

Introduction to

Algorithm Design and Analysis

[17] Dynamic Programming 2

Jingwei Xu

[https://cs.nju.edu.cn/ics/people/jingweixu/
index.html](https://cs.nju.edu.cn/ics/people/jingweixu/index.html)

Institute of Computer Software

Nanjing University

In the last class...

- Basic idea of DP
- Least cost matrix multiplication
 - BF1, BF2
 - A DP solution
- Weighted binary search tree
 - The same DP solution

DP - II

- From the DP perspective
 - All-pairs shortest paths; SSSP over DAG
- More DP problems
 - Edit distance
 - Highway restaurants; Separating sequence of words
 - Changing coins
- Elements of DP

All-pairs Shortest Paths

- BF2
 - Path length k
 - $k \in [1, n]$
- Floyd algorithm
 - Index range k
 - $k \in [1, n]$

BF2

Length of the shortest path of at most k edges

$$dist(u, v, k) = \begin{cases} 0 & \text{if } u = v \\ \infty & \text{if } k = 0 \text{ and } u \neq v \\ \min_x (dist(u, x, k - 1) + w(x \rightarrow v)) & \text{otherwise} \end{cases}$$

APSP(V, E, w):

```

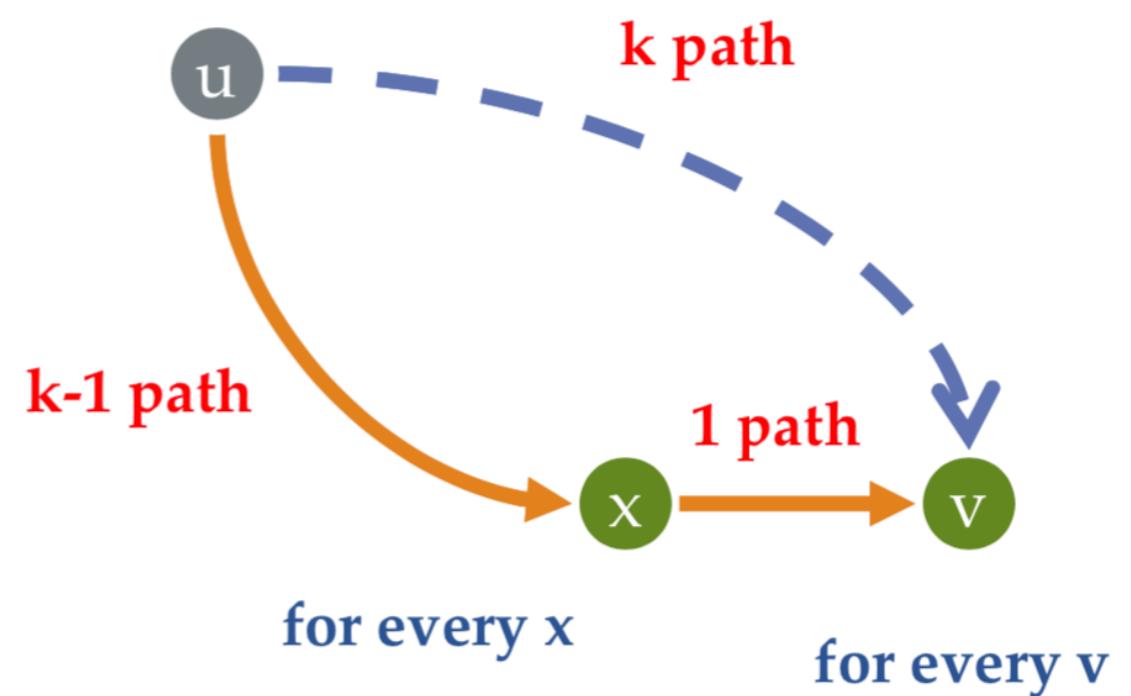
for all vertices u
    for all vertices v
        if  $u = v$ 
             $dist[u, v, 0] \leftarrow 0$ 
        else
             $dist[u, v, 0] \leftarrow \infty$ 

for  $k \leftarrow 1$  to  $V - 1$ 
    for all vertices u
        for all vertices v
             $dist[u, v, k] \leftarrow \infty$ 
        for all vertices x
            if  $dist[u, v, k] > dist[u, x, k - 1] + w(x \rightarrow v)$ 
                 $dist[u, v, k] \leftarrow dist[u, x, k - 1] + w(x \rightarrow v)$ 

