CSE 431/531: Analysis of Algorithms

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

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] ← 0
F[1] ← 1
for i ← 2 to n do
    F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

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

Outline

1. Weighted Interval Scheduling
2. Subset Sum Problem
3. Knapsack Problem
4. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
5. Shortest Paths in Graphs with Negative Weights
   - Shortest Paths in Directed Acyclic Graphs
   - Bellman-Ford Algorithm
6. All-Pair Shortest Paths and Floyd-Warshall
7. Matrix Chain Multiplication
8. Summary

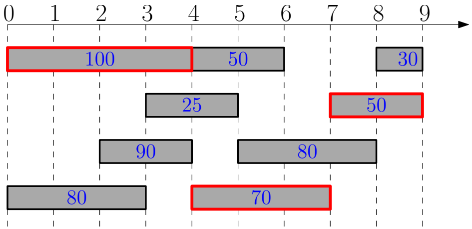
Recall: Interval Schduling

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



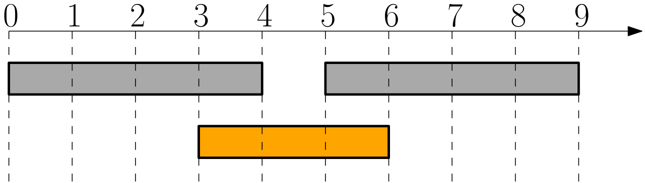
Optimum value = 220

Hard to Design a Greedy Algorithm

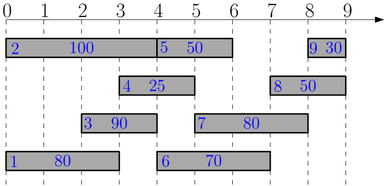
Q: Which job is safe to schedule?

- 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



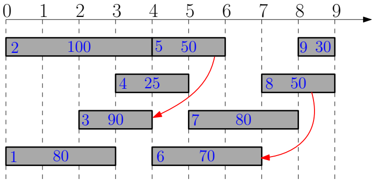
Designing a Dynamic Programming Algorithm



- 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: $v_i + opt[p_i]$, $\quad p_i =$ 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 $f_j \le s_i$

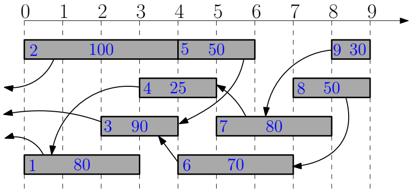
Recursion for $opt[i]$:

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

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

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

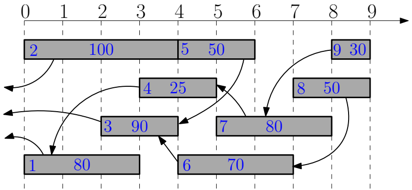


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\} = 100$
- $opt[4] = \max\{opt[3], 25 + opt[1]\} = 105$
- $opt[5] = \max\{opt[4], 50 + opt[3]\} = 150$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

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



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

Recursive Algorithm to Compute $opt[n]$

1. sort jobs by non-decreasing order of finishing times
2. compute $p_1, p_2, \cdots, p_n$
3. return compute-opt(n)

compute-opt(i)

1. if $i = 0$ then
2.   return 0
3. else
4.   return $\max\{$compute-opt$(i-1), v_i +$ compute-opt$(p_i)\}$

- Running time can be exponential in n
- Reason: we are computed each $opt[i]$ many times
- Solution: store the value of $opt[i]$, so it's computed only once

Memoized Recursive Algorithm

1. sort jobs by non-decreasing order of finishing times
2. compute $p_1, p_2, \cdots, p_n$
3. $opt[0] \leftarrow 0$ and $opt[i] \leftarrow \perp$ for every $i = 1, 2, 3, \cdots, n$
4. return compute-opt(n)

compute-opt(i)

1. if $opt[i] = \perp$ then
2. $\quad opt[i] \leftarrow \max\{$compute-opt$(i-1), v_i +$ compute-opt$(p_i)\}$
3. return $opt[i]$

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

Dynamic Programming

1. sort jobs by non-decreasing order of finishing times
2. compute $p_1, p_2, \cdots, p_n$
3. $opt[0] \leftarrow 0$
4. for $i \leftarrow 1$ to n
5. $\quad opt[i] \leftarrow \max\{opt[i-1], v_i + opt[p_i]\}$

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

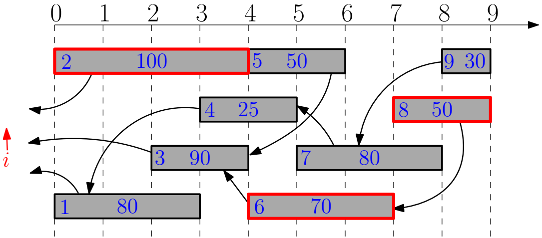
How Can We Recover the Optimum Schedule?

