a table to store solutions for sub-problems for reuse

Recall: Computing the n-th Fibonacci Number

- $F_0 = 0, F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}, \forall n \geq 2$
- Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \cdots$

Fib(n)

$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2$ to n **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

- Store each $F[i]$ for future use.

Outline

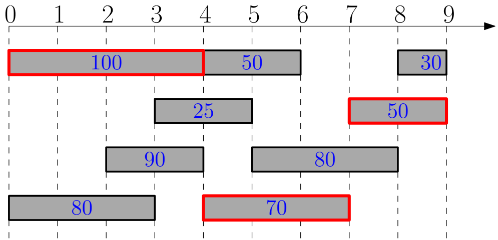
Recall: Interval Schduling

Input: n jobs, job i with start time s_i and finish time f_i

each job has a weight (or value) $v_i > 0$

i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint

Output: a maximum-size subset of mutually compatible jobs



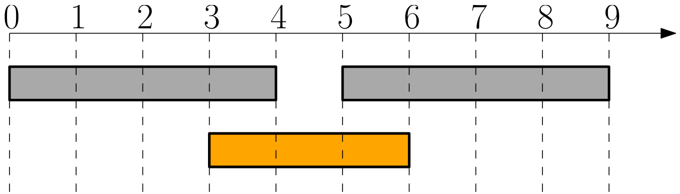
Optimum value $= 220$

Hard to Design a Greedy Algorithm

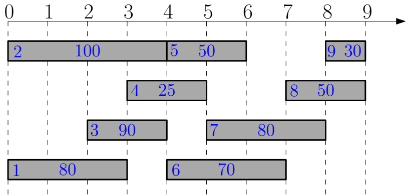
Q: Which job is safe to schedule?

- Job with the earliest finish time? No, we are ignoring weights
- Job with the largest weight? No, we are ignoring times
- Job with the largest $\frac{\text{weight}}{\text{length}}$?

  No, when weights are equal, this is the shortest job



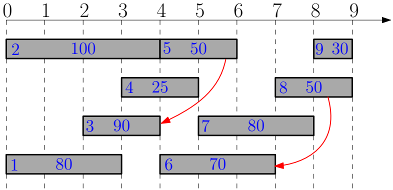
Designing a Dynamic Programming Algorithm



- Sort jobs according to non-decreasing order of finish times
- $opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$

i	$opt[i]$
0	0
1	80
2	100
3	100
4	105
5	150
6	170
7	185
8	220
9	220

Designing a Dynamic Programming Algorithm



- Focus on instance $\{1, 2, 3, \cdots, i\}$,
- $opt[i]$: optimal value for the instance
- assume we have computed $opt[0], opt[1], \cdots, opt[i-1]$

Q: The value of optimal solution that does not contain i?

A: $opt[i-1]$

Q: The value of optimal solution that contains job i?

A: $v_i + opt[p_i]$, p_i = the largest j such that $f_j \le s_i$

Designing a Dynamic Programming Algorithm

Q: The value of optimal solution that does not contain i?

A: $opt[i-1]$

Q: The value of optimal solution that contains job i?

A: $v_i + opt[p_i]$, $p_i =$ the largest j such that $f_j \leq s_i$

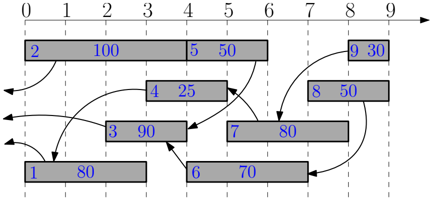
Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

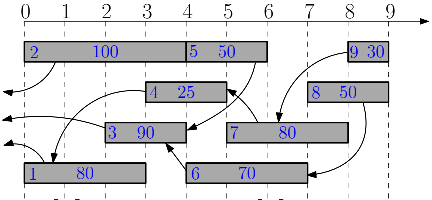


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\} = 100$
- $opt[4] = \max\{opt[3], 25 + opt[1]\} = 105$
- $opt[5] = \max\{opt[4], 50 + opt[3]\} = 150$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$