```

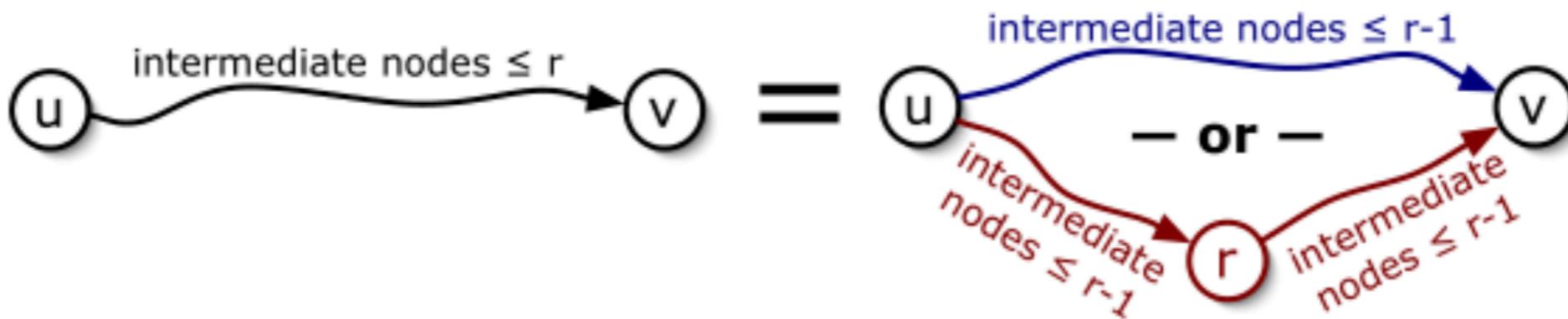
$O(n^4)$

for every u



Floyd Algorithm

- Basic idea



- Smart recursion

$$dist(u, v, r) = \begin{cases} w(u \rightarrow v) & \text{if } r = 0 \\ \min \{ dist(u, v, r - 1), dist(u, r, r - 1) + dist(r, v, r - 1) \} & \text{otherwise} \end{cases}$$

Floyd Algorithm

- Basic DP (3-dimensional)

```
FLOYDWARSHALL( $V, E, w$ ):
    for all vertices  $u$ 
        for all vertices  $v$ 
             $dist[u, v, 0] \leftarrow w(u \rightarrow v)$ 
    for  $r \leftarrow 1$  to  $V$ 
        for all vertices  $u$ 
            for all vertices  $v$ 
                if  $dist[u, v, r - 1] < dist[u, r, r - 1] + dist[r, v, r - 1]$ 
                     $dist[u, v, r] \leftarrow dist[u, v, r - 1]$ 
                else
                     $dist[u, v, r] \leftarrow dist[u, r, r - 1] + dist[r, v, r - 1]$ 
```

O(n^3)

- Improved DP (2-dimensional)

```
FLOYDWARSHALL2( $V, E, w$ ):
    for all vertices  $u$ 
        for all vertices  $v$ 
             $dist[u, v] \leftarrow w(u \rightarrow v)$ 
    for all vertices  $r$ 
        for all vertices  $u$ 
            for all vertices  $v$ 
                if  $dist[u, v] > dist[u, r] + dist[r, v]$ 
                     $dist[u, v] \leftarrow dist[u, r] + dist[r, v]$ 
```

O(n^3)