- 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
- if $opt[i-1] \geq v_i + opt[p_i]$
- $opt[i] \leftarrow opt[i-1]$
- $b[i] \leftarrow$ N
- else
- $opt[i] \leftarrow v_i + opt[p_i]$
- $b[i] \leftarrow$ Y

- $i \leftarrow n, S \leftarrow \emptyset$
- while $i \neq 0$
- if $b[i] =$ N
- $i \leftarrow i - 1$
- 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

1. Weighted Interval Scheduling
2. Subset Sum Problem
3. Knapsack Problem
4. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
5. Shortest Paths in Graphs with Negative Weights
   - Shortest Paths in Directed Acyclic Graphs
   - Bellman-Ford Algorithm
6. All-Pair Shortest Paths and Floyd-Warshall
7. Matrix Chain Multiplication
8. Summary

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

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

- 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
    $opt[0, W'] \leftarrow 0$
for $i \leftarrow 1$ to n
    for $W' \leftarrow 0$ to W
        $opt[i, W'] \leftarrow opt[i-1, W']$
        if $w_i \geq W'$ 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
    $opt[0, W'] \leftarrow 0$
for $i \leftarrow 1$ to n
    for $W' \leftarrow 0$ to W
        $opt[i, W'] \leftarrow opt[i-1, W']$
        $b[i, W'] \leftarrow$ 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 $i > 0$
    if $b[i, W'] =$ 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
    $opt[0, W'] \leftarrow 0$
for $i \leftarrow 1$ to n
    for $W' \leftarrow 0$ to W
        $opt[i, W'] \leftarrow opt[i-1, W']$
        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.

Outline

1. Weighted Interval Scheduling
2. Subset Sum Problem
3. Knapsack Problem
4. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
5. Shortest Paths in Graphs with Negative Weights
   - Shortest Paths in Directed Acyclic Graphs
   - Bellman-Ford Algorithm
6. All-Pair Shortest Paths and Floyd-Warshall
7. Matrix Chain Multiplication
8. Summary

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

Greedy Algorithm

1. sort items according to non-increasing order of v_i/w_i
2. for each item in the ordering
3. take the item if we have enough budget

- Bad example: $W = 100, n = 2, w = (1, 100), v = (1.1, 100)$.
- Optimum takes item 2 and greedy takes item 1.

Fractional Knapsack Problem

Input: 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 vector $(\alpha_1, \alpha_2, \cdots, \alpha_n) \in [0,1]^n$ that

$$\text{maximizes } \sum_{i=1}^{n} \alpha_i v_i \qquad \text{s.t. } \sum_{i=1}^{n} \alpha_i w_i \le W.$$

Greedy Algorithm for Fractional Knapsack

1. sort items according to non-increasing order of v_i/w_i,
2. for each item according to the ordering, take as much fraction of the item as possible.

Greedy is Optimum for Fractional Knapsack

Greedy Algorithm for Fractional Knapsack

1. sort items according to non-increasing order of v_i/w_i,
2. for each item according to the ordering, take as much fraction of the item as possible.

- $W = 100, n = 2, w = (1, 100), v = (1.1, 100)$.
- $\alpha_1 = 1, \alpha_2 = 0.99$, value $= 1.1 + 99 = 100.1$.
- Idea of proof: exchanging argument. (Left as homework exercise).

DP for ($\{0,1\}$-)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 $W' = 0, 1, 2, \cdots, W$.

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

Avoiding Unncessary Computation and Memory Using Memoized Algorithm and Hash Map

compute-opt(i, W')

```
if opt[i, W'] ≠ ⊥ 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) + v_i
        if r' > r then r ← r'
opt[i, W'] ← r
return r
```

- Use hash map for opt

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

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

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

Outline

1. Weighted Interval Scheduling
2. Subset Sum Problem
3. Knapsack Problem
4. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
5. Shortest Paths in Graphs with Negative Weights
   - Shortest Paths in Directed Acyclic Graphs
   - Bellman-Ford Algorithm
6. All-Pair Shortest Paths and Floyd-Warshall
7. Matrix Chain Multiplication
8. Summary

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 $B[1 .. m]$

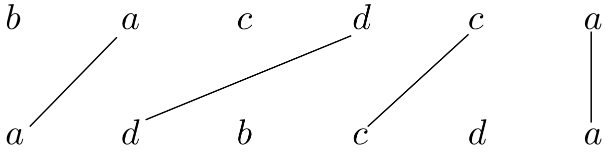
Output: the longest common subsequence of A and B

Example:

- $A = `bacdca'$
- $B = `adbcda'$
- $\text{LCS}(A, B) = `adca'$

- 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

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

Dynamic Programming for LCS

- $opt[i,j], 0 \le i \le n, 0 \le j \le m$: 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
    $opt[i, 0] \leftarrow 0$
    for $j \leftarrow 1$ to m
        if $A[i] = B[j]$ then
            $opt[i, j] \leftarrow opt[i-1, j-1] + 1$, $\pi[i, j] \leftarrow$ "$\nwarrow$"
        elseif $opt[i, j-1] \geq opt[i-1, j]$ 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

$i \leftarrow n, j \leftarrow m, S \leftarrow$ “”
while $i > 0$ and $j > 0$
    if $\pi[i, j] =$ “$\nwarrow$” then
        $S \leftarrow A[i] \bowtie S, i \leftarrow i - 1, j \leftarrow j - 1$
    else if $\pi[i, j] =$ “$\uparrow$”
        $i \leftarrow i - 1$
    else
        $j \leftarrow j - 1$
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)

- $opt[i,j], 0 \le i \le n, 0 \le j \le m$: edit distance between $A[1 \mathrel{..} i]$ and $B[1 \mathrel{..} j]$.
- 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

1. Weighted Interval Scheduling
2. Subset Sum Problem
3. Knapsack Problem
4. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
5. Shortest Paths in Graphs with Negative Weights
   - Shortest Paths in Directed Acyclic Graphs
   - Bellman-Ford Algorithm
6. All-Pair Shortest Paths and Floyd-Warshall
7. Matrix Chain Multiplication
8. Summary

Computing the Length of LCS

```
for j ← 0 to m do
    opt[0, j] ← 0
for i ← 1 to n
    opt[i, 0] ← 0
    for j ← 1 to m
        if A[i] = B[j]
            opt[i, j] ← opt[i − 1, j − 1] + 1
        elseif opt[i, j − 1] ≥ opt[i − 1, j]
            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 $j \leftarrow 0$ to m do
  $opt[0, j] \leftarrow 0$
for $i \leftarrow 1$ to n
  $opt[i \text{ mod } 2, 0] \leftarrow 0$
  for $j \leftarrow 1$ to m
    if $A[i] = B[j]$
      $opt[i \text{ mod } 2, j] \leftarrow opt[i - 1 \text{ mod } 2, j - 1] + 1$
    elseif $opt[i \text{ mod } 2, j - 1] \geq opt[i - 1 \text{ mod } 2, j]$
      $opt[i \text{ mod } 2, j] \leftarrow opt[i \text{ mod } 2, j - 1]$
    else
      $opt[i \text{ mod } 2, j] \leftarrow opt[i - 1 \text{ mod } 2, j]$
return $opt[n \text{ mod } 2, m]$

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

1. Weighted Interval Scheduling
2. Subset Sum Problem
3. Knapsack Problem
4. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
5. Shortest Paths in Graphs with Negative Weights
   - Shortest Paths in Directed Acyclic Graphs
   - Bellman-Ford Algorithm
6. All-Pair Shortest Paths and Floyd-Warshall
7. Matrix Chain Multiplication
8. Summary

Recall: Single Source Shortest Path Problem

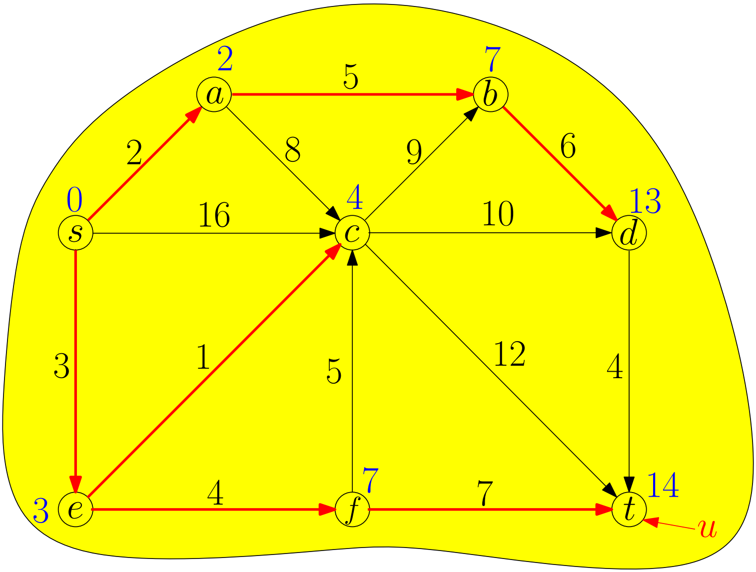
Single Source Shortest Paths

Input: directed graph $G = (V, E)$, $s \in V$

$w : E \to \mathbb{R}_{\geq 0}$

Output: shortest paths from s to all other vertices $v \in V$

- Algorithm for the problem: Dijkstra's algorithm


2
7
5
a
b
2
8
9
6
0
4
13
16
10
s
c
d
3
1
5
12
4
7
14
3
4
7
e
f
t
u

Dijkstra's Algorithm Using Priorty Queue

Dijkstra(G, w, s)