- $opt[0] = 0, \quad opt[1] = 80, \quad opt[2] = 100$
- $opt[3] = 100, \quad opt[4] = 105, \quad opt[5] = 150$
- $opt[6] = \max\{opt[5], 70 + opt[3]\} = 170$
- $opt[7] = \max\{opt[6], 80 + opt[4]\} = 185$
- $opt[8] = \max\{opt[7], 50 + opt[6]\} = 220$
- $opt[9] = \max\{opt[8], 30 + opt[7]\} = 220$

Dynamic Programming

```
sort jobs by non-decreasing order of finishing times
compute p_1, p_2, ..., p_n
opt[0] ← 0
for i ← 1 to n do
    opt[i] ← max{opt[i-1], v_i + opt[p_i]}
```

- Running time sorting: $O(n \lg n)$
- Running time for computing p: $O(n \lg n)$ via binary search
- Running time for computing $opt[n]$: $O(n)$

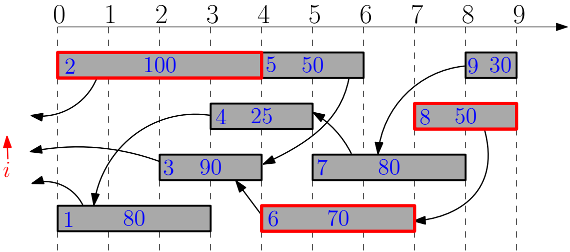
How Can We Recover the Optimum Schedule?

sort jobs by non-decreasing order of finishing times
compute $p_1, p_2, \cdots, p_n$
$opt[0] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  if $opt[i-1] \geq v_i + opt[p_i]$ **then**
    $opt[i] \leftarrow opt[i-1]$
    $b[i] \leftarrow$ N
  else
    $opt[i] \leftarrow v_i + opt[p_i]$
    $b[i] \leftarrow$ Y

$i \leftarrow n, S \leftarrow \emptyset$
while $i \neq 0$ **do**
  if $b[i] =$ N **then**
    $i \leftarrow i - 1$
  else
    $S \leftarrow S \cup \{i\}$
    $i \leftarrow p_i$
return S

Recovering Optimum Schedule: Example

i	$opt[i]$	$b[i]$
0	0	$\perp$
1	80	Y
2	100	Y
3	100	N
4	105	Y
5	150	Y
6	170	Y
7	185	Y
8	220	Y
9	220	N



Dynamic Programming

- Break up a problem into many overlapping sub-problems
- Build solutions for larger and larger sub-problems
- Use a table to store solutions for sub-problems for reuse

Outline

Subset Sum Problem

Input: an integer bound $W > 0$

a set of n items, each with an integer weight $w_i > 0$

Output: a subset S of items that

$$\text{maximizes} \sum_{i \in S} w_i \qquad \text{s.t.} \sum_{i \in S} w_i \leq W.$$

- Motivation: you have budget W, and want to buy a subset of items, so as to spend as much money as possible.

Example:

- $W = 35, n = 5, w = (14, 9, 17, 10, 13)$
- Optimum: $S = \{1, 2, 4\}$ and $14 + 9 + 10 = 33$

Greedy Algorithms for Subset Sum

Candidate Algorithm:

- Sort according to non-increasing order of weights
- Select items in the order as long as the total weight remains below W

Q: Does candidate algorithm always produce optimal solutions?

A: No. $W = 100, n = 3, w = (51, 50, 50)$.

Q: What if we change "non-increasing" to "non-decreasing"?

A: No. $W = 100, n = 3, w = (1, 50, 50)$

Design a Dynamic Programming Algorithm

- Consider the instance: $i, W', (w_1, w_2, \cdots, w_i)$;
- $opt[i, W']$: the optimum value of the instance

Q: The value of the optimum solution that does not contain i?

A: $opt[i-1, W']$

Q: The value of the optimum solution that contains i?