SSSP over a DAG

- Subproblems

- One problem for each node
 - $\text{dis}[1..n]$

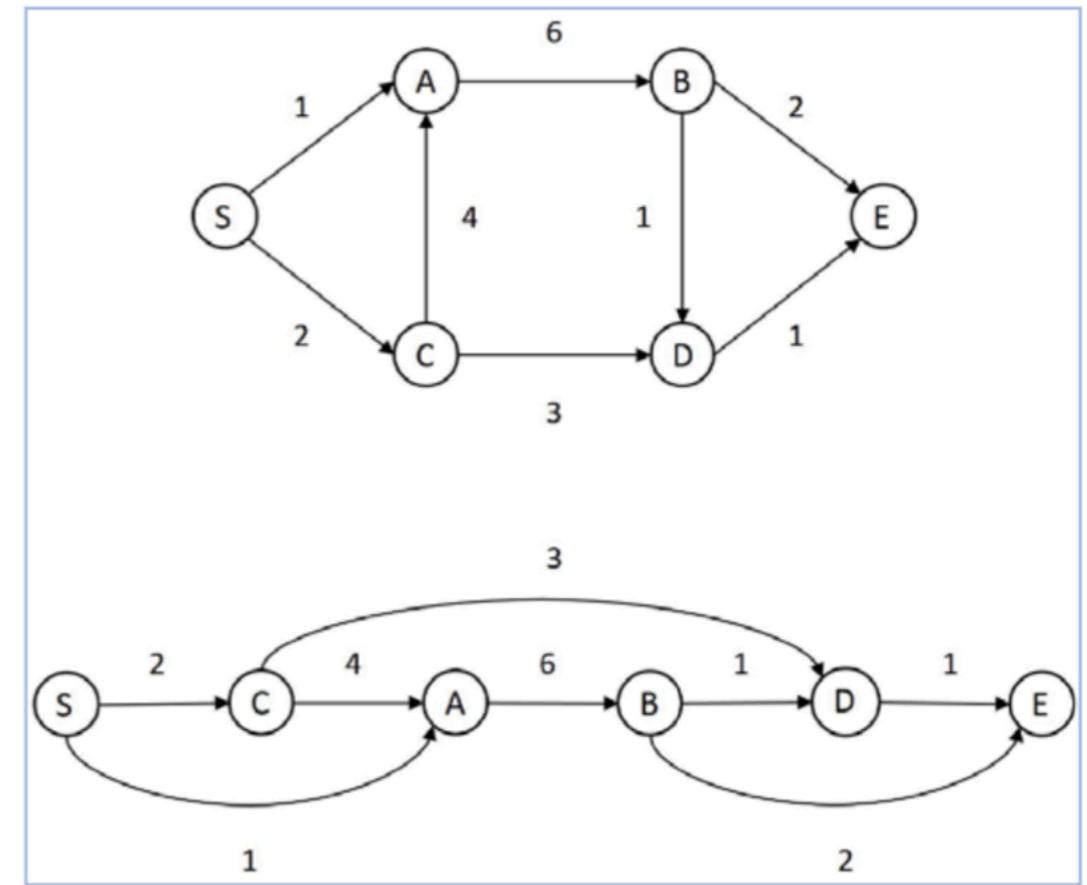
- Dynamic programming

- Topological ordering of nodes in a DAG

- More than SSSP

- As long as the recursion succeeds

$$D.\text{dis} = \min \{ B.\text{dis} + 1, C.\text{dis} + 3 \}$$



Edit Distance

- You can edit a word by

- Insert, Delete, Replace

- Edit distance

- Minimum number of edit operations

- Problem

- Given two strings,
compute the edit distance

F	O	O	D	
M	O	N	E	Y

4 op: **R** **R** **I** **R**

3 op: not possible

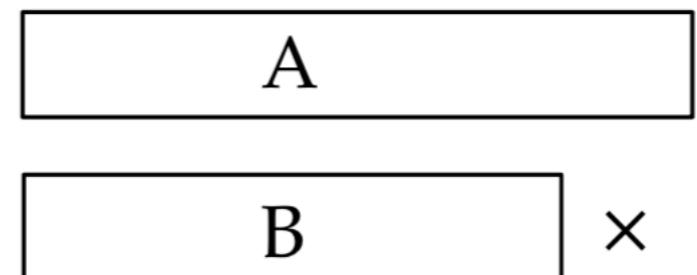
The edit
distance is **4**

“BF” Recursion

- Case 1

- 1.1 Insert
- 1.2 dual of case 1.1

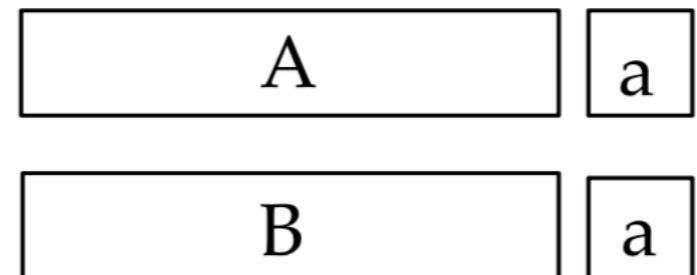
Case 1.1



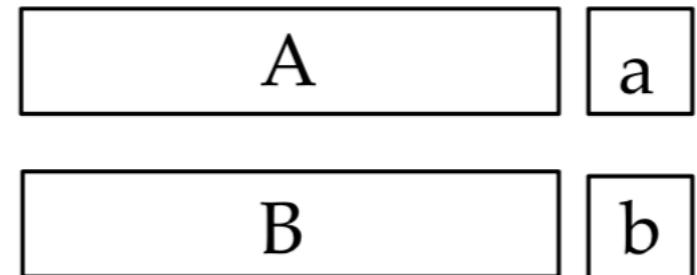
- Case 2

- 2.1 a=a
- 2.2 a!=b

Case 2.1



Case 2.2



“BF” Recursion

- **EditDis(i,j)**

- Base case:

- If $i=0$, $\text{EditDis}(i,j)=j$

- If $j=0$, $\text{EditDis}(i,j)=i$

- Recursion:

$$\text{EditDis}(A[1..m], B[1..n]) = \min \begin{cases} \text{EditDis}(A[1..m - 1], B[1..n]) + 1 \\ \text{EditDis}(A[1..m], B[1..n - 1]) + 1 \\ \text{EditDis}(A[1..m - 1], B[1..n - 1]) + I\{A[m] \neq B[n]\} \end{cases}$$