```
S ← ∅, d(s) ← 0 and d(v) ← ∞ for every v ∈ V \ {s}
Q ← empty queue, for each v ∈ V: Q.insert(v, d(v))
while S ≠ V, do
    u ← Q.extract_min()
    S ← S ∪ {u}
    for each v ∈ V \ S such that (u, v) ∈ E
        if d(u) + w(u, v) < d(v) then
            d(v) ← d(u) + w(u, v), Q.decrease_key(v, d(v))
            π(v) ← u
return (π, d)
```

- Running time $= O(m + n \lg n)$.

Single Source Shortest Paths, Weights May be Negative

Input: directed graph $G = (V, E)$, $s \in V$

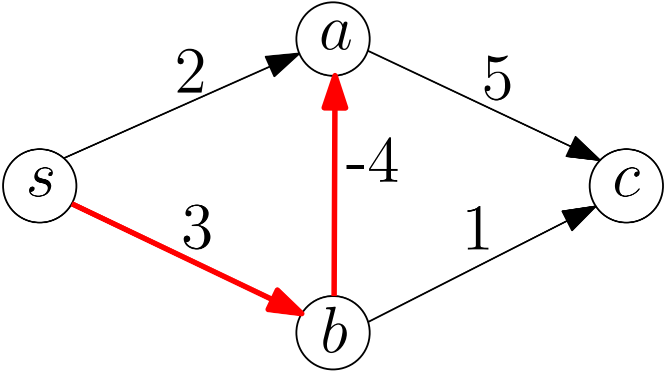
assume all vertices are reachable from s

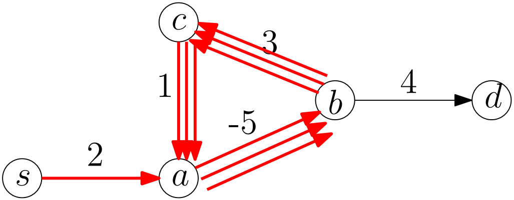
$w : E \to \mathbb{R}$

Output: shortest paths from s to all other vertices $v \in V$

- In transition graphs, negative weights make sense
- If we sell a item: 'having the item' $\to$ 'not having the item', weight is negative (we gain money)
- Dijkstra's algorithm does not work any more!

Dijkstra's Algorithm Fails if We Have Negative Weights





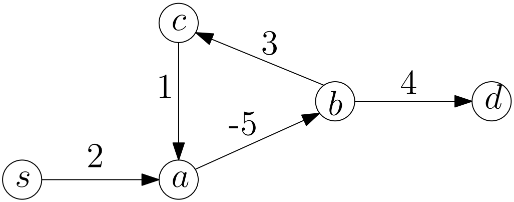
Q: What is the length of the shortest path from s to d?

A: $-\infty$

Def. A negative cycle is a cycle in which the total weight of edges is negative.

Dealing with Negative Cycles

- assume the input graph does not contain negative cycles, or
- allow algorithm to report "negative cycle exists"



Q: What is the length of the shortest simple path from s to d?

A: 1

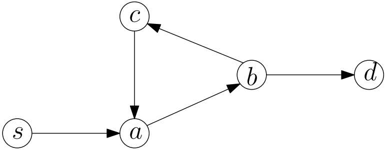
- Unfortunately, computing the shortest simple between two vertices is an NP-hard problem.

Outline

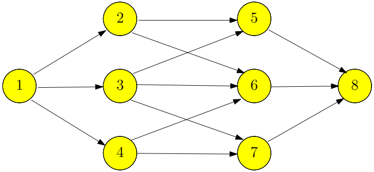
1. Weighted Interval Scheduling
2. Subset Sum Problem
3. Knapsack Problem
4. Longest Common Subsequence
    - Longest Common Subsequence in Linear Space
5. Shortest Paths in Graphs with Negative Weights
    - Shortest Paths in Directed Acyclic Graphs
    - Bellman-Ford Algorithm
6. All-Pair Shortest Paths and Floyd-Warshall
7. Matrix Chain Multiplication
8. Summary

Directed Acyclic Graphs

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



not a DAG



a DAG

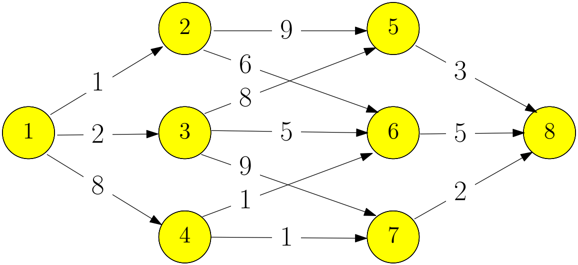
Lemma A directed graph is a DAG if and only its vertices can be topologically sorted.

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

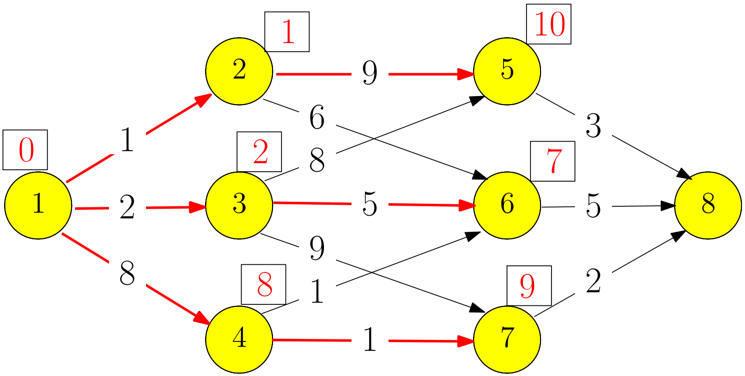
Shortest Paths in DAG

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

print-path(t)

if $t = 1$ then
    print(1)
    return
print-path$(\pi(t))$
print(",", t)

Example



Outline

1. Weighted Interval Scheduling
2. Subset Sum Problem
3. Knapsack Problem
4. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
5. Shortest Paths in Graphs with Negative Weights
   - Shortest Paths in Directed Acyclic Graphs
   - Bellman-Ford Algorithm
6. All-Pair Shortest Paths and Floyd-Warshall
7. Matrix Chain Multiplication
8. Summary

Defining Cells of Table

Single Source Shortest Paths, Weights May be Negative

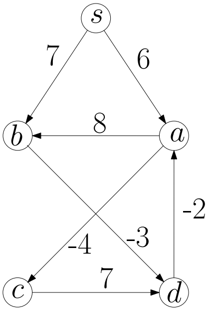