A: $opt[i-1, W'-w_i] + w_i$

Dynamic Programming

- Consider the instance: $i, W', (w_1, w_2, \cdots, w_i)$;
- $opt[i, W']$: the optimum value of the instance

$$opt[i, W'] = \begin{cases} 0 & i = 0 \\ opt[i-1, W'] & i > 0, w_i > W' \\ \max\left\{ \begin{array}{c} opt[i-1, W'] \\ opt[i-1, W'-w_i] + w_i \end{array} \right\} & i > 0, w_i \le W' \end{cases}$$

Dynamic Programming

for $W' \leftarrow 0$ to W **do**
    $opt[0, W'] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    for $W' \leftarrow 0$ to W **do**
        $opt[i, W'] \leftarrow opt[i-1, W']$
        if $w_i \leq W'$ and $opt[i-1, W'-w_i] + w_i \geq opt[i, W']$ **then**
            $opt[i, W'] \leftarrow opt[i-1, W'-w_i] + w_i$
return $opt[n, W]$

Recover the Optimum Set

for $W' \leftarrow 0$ to W **do**

  $opt[0, W'] \leftarrow 0$

for $i \leftarrow 1$ to n **do**

  for $W' \leftarrow 0$ to W **do**

    $opt[i, W'] \leftarrow opt[i-1, W']$

    $b[i, W'] \leftarrow$ N

    if $w_i \leq W'$ and $opt[i-1, W'-w_i] + w_i \geq opt[i, W']$ **then**

      $opt[i, W'] \leftarrow opt[i-1, W'-w_i] + w_i$

      $b[i, W'] \leftarrow$ Y

return $opt[n, W]$

Recover the Optimum Set

$i \leftarrow n, W' \leftarrow W, S \leftarrow \emptyset$
while $i > 0$ **do**
    if $b[i, W'] = \text{Y}$ **then**
        $W' \leftarrow W' - w_i$
        $S \leftarrow S \cup \{i\}$
    $i \leftarrow i - 1$
return S

Running Time of Algorithm

for $W' \leftarrow 0$ to W **do**
    $opt[0, W'] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    for $W' \leftarrow 0$ to W **do**
        $opt[i, W'] \leftarrow opt[i-1, W']$
        if $w_i \leq W'$ and $opt[i-1, W'-w_i] + w_i \geq opt[i, W']$ **then**
            $opt[i, W'] \leftarrow opt[i-1, W'-w_i] + w_i$
return $opt[n, W]$

- Running time is $O(nW)$
- Running time is pseudo-polynomial because it depends on value of the input integers.

Avoiding Unncessary Computation and Memory Using Memoized Algorithm and Hash Map

compute-opt(i, W')

```
if opt[i, W'] ≠ ⊥ then return opt[i, W']
if i = 0 then r ← 0
else
    r ← compute-opt(i − 1, W')
    if w_i ≤ W' then
        r' ← compute-opt(i − 1, W' − w_i) + w_i
        if r' > r then r ← r'
opt[i, W'] ← r
return r
```

- Use hash map for opt

Outline

Knapsack Problem

Input: an integer bound $W > 0$

a set of n items, each with an integer weight $w_i > 0$

a value $v_i > 0$ for each item i

Output: a subset S of items that

$$\text{maximizes } \sum_{i \in S} v_i \qquad \text{s.t.} \sum_{i \in S} w_i \leq W.$$

- Motivation: you have budget W, and want to buy a subset of items of maximum total value

DP for Knapsack Problem

- $opt[i, W']$: the optimum value when budget is W' and items are $\{1, 2, 3, \cdots, i\}$.
- If $i = 0$, $opt[i, W'] = 0$ for every $W' = 0, 1, 2, \cdots, W$.

$$opt[i, W'] = \begin{cases} 0 & i = 0 \\ opt[i-1, W'] & i > 0, w_i > W' \\ \max\begin{Bmatrix} opt[i-1, W'] \\ opt[i-1, W' - w_i] + v_i \end{Bmatrix} & i > 0, w_i \le W' \end{cases}$$

Exercise: Items with 3 Parameters

Input: integer bounds $W > 0, Z > 0$,
a set of n items, each with an integer weight $w_i > 0$
a size $z_i > 0$ for each item i
a value $v_i > 0$ for each item i

Output: a subset S of items that

$$\text{maximizes} \sum_{i \in S} v_i \qquad \text{s.t.}$$

$$\sum_{i \in S} w_i \leq W \text{ and } \sum_{i \in S} z_i \leq Z$$

Outline

Subsequence

- $A = bacdca$
- $C = adca$
- C is a subsequence of A

Def. Given two sequences $A[1..n]$ and $C[1..t]$ of letters, C is called a subsequence of A if there exists integers $1 \leq i_1 < i_2 < i_3 < \ldots < i_t \leq n$ such that $A[i_j] = C[j]$ for every $j = 1, 2, 3, \cdots, t$.

