Dynamic Programming - II

Algorithm : Design & Analysis

[17]

In the last class…

- Recursion and Subproblem Graph
- Basic Idea of Dynamic Programming
- Least Cost of Matrix Multiplication
- Extracting Optimal Multiplication Order

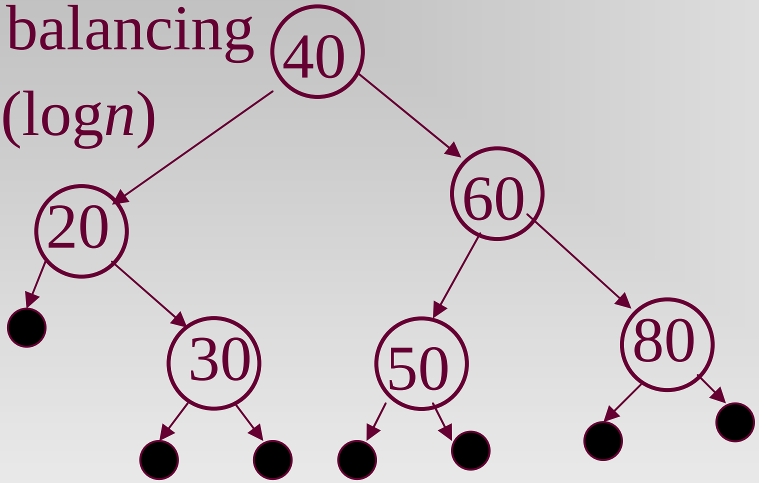
Dynamic Programming - II

- Optimal Binary