Input: directed graph $G = (V, E)$, $s \in V$

assume all vertices are reachable from s

$w : E \to \mathbb{R}$

Output: shortest paths from s to all other vertices $v \in V$

- first try: $f[v]$: length of shortest path from s to v
- issue: do not know in which order we compute $f[v]$'s
- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges



- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges
- $f^2[a] = 6$
- $f^3[a] = 2$

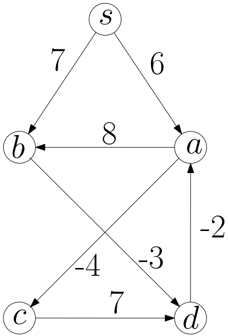
$$f^\ell[v] = \begin{cases} 0 & \ell = 0, v = s \\ \infty & \ell = 0, v \neq s \\ \min \begin{cases} f^{\ell-1}[v] \\ \min_{u:(u,v)\in E} \left(f^{\ell-1}[u] + w(u,v) \right) \end{cases} & \ell > 0 \end{cases}$$

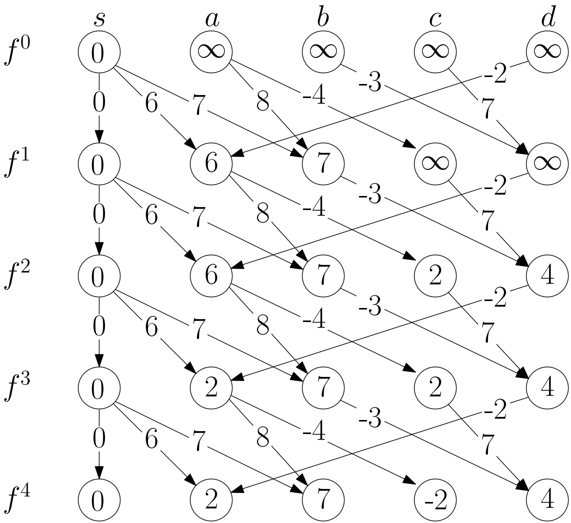
dynamic-programming(G, w, s)

$f^0[s] \leftarrow 0$ and $f^0[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
for $\ell \leftarrow 1$ to $n-1$ do
    copy $f^{\ell-1} \to f^{\ell}$
    for each $(u, v) \in E$
        if $f^{\ell-1}[u] + w(u, v) < f^{\ell}[v]$
            $f^{\ell}[v] \leftarrow f^{\ell-1}[u] + w(u, v)$
return $(f^{n-1}[v])_{v \in V}$

Obs. Assuming there are no negative cycles, then a shortest path contains at most $n-1$ edges

Dynamic Programming: Example





dynamic-programming(G, w, s)

$f^0[s] \leftarrow 0$ and $f^0[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
for $\ell \leftarrow 1$ to $n - 1$ do
    copy $f^{\ell-1} \to f^\ell$
    for each $(u, v) \in E$
        if $f^{\ell-1}[u] + w(u, v) < f^\ell[v]$
            $f^\ell[v] \leftarrow f^{\ell-1}[u] + w(u, v)$
return $(f^{n-1}[v])_{v \in V}$

Obs. Assuming there are no negative cycles, then a shortest path contains at most $n - 1$ edges

Q: What if there are negative cycles?

Dynamic Programming With Negative Cycle Detection

dynamic-programming(G, w, s)

$f^0[s] \leftarrow 0$ and $f^0[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
for $\ell \leftarrow 1$ to $n-1$ do
    copy $f^{\ell-1} \rightarrow f^{\ell}$
    for each $(u, v) \in E$
        if $f^{\ell-1}[u] + w(u, v) < f^{\ell}[v]$
            $f^{\ell}[v] \leftarrow f^{\ell-1}[u] + w(u, v)$
for each $(u, v) \in E$
    if $f^{n-1}[u] + w(u, v) < f^{n-1}[v]$
        report "negative cycle exists" and exit
return $(f^{n-1}[v])_{v \in V}$

Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

1. $f[s] \leftarrow 0$ and $f[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
2. for $\ell \leftarrow 1$ to $n-1$ do
3.   for each $(u, v) \in E$
4.   if $f[u] + w(u, v) < f[v]$
5. $f[v] \leftarrow f[u] + w(u, v)$
6. return f

- Issue: when we compute $f[u] + w(u, v)$, $f[u]$ may be changed since the end of last iteration
- This is OK: it can only "accelerate" the process!

- After iteration ℓ, $f[v]$ is at most the length of the shortest path from s to v that uses at most ℓ edges
- $f[v]$ is always the length of some path from s to v

Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