- Exercise: how to check if sequence C is a subsequence of A?

Longest Common Subsequence

Input: $A[1 .. n]$ and $B[1 .. m]$

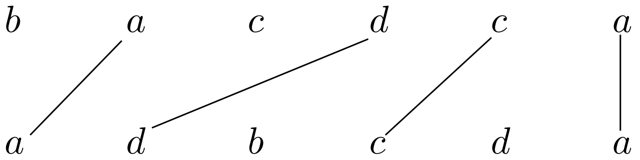
Output: the longest common subsequence of A and B

Example:

- $A =$ '$bacdca$'
- $B =$ '$adbcda$'
- LCS$(A, B) =$ '$adca$'

- Applications: edit distance (diff), similarity of DNAs

Matching View of LCS



- Goal of LCS: find a maximum-size non-crossing matching between letters in A and letters in B.

Reduce to Subproblems

- $A = `bacdca'$
- $B = `adbcda'$
- either the last letter of A is not matched:
- need to compute LCS($`bacd', `adbcd'$)
- or the last letter of B is not matched:
- need to compute LCS($`bacdc', `adbc'$)

Dynamic Programming for LCS

- $opt[i,j], 0 \leq i \leq n, 0 \leq j \leq m$: length of longest common sub-sequence of $A[1 .. i]$ and $B[1 .. j]$.
- if $i = 0$ or $j = 0$, then $opt[i,j] = 0$.
- if $i > 0, j > 0$, then

$$opt[i,j] = \begin{cases} opt[i-1,j-1]+1 & \text{if } A[i] = B[j] \\ \max\begin{cases} opt[i-1,j] \\ opt[i,j-1] \end{cases} & \text{if } A[i] \neq B[j] \end{cases}$$

Dynamic Programming for LCS

for $j \leftarrow 0$ to m **do**
    $opt[0, j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    $opt[i, 0] \leftarrow 0$
    for $j \leftarrow 1$ to m **do**
        if $A[i] = B[j]$ **then**
            $opt[i, j] \leftarrow opt[i-1, j-1] + 1$, $\pi[i, j] \leftarrow$ "$\nwarrow$"
        else if $opt[i, j-1] \geq opt[i-1, j]$ **then**
            $opt[i, j] \leftarrow opt[i, j-1]$, $\pi[i, j] \leftarrow$ "$\leftarrow$"
        else
            $opt[i, j] \leftarrow opt[i-1, j]$, $\pi[i, j] \leftarrow$ "$\uparrow$"

Example

	1	2	3	4	5	6
A	b	a	c	d	c	a
B	a	d	b	c	d	a

	0	1	2	3	4	5	6
0	0 $\perp$	0 $\perp$	0 $\perp$	0 $\perp$	0 $\perp$	0 $\perp$	0 $\perp$
1	0 $\perp$	0 $\leftarrow$	0 $\leftarrow$	1 $\nwarrow$	1 $\leftarrow$	1 $\leftarrow$	1 $\leftarrow$
2	0 $\perp$	1 $\nwarrow$	1 $\leftarrow$	1 $\leftarrow$	1 $\leftarrow$	1 $\leftarrow$	2 $\nwarrow$
3	0 $\perp$	1 $\uparrow$	1 $\leftarrow$	1 $\leftarrow$	2 $\nwarrow$	2 $\leftarrow$	2 $\leftarrow$
4	0 $\perp$	1 $\uparrow$	2 $\nwarrow$	2 $\leftarrow$	2 $\leftarrow$	3 $\nwarrow$	3 $\leftarrow$
5	0 $\perp$	1 $\uparrow$	2 $\uparrow$	2 $\leftarrow$	3 $\nwarrow$	3 $\leftarrow$	3 $\leftarrow$
6	0 $\perp$	1 $\nwarrow$	2 $\uparrow$	2 $\leftarrow$	3 $\uparrow$	3 $\leftarrow$	4 $\nwarrow$