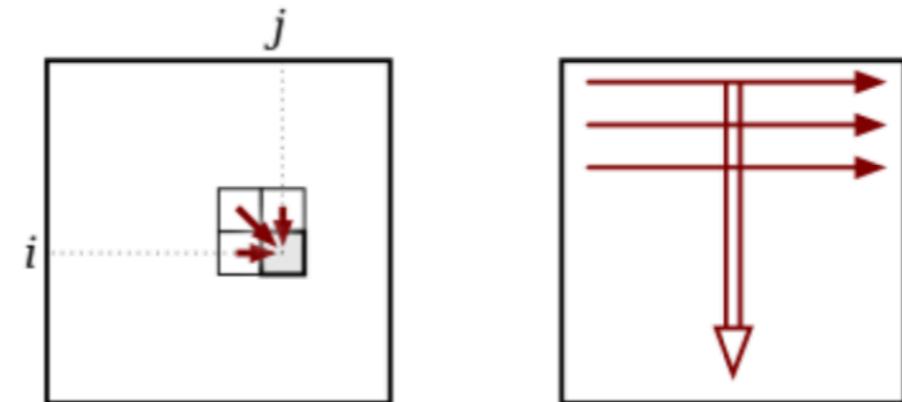
Smart Programming

- DP dict

- $\text{EditDis}[1..m, 1..n]$

- DP algorithm

dependencies



```
EDITDISTANCE( $A[1..m], B[1..n]$ ):
```

```
    for  $j \leftarrow 1$  to  $n$ 
         $Edit[0, j] \leftarrow j$ 

    for  $i \leftarrow 1$  to  $m$ 
         $Edit[i, 0] \leftarrow i$ 
        for  $j \leftarrow 1$  to  $n$ 
            if  $A[i] = B[j]$ 
                 $Edit[i, j] \leftarrow \min \{Edit[i - 1, j] + 1, Edit[i, j - 1] + 1, Edit[i - 1, j - 1]\}$ 
            else
                 $Edit[i, j] \leftarrow \min \{Edit[i - 1, j] + 1, Edit[i, j - 1] + 1, Edit[i - 1, j - 1] + 1\}$ 

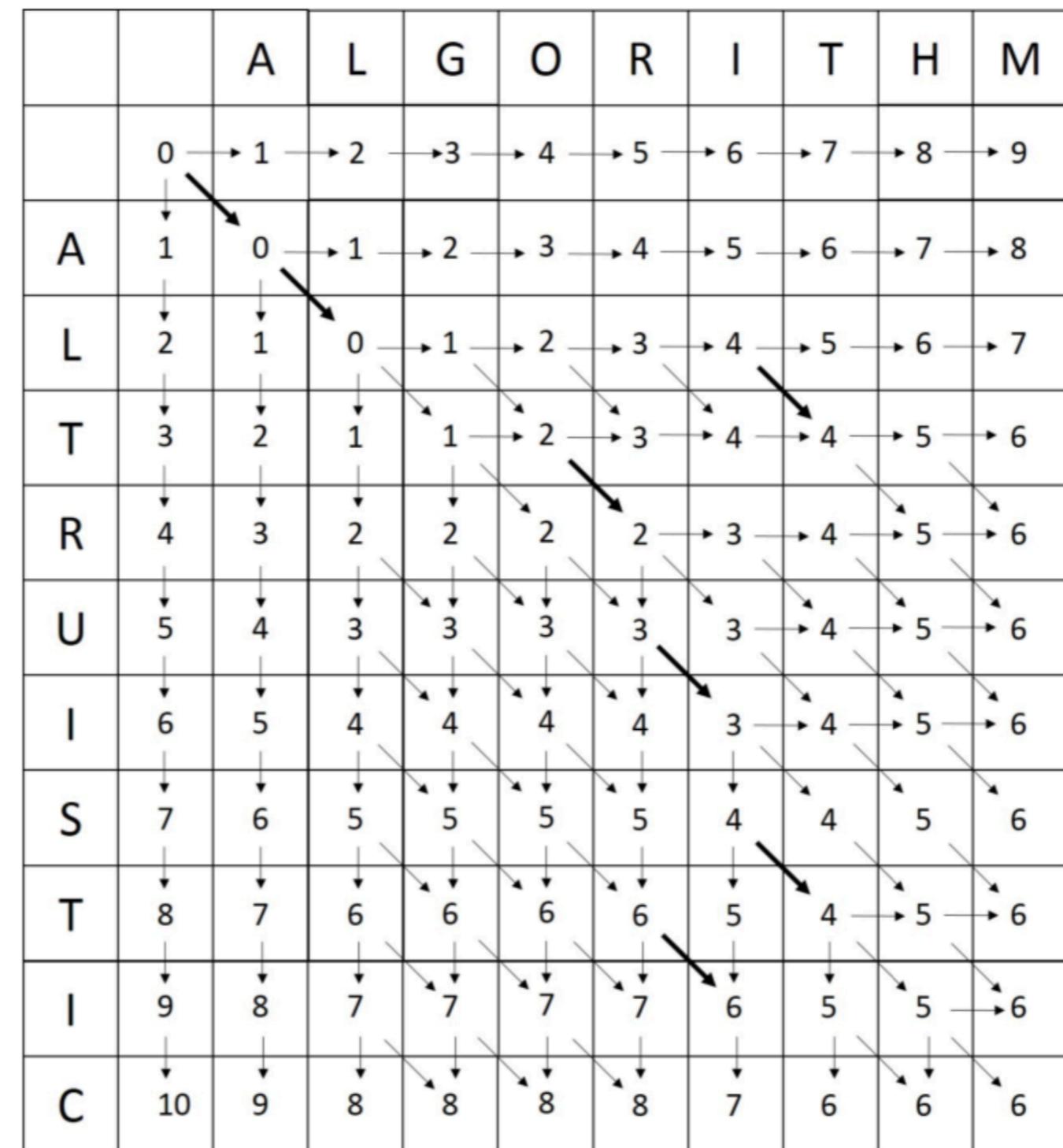
    return  $Edit[m, n]$ 
```

Example

algorithm

vs.

altruistic



DP in One Dimension

- Highway restaurants

- n possible locations on a straight line
 - $m_1, m_2, m_3, \dots, m_n$
 - At most one restaurant at one location
 - Expected profit for location i is p_i
 - Any two restaurants should be at least k miles apart

- How to arrange the restaurants

- To obtain the maximum expected profit

Highway Restaurants

- The recursion

- $P(j)$: the max profit achievable using only first j locations
 - $P(0)=0$
- $\text{prev}[j]$: largest index before j and k miles away

$$P(j) = \max(p_j + P(\text{prev}[j]), P(j - 1))$$

Highway Restaurants

- One dimension DP algorithm

- Fill in $P[0], P[1], \dots, P[n]$

(First compute the $\text{prev}[\cdot]$ array)

```
i = 0
for j = 1 to n:
    while  $m_{i+1} \leq m_j - k$ :
        i = i + 1
        prev[j] = i
```

(Now the dynamic programming begins)

```
P[0] = 0
for j = 1 to n:
    P[j] = max( $p_j + P[\text{prev}[j]]$ ,  $P[j - 1]$ )
return P[n]
```