```
f[s] ← 0 and f[v] ← ∞ for any v ∈ V \ {s}
for ℓ ← 1 to n − 1 do
    for each (u, v) ∈ E
        if f[u] + w(u, v) < f[v]
            f[v] ← f[u] + w(u, v)
return f
```

- After iteration ℓ, $f[v]$ is at most the length of the shortest path from s to v that uses at most ℓ edges
- $f[v]$ is always the length of some path from s to v
- Assuming there are no negative cycles, after iteration $n-1$, $f[v] =$ length of shortest path from s to v

Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

1. $f[s] \leftarrow 0$ and $f[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
2. for $\ell \leftarrow 1$ to n do
3. $\quad$ $updated \leftarrow$ false
4. $\quad$ for each $(u, v) \in E$
5. $\quad\quad$ if $f[u] + w(u, v) < f[v]$
6. $\quad\quad\quad$ $f[v] \leftarrow f[u] + w(u, v)$, $\pi[v] \leftarrow u$
7. $\quad\quad\quad$ $updated \leftarrow$ true
8. $\quad$ if not $updated$, then return f
9. output "negative cycle exists"

- $\pi[v]$: the parent of v in the shortest path tree
- Running time $= O(nm)$

Outline

1. Weighted Interval Scheduling
2. Subset Sum Problem
3. Knapsack Problem
4. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
5. Shortest Paths in Graphs with Negative Weights
   - Shortest Paths in Directed Acyclic Graphs
   - Bellman-Ford Algorithm
6. All-Pair Shortest Paths and Floyd-Warshall
7. Matrix Chain Multiplication
8. Summary

All-Pair Shortest Paths

All Pair Shortest Paths

Input: directed graph $G = (V, E)$,
$w : E \to \mathbb{R}$ (can be negative)

Output: shortest path from u to v for every $u, v \in V$

1. for every starting point $s \in V$ do
2.   run Bellman-Ford(G, w, s)

- Running time $= O(n^2 m)$

Design a Dynamic Programming Algorithm

- It is convenient to assume $V = \{1, 2, 3, \cdots, n\}$
- For simplicity, extend the w values to non-edges:

$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

- For now assume there are no negative cycles

Cells for Floyd-Warshall Algorithm

- First try: $f[i,j]$ is length of shortest path from i to j
- Issue: do not know in which order we compute $f[i,j]$'s
- $f^k[i,j]$: length of shortest path from i to j that only uses vertices $\{1, 2, 3, \cdots, k\}$ as intermediate vertices

$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

- $f^k[i,j]$: length of shortest path from i to j that only uses vertices $\{1, 2, 3, \cdots, k\}$ as intermediate vertices

$$f^k[i,j] = \begin{cases} w(i,j) & k = 0 \\ \min \begin{cases} f^{k-1}[i,j] \\ f^{k-1}[i,k] + f^{k-1}[k,j] \end{cases} & k = 1, 2, \cdots, n \end{cases}$$

Floyd-Warshall(G, w)

$f^0 \leftarrow w$
for $k \leftarrow 1$ to n do
    copy $f^{k-1} \rightarrow f^k$
    for $i \leftarrow 1$ to n do
        for $j \leftarrow 1$ to n do
            if $f^{k-1}[i,k] + f^{k-1}[k,j] < f^k[i,j]$ then
                $f^k[i,j] \leftarrow f^{k-1}[i,k] + f^{k-1}[k,j]$

Floyd-Warshall(G, w)

$f^{\text{old}} \leftarrow w$
for $k \leftarrow 1$ to n do
  copy $f^{\text{old}} \rightarrow f^{\text{new}}$
  for $i \leftarrow 1$ to n do
    for $j \leftarrow 1$ to n do