Example: Find Common Subsequence

	1	2	3	4	5	6
A	b	a	c	d	c	a
B	a	d	b	c	d	a

	0	1	2	3	4	5	6
0	0 $\perp$	0 $\perp$	0 $\perp$	0 $\perp$	0 $\perp$	0 $\perp$	0 $\perp$
1	0 $\perp$	0 $\leftarrow$	0 $\leftarrow$	1 $\nwarrow$	1 $\leftarrow$	1 $\leftarrow$	1 $\leftarrow$
2	0 $\perp$	1 $\nwarrow$	1 $\leftarrow$	1 $\leftarrow$	1 $\leftarrow$	1 $\leftarrow$	2 $\nwarrow$
3	0 $\perp$	1 $\uparrow$	1 $\leftarrow$	1 $\leftarrow$	2 $\nwarrow$	2 $\leftarrow$	2 $\leftarrow$
4	0 $\perp$	1 $\uparrow$	2 $\nwarrow$	2 $\leftarrow$	2 $\leftarrow$	3 $\nwarrow$	3 $\leftarrow$
5	0 $\perp$	1 $\uparrow$	2 $\uparrow$	2 $\leftarrow$	3 $\nwarrow$	3 $\leftarrow$	3 $\leftarrow$
6	0 $\perp$	1 $\nwarrow$	2 $\uparrow$	2 $\leftarrow$	3 $\uparrow$	3 $\leftarrow$	4 $\nwarrow$

Find Common Subsequence

```
i ← n, j ← m, S ← ""
while i > 0 and j > 0 do
    if π[i, j] = "↖" then
        S ← A[i] ⋈ S, i ← i − 1, j ← j − 1
    else if π[i, j] = "↑" then
        i ← i − 1
    else
        j ← j − 1
return S
```

Variants of Problem

Edit Distance with Insertions and Deletions

Input: a string A

each time we can delete a letter from A or insert a letter to A

Output: minimum number of operations (insertions or deletions) we need to change A to B?

Example:

- $A =$ ocurrance, $B =$ occurrence
- 3 operations: insert 'c', remove 'a' and insert 'e'

Obs. $\#$OPs $=$ length(A) $+$ length(B) - 2 $\cdot$ length(LCS(A, B))

Variants of Problem

Edit Distance with Insertions, Deletions and Replacing

Input: a string A,

each time we can delete a letter from A, insert a letter to A or change a letter

Output: how many operations do we need to change A to B?

Example:

- $A =$ ocurrance, $B =$ occurrence.
- 2 operations: insert 'c', change 'a' to 'e'

- Not related to LCS any more

Edit Distance (with Replacing)

- $opt[i,j], 0 \leq i \leq n, 0 \leq j \leq m$: edit distance between $A[1 .. i]$ and $B[1 .. j]$.
- if $i = 0$ then $opt[i,j] = j$; if $j = 0$ then $opt[i,j] = i$.
- if $i > 0, j > 0$, then

$$opt[i,j] = \begin{cases} opt[i-1, j-1] & \text{if } A[i] = B[j] \\ \min \begin{cases} opt[i-1, j] + 1 \\ opt[i, j-1] + 1 \\ opt[i-1, j-1] + 1 \end{cases} & \text{if } A[i] \neq B[j] \end{cases}$$

Exercise: Longest Palindrome

Def. A palindrome is a string which reads the same backward or forward.

- example: "racecar", "wasitacaroracatisaw", "putitup"

Longest Palindrome Subsequence

Input: a sequence A

Output: the longest subsequence C of A that is a palindrome.

Example:

- Input: acbcedeacab
- Output: acedeca

Outline

Computing the Length of LCS

```
for j ← 0 to m do
    opt[0, j] ← 0
for i ← 1 to n do
    opt[i, 0] ← 0
    for j ← 1 to m do
        if A[i] = B[j] then
            opt[i, j] ← opt[i − 1, j − 1] + 1
        else if opt[i, j − 1] ≥ opt[i − 1, j] then
            opt[i, j] ← opt[i, j − 1]
        else
            opt[i, j] ← opt[i − 1, j]
```

Obs. The i-th row of table only depends on $(i-1)$-th row.