Words into Lines

- Words into lines
 - Word-length w_1, w_2, \dots, w_n and line-width: W
- Basic constraint
 - If w_1, w_2, \dots, w_j are in one line, then $w_i + w_{i+1} + \dots + w_j \leq W$
- Penalty for one line: some function of X . X is:
 - 0 for the last line in a paragraph, and
 - $W - (w_i + w_{i+1} + \dots + w_j)$ for other lines
- The problem
 - How to make the penalty of the paragraph, which is the sum of the penalties of individual lines, minimized

Greedy Solution

i	word	w
1	Those	6
2	who	4
3	cannot	7
4	remember	9
5	the	4
6	past	5
7	are	4
8	condemned	10
9	to	3
10	repeat	7
11	it.	4

W is 17, and penalty is X^3

Solution by greedy strategy

words	(1,2,3)	(4,5)	(6,7)	(8,9)	(10,11)
X	0	4	8	4	0
penalty	0	64	512	64	0

Total penalty is **640**

An improved solution

words	(1,2)	(3,4)	(5,6,7)	(8,9)	(10,11)
X	7	1	4	4	0
penalty	343	1	64	64	0

Total penalty is **472**

Problem Decomposition

- Representation of subproblem: a pair of indexes (i,j) , breaking words i through j into lines with minimum penalty.
- Two kinds of subproblem
 - (k,n) : the penalty of the last line is 0
 - all other subproblems
- For some k , the combination of the optimal solution for $(1,k)$ and $(k+1, n)$ gives a optimal solution for $(1,n)$
- Subproblem graph
 - About n^2 vertices
 - Each vertex (i,j) has an edge to about $j-i$ other vertices, so, the number of edges is in $\Theta(n^3)$

Simpler Identification of Subproblems

- If a subproblem concludes the paragraph, then (k, n) can be simplified as (k)
 - About k subproblems
- Can we eliminate the use of (i, j) with $j < n$?
 - Put the first k words in the first line (with the basic constraint satisfied), the subproblem to be solved is $(k+1, n)$
 - Optimizing the solution over all k 's. (k is at most $W/2$)

One-dimension Recursion

- One-dimension problem space

- (1,n), (2,n), ..., (n,n)

Subproblem (i,n)

Algorithm: lineBreak(w, W, i, n, L)

if $w_i + w_{i+1} + \dots + w_n \leq W$ **then**

<Put all words on line L , set penalty to 0> ;

else

for $k = 1; w_i + \dots + w_{i+k-1} \leq W; k++$ **do**

$X = W - (w_i + \dots + w_{i+k-1})$;

$kPenalty = lineCost(X) + lineBreak(w, W, i + k, n, L + 1)$;

<Set penalty always to the minimum $kPenalty$ > ;

<Updating k_{min} , which records the k part that produced the minimum penalty> ;

<Put words i through $i + k_{min} - 1$ on line L > ;

return $penalty$;

Dynamic Programming

- Topological ordering of subproblems

- $\text{Penalty}[n] \rightarrow \text{Penalty}[n-1] \rightarrow \dots \rightarrow \text{Penalty}[1]$

Algorithm: lineBreakDP

```
for  $i = n; i \geq 1; i --$  do
    if all words through  $w_i$  to  $w_n$  can be put in one line then
         $\text{Penalty}[i] = 0$  ;
        <put all words through  $i$  to  $n$  in one line> ;
    else
        for  $k = 1; w_i + \dots + w_{i+k-1} \leq W; k ++$  do
            calculate the penalty  $\text{Cost}_{cur}$  of putting  $k$  words in this line ;
             $\text{minCost} = \min\{\text{minCost}, \text{Cost}_{cur} + \text{Penalty}[i + k]\}$  ;
            <Updating  $k_{min}$ , which records the  $k$  part that produced the minimum
            penalty> ;
            <Put words  $i$  through  $i + k_{min} - 1$  on one line> ;
         $\text{Penalty}[i] = \text{minCost}$  ;
```

Analysis of lineBreakDP

- Each subproblem is identified by only one integer k , for (k,n)
 - Number of vertex in the subproblem graph: at most n .
 - So, in DP version, the recursion is executed at most n times.
- So, the running time is in $\Theta(Wn)$
 - The loop is executed at most $W/2$ times.
 - In fact, W , the line width, is usually a constant. So, $\Theta(n)$.
 - The extra space for the dictionary is in $\Theta(n)$.