      if $f^{\text{old}}[i,k] + f^{\text{old}}[k,j] < f^{\text{new}}[i,j]$ then
        $f^{\text{new}}[i,j] \leftarrow f^{\text{old}}[i,k] + f^{\text{old}}[k,j]$

Lemma Assume there are no negative cycles in G. After iteration k, for $i, j \in V$, $f[i,j]$ is exactly the length of shortest path from i to j that only uses vertices in $\{1, 2, 3, \cdots, k\}$ as intermediate vertices.

- Running time $= O(n^3)$.

Recovering Shortest Paths

Floyd-Warshall(G, w)

$f \leftarrow w$, $\pi[i,j] \leftarrow \perp$ for every $i, j \in V$
for $k \leftarrow 1$ to n do
  for $i \leftarrow 1$ to n do
    for $j \leftarrow 1$ to n do
      if $f[i,k] + f[k,j] < f[i,j]$ then
        $f[i,j] \leftarrow f[i,k] + f[k,j]$, $\pi[i,j] \leftarrow k$

print-path(i, j)

if $\pi[i,j] = \perp$ then
  if $i \neq j$ then print$(i, \text{“,”})$
else
  print-path$(i, \pi[i,j])$, print-path$(\pi[i,j], j)$

Detecting Negative Cycles

Floyd-Warshall(G, w)

$f \leftarrow w$, $\pi[i,j] \leftarrow \perp$ for every $i, j \in V$
for $k \leftarrow 1$ to n do
    for $i \leftarrow 1$ to n do
        for $j \leftarrow 1$ to n do
            if $f[i,k] + f[k,j] < f[i,j]$ then
                $f[i,j] \leftarrow f[i,k] + f[k,j]$, $\pi[i,j] \leftarrow k$
for $k \leftarrow 1$ to n do
    for $i \leftarrow 1$ to n do
        for $j \leftarrow 1$ to n do
            if $f[i,k] + f[k,j] < f[i,j]$ then
                report "negative cycle exists" and exit

Outline

1. Weighted Interval Scheduling
2. Subset Sum Problem
3. Knapsack Problem
4. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
5. Shortest Paths in Graphs with Negative Weights
   - Shortest Paths in Directed Acyclic Graphs
   - Bellman-Ford Algorithm
6. All-Pair Shortest Paths and Floyd-Warshall
7. Matrix Chain Multiplication
8. Summary

Matrix Chain Multiplication

Matrix Chain Multiplication

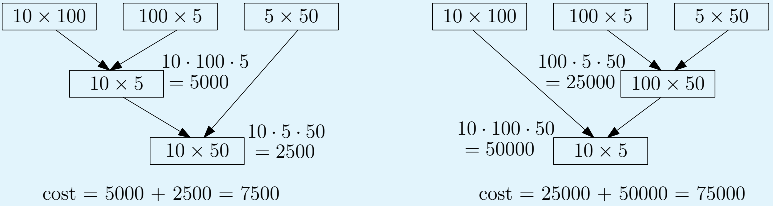
Input: n matrices $A_1, A_2, \cdots, A_n$ of sizes $r_1 \times c_1, r_2 \times c_2, \cdots, r_n \times c_n$, such that $c_i = r_{i+1}$ for every $i = 1, 2, \cdots, n-1$.

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:

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



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

Matrix Chain Multiplication: Design DP

- Assume the last step is $(A_1A_2\cdots A_i)(A_{i+1}A_{i+2}\cdots A_n)$
- Cost of last step: $r_1 \times c_i \times c_n$
- Optimality for sub-instances: we need to compute $A_1A_2\cdots A_i$ and $A_{i+1}A_{i+2}\cdots A_n$ optimally
- $opt[i,j]$: the minimum cost of computing $A_iA_{i+1}\cdots A_j$

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

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

let $opt[i,i] \leftarrow 0$ for every $i = 1, 2, \cdots, n$

for $\ell \leftarrow 2$ to n

    for $i \leftarrow 1$ to $n - \ell + 1$

        $j \leftarrow i + \ell - 1$

        $opt[i,j] \leftarrow \infty$

        for $k \leftarrow i$ to $j - 1$

            if $opt[i,k] + opt[k+1,j] + r_i c_k c_j < opt[i,j]$

                $opt[i,j] \leftarrow opt[i,k] + opt[k+1,j] + r_i c_k c_j$

return $opt[1,n]$