Reducing Space to $O(n+m)$

Obs. The i-th row of table only depends on $(i-1)$-th row.

Q: How to use this observation to reduce space?

A: We only keep two rows: the $(i-1)$-th row and the i-th row.

Linear Space Algorithm to Compute Length of LCS

```
for j ← 0 to m do
    opt[0, j] ← 0
for i ← 1 to n do
    opt[i mod 2, 0] ← 0
    for j ← 1 to m do
        if A[i] = B[j] then
            opt[i mod 2, j] ← opt[i − 1 mod 2, j − 1] + 1
        else if opt[i mod 2, j − 1] ≥ opt[i − 1 mod 2, j] then
            opt[i mod 2, j] ← opt[i mod 2, j − 1]
        else
            opt[i mod 2, j] ← opt[i − 1 mod 2, j]
return opt[n mod 2, m]
```

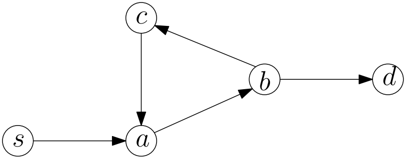
How to Recover LCS Using Linear Space?

- Only keep the last two rows: only know how to match $A[n]$
- Can recover the LCS using n rounds: time $= O(n^2m)$
- Using Divide and Conquer + Dynamic Programming:
  - Space: $O(m+n)$
  - Time: $O(nm)$

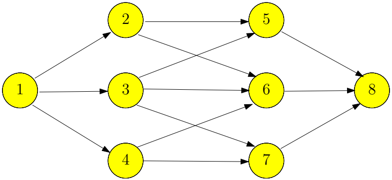
Outline

Directed Acyclic Graphs

Def. A directed acyclic graph (DAG) is a directed graph without (directed) cycles.



not a DAG



a DAG

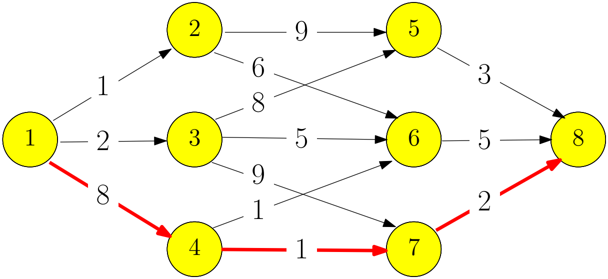
Lemma A directed graph is a DAG if and only its vertices can be topologically sorted.

Shortest Paths in DAG

Input: directed acyclic graph $G = (V, E)$ and $w : E \to \mathbb{R}$.

Assume $V = \{1, 2, 3 \cdots, n\}$ is topologically sorted: if $(i, j) \in E$, then $i < j$

Output: the shortest path from 1 to i, for every $i \in V$



Shortest Paths in DAG

- $f[i]$: length of the shortest path from 1 to i

$$f[i] = \begin{cases} 0 & i = 1 \\ \min_{j:(j,i)\in E} \{f(j) + w(j,i)\} & i = 2, 3, \cdots, n \end{cases}$$

Shortest Paths in DAG

- Use an adjacency list for incoming edges of each vertex i

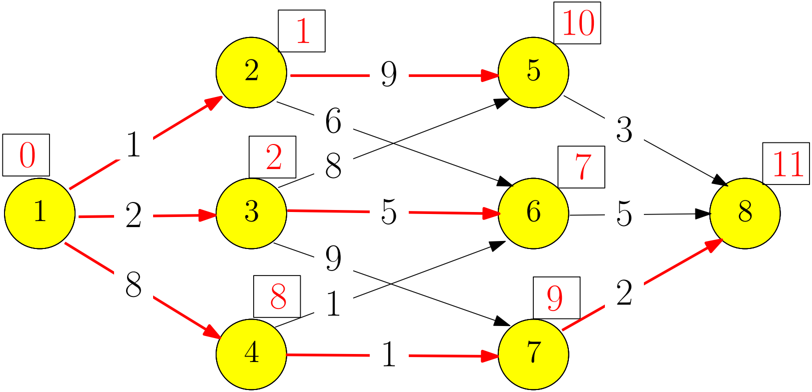
Shortest Paths in DAG

$f[1] \leftarrow 0$
for $i \leftarrow 2$ to n **do**
    $f[i] \leftarrow \infty$
    for each incoming edge (j, i) of i **do**
        if $f[j] + w(j, i) < f[i]$ **then**
            $f[i] \leftarrow f[j] + w(j, i)$
            $\pi(i) \leftarrow j$