Making Change: Revisited

- How to pay a given amount of money?
 - Using the smallest possible number of coins
 - With certain systems of coinage
- We have known that the greedy strategy fails sometimes?

Subproblems

- Assumptions

- Given n different denotations
- A coin of denomination i has d_i units
- The amount to be paid: N

- Subproblem $[i,j]$

- The minimum number of coins required to pay an amount of j units, using only coins of denominations 1 to i .

- The problem

- Figure out subproblem $[n, N]$ (as $c[n, N]$)

Dependency of Subproblems

- $c[i,0]$ is 0 for all i
- When we are to pay an amount j using coins of denominations 1 to i , we have two choices:
 - No coins of denomination i is used: $c[i-1, j]$
 - One coins of denomination i is used: $1+c[i,j-d_i]$
- So, $c[i,j]=\min(c[i-1,j], 1+c[i,j-d_i])$

Data Structure

Define a array $\text{coin}[1..n, 0..N]$ for all $c[i, j]$

an example

	0	1	2	3	4	5	6	7	8
$d_1=1$	0	1	2	3	4	5	6	7	8
$d_2=4$	0	1	2	3	1	2	3	4	2
$d_3=6$	0	1	2	3	1	2	1	2	2

direction of computation

The Procedure

```
int coinChange(int N, int n, int[] coin)
    int denomination[]={d1,d2,...,dn};
    for (i=1; i≤n; i++)
        coin[i,0]=0;
    for (i=1; i≤n; i++) {
        for (j=1; j≤N; j++)
            if (i==1 && j<denomination[i]) coin[i,j]=+∞ ;
            else if (i==1) coin[i,j]=1+coin[1,j-denomination[1]];
            else if (j<denomination[i]) coin[i,j]=cost[i-1,j];
            else coin[i,j]=min(coin[i-1,j], 1+coin[i,j-denomination[i]]);
    }
    return coin[n,N];
```

in $\Theta(nN)$,
n is usually a constant

Other DP Problems

- **Text string problems**
 - Longest common subsequence, ...
 - Variations of standard text string problems, ...
- **One dimensional problems**
 - Arrangements along a straight line, ...
- **Graph problems**
 - Vertex cover, ...
- **Hard problems**
 - Knapsack problems and variations, ...

Principle of optimality

- Given an optimal sequence of decisions, each subsequence must be optimal by itself.
 - Positive example: shortest path
 - Counterexample: longest (simple) path
- DP relies on the principle of optimality
 - The optimal solution to any nontrivial instance of a problem is a combination of optimal solutions to some of its sub-instances.
 - It is often not obvious which sub-instances are relevant to the instance under consideration.

Principle of optimality

- Given an optimal sequence of decisions, each subsequence must be optimal by itself.

- Positive example: shortest path
- Counterexample: longest (simple) path

- DP relies on the principle of optimality

Optimal
Substructure

- The optimal solution to any nontrivial instance of a problem is a combination of optimal solutions to some of its sub-instances.

- It is often not obvious which sub-instances are relevant to the instance under consideration.

Elements of Dynamic Programming

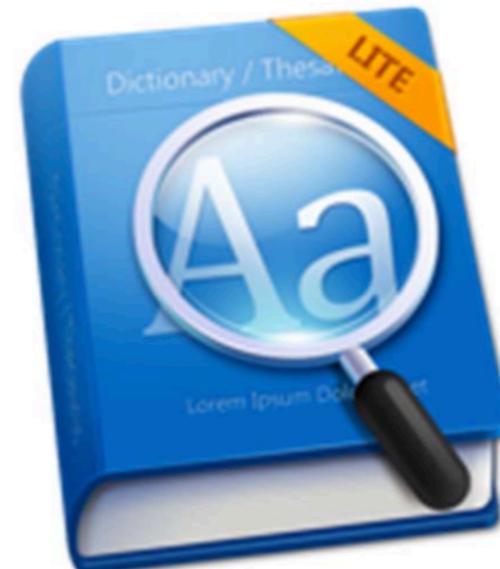
- Symptoms of DP

- Overlapping subproblems
- Optimal substructure

DP Dictionary

- How to use DP

- “Brute force” recursion
 - Overlapping subproblems
- “Smart” programming
 - Topological ordering of subproblems



Thank you!
Q & A