Outline

1. Weighted Interval Scheduling
2. Subset Sum Problem
3. Knapsack Problem
4. Longest Common Subsequence
   - Longest Common Subsequence in Linear Space
5. Shortest Paths in Graphs with Negative Weights
   - Shortest Paths in Directed Acyclic Graphs
   - Bellman-Ford Algorithm
6. All-Pair Shortest Paths and Floyd-Warshall
7. Matrix Chain Multiplication
8. Summary

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

Definition of Cells for Problems We Learnt

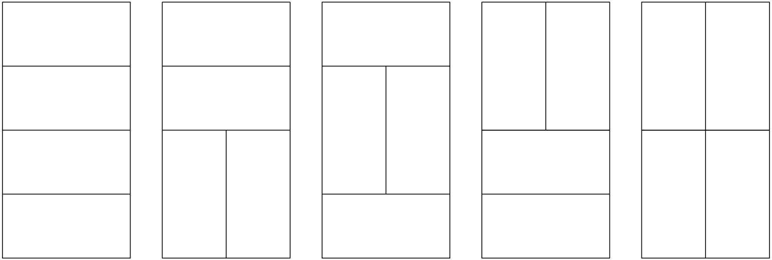
- Weighted interval scheduling: $opt[i] =$ value of instance defined by jobs $\{1, 2, \cdots, i\}$
- Subset sum, knapsack: $opt[i, W'] =$ value of instance with items $\{1, 2, \cdots, i\}$ and budget W'
- Longest common subsequence: $opt[i, j] =$ value of instance defined by $A[1..i]$ and $B[1..j]$
- Matrix chain multiplication: $opt[i, j] =$ value of instances defined by matrices i to j
- Shortest paths in DAG: $f[v] =$ length of shortest path from s to v
- Bellman-Ford: $f^\ell[v] =$ length of shortest path from s to v that uses at most ℓ edges
- Floyd-Warshall: $f^k[i, j] =$ length of shortest path from i to j that only uses $\{1, 2, \cdots, k\}$ as intermediate vertices

Exercise: Counting Number of Domino Coverings

Input: n

Output: number of ways to cover a $n \times 2$ grid using domino tiles

- **Example**: 5 different ways if $n = 4$:



- How about number of ways to cover a $n \times 3$ grid?

Exercise: Maximum-Weight Subset with Gaps

Input: n, integers $w_1, w_2, \cdots, w_n \geq 0$.

Output: a set $S \subseteq \{1, 2, 3 \cdots, n\}$ that

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

$$\forall i, j \in S, i \neq j, \text{ we have } |i - j| \geq 2.$$

- Example: $n = 7, w = (10, 80, 100, 90, 30, 50, 70)$
- Choose items $2, 4, 7$: value $= 80 + 90 + 70 = 240$

Def. Given a sequence $A = (a_1, a_2, \cdots, a_n)$ of n numbers, an increasing subsequence of A is a subsequence $(A_{i_1}, A_{i_2}, A_{i_3}, \cdots, A_{i,t})$ such that $1 \leq i_1 < i_2 < i_3 < \cdots < i_t \leq n$ and $a_{i_1} < a_{i_2} < a_{i_3} < \cdots < a_{i_t}$.

Exercise: Longest Increasing Subsequence

Input: $A = (a_1, a_2, \cdots, a_n)$ of n numbers

Output: The length of the longest increasing sub-sequence of A

Example:

- Input: $(10, 3, 9, 8, 2, 5, 7, 1, 12)$
- Output: 4

Def. A sequence $X[1..m]$ of numbers is oscillating if $X[i] < X[i+1]$ for all even $i \le m-1$, and $X[i] > X[i+1]$ for all odd $i \le m-1$.

Example:

- $5, 3, 9, 7, 8, 6, 12, 11$ is an oscillating sequence: $5 > 3 < 9 > 7 < 8 > 6 < 12 > 11$

Exercise: Longest Oscillating Subsequence

Input: A sequence A of n numbers

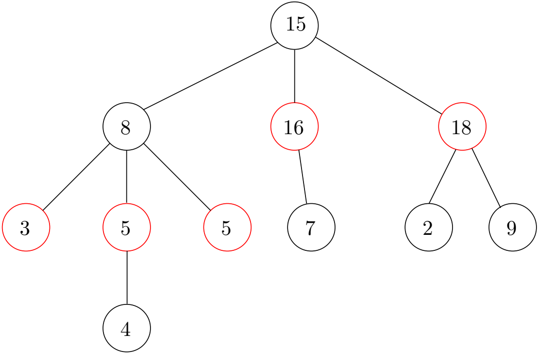
Output: The length of the longest oscillating subsequence of A

- Recall: an independent set of $G = (V, E)$ is a set $U \subseteq V$ such that there are no edges between vertices in U.

Maximum Weighted Independent Set in A Tree

Input: a tree with node weights

Output: the independent set of the tree with the maximum total weight



maximum-weight independent set has weight 47.