print-path(t)

if $t = 1$ **then**
    print(1)
    return
print-path$(\pi(t))$
print(“,”, t)

Example



Variant: Heavest Path in a Directed Acyclic Graph

Heaviest Path in a Directed Acyclic Graph

Input: directed acyclic graph $G = (V, E)$ and $w : E \to \mathbb{R}$.
Assume $V = \{1, 2, 3 \cdots, n\}$ is topologically sorted: if $(i, j) \in E$, then $i < j$

Output: the path with the largest weight (the heaviest path) from 1 to n.

- $f[i]$: weight of the heaviest path from 1 to i

$$f[i] = \begin{cases} 0 & i = 1 \\ \max_{j:(j,i)\in E} \{f(j) + w(j, i)\} & i = 2, 3, \cdots, n \end{cases}$$

Outline

Matrix Chain Multiplication

Matrix Chain Multiplication

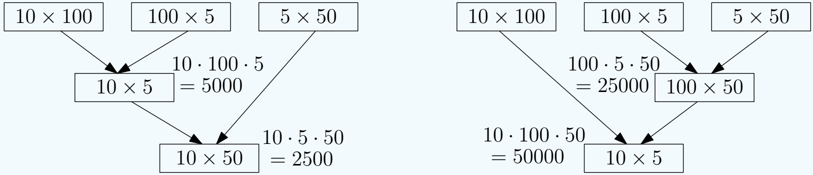
Input: n matrices $A_1, A_2, \cdots, A_n$ of sizes $r_1 \times c_1, r_2 \times c_2, \cdots, r_n \times c_n$, such that $c_i = r_{i+1}$ for every $i = 1, 2, \cdots, n-1$.

Output: the order of computing $A_1 A_2 \cdots A_n$ with the minimum number of multiplications

Fact Multiplying two matrices of size $r \times k$ and $k \times c$ takes $r \times k \times c$ multiplications.

Example:

- $A_1 : 10 \times 100, \quad A_2 : 100 \times 5, \quad A_3 : 5 \times 50$



$\text{cost} = 5000 + 2500 = 7500$ $\qquad$ $\text{cost} = 25000 + 50000 = 75000$

- $(A_1A_2)A_3$: $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$
- $A_1(A_2A_3)$: $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$

Matrix Chain Multiplication: Design DP

- Assume the last step is $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$
- Cost of last step: $r_1 \times c_i \times c_n$
- Optimality for sub-instances: we need to compute $A_1 A_2 \cdots A_i$ and $A_{i+1} A_{i+2} \cdots A_n$ optimally
- $opt[i, j]$: the minimum cost of computing $A_i A_{i+1} \cdots A_j$

$$opt[i, j] = \begin{cases} 0 & i = j \\ \min_{k: i \le k < j} \left(opt[i, k] + opt[k+1, j] + r_i c_k c_j \right) & i < j \end{cases}$$

Matrix Chain Multiplication: Design DP

matrix-chain-multiplication$(n, r[1..n], c[1..n])$

let $opt[i,i] \leftarrow 0$ for every $i = 1, 2, \cdots, n$
for $\ell \leftarrow 2$ to n **do**
    for $i \leftarrow 1$ to $n - \ell + 1$ **do**
        $j \leftarrow i + \ell - 1$
        $opt[i,j] \leftarrow \infty$
        for $k \leftarrow i$ to $j - 1$ **do**
            if $opt[i,k] + opt[k+1,j] + r_i c_k c_j < opt[i,j]$ **then**
                $opt[i,j] \leftarrow opt[i,k] + opt[k+1,j] + r_i c_k c_j$
                $\pi[i,j] \leftarrow k$
return $opt[1,n]$

Constructing Optimal Solution

Print-Optimal-Order(i, j)

if $i = j$ **then**
  print(“A”$_i$)
else
  print(“(”)
  Print-Optimal-Order$(i, \pi[i, j])$
  Print-Optimal-Order$(\pi[i, j] + 1, j)$
  print(“)”)

Outline

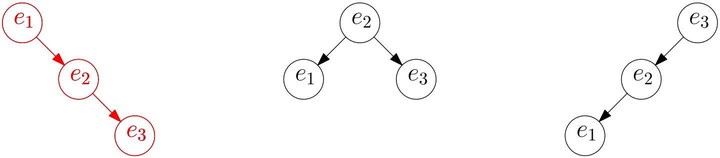
Optimum Binary Search Tree

- n elements $e_1 < e_2 < e_3 < \cdots < e_n$
- e_i has frequency f_i
- goal: build a binary search tree for $\{e_1, e_2, \cdots, e_n\}$ with the minimum accessing cost:

$$\sum_{i=1}^{n} f_i \times (\text{depth of } e_i \text{ in the tree})$$

Optimum Binary Search Tree

- Example: $f_1 = 10, f_2 = 5, f_3 = 3$



- $10 \times 1 + 5 \times 2 + 3 \times 3 = 29$
- $10 \times 2 + 5 \times 1 + 3 \times 2 = 31$
- $10 \times 3 + 5 \times 2 + 3 \times 1 = 43$

- suppose we decided to let e_i be the root
- $e_1, e_2, \cdots, e_{i-1}$ are on left sub-tree
- $e_{i+1}, e_{i+2}, \cdots, e_n$ are on right sub-tree

- d_j: depth of e_j in our tree
- C, C_L, C_R: cost of tree, left sub-tree and right sub-tree respectively

$$\begin{aligned} C &= \sum_{j=1}^{n} f_j d_j = \sum_{j=1}^{n} f_j + \sum_{j=1}^{n} f_j(d_j - 1) \\ &= \sum_{j=1}^{n} f_j + \sum_{j=1}^{i-1} f_j(d_j - 1) + \sum_{j=i+1}^{n} f_j(d_j - 1) \\ &= \sum_{j=1}^{n} f_j + C_L + C_R \end{aligned}$$

$$C = \sum_{j=1}^{n} f_j + C_L + C_R$$

- In order to minimize C, need to minimize C_L and C_R respectively
- $opt_{i,j}$: the optimum cost for the instance $(f_i, f_{i+1}, \cdots, f_j)$
- for every $i \in \{1, 2, \cdots, n, n+1\}$: $opt[i, i-1] = 0$
- for every i, j such that $1 \leq i \leq j \leq n$,

$$opt[i, j] = \sum_{k=i}^{j} f_k + \min_{k: i \leq k \leq j} \big(opt[i, k-1] + opt[k+1, j]\big)$$

Outline

Dynamic Programming

- Break up a problem into many overlapping sub-problems
- Build solutions for larger and larger sub-problems
- Use a table to store solutions for sub-problems for reuse

Comparison with greedy algorithms

- Greedy algorithm: each step is making a small progress towards constructing the solution
- Dynamic programming: the whole solution is constructed in the last step

Comparison with divide and conquer

- Divide and conquer: an instance is broken into many independent sub-instances, which are solved separately.
- Dynamic programming: the sub-instances we constructed are overlapping.

Definition of Cells for Problems We Learnt

- Weighted interval scheduling: $opt[i]$ = value of instance defined by jobs $\{1, 2, \cdots, i\}$
- Subset sum, knapsack: $opt[i, W']$ = value of instance with items $\{1, 2, \cdots, i\}$ and budget W'
- Longest common subsequence: $opt[i, j]$ = value of instance defined by $A[1..i]$ and $B[1..j]$
- Shortest paths in DAG: $f[v]$ = length of shortest path from s to v
- Matrix chain multiplication, optimum binary search tree: $opt[i, j]$ = value of instances defined by matrices i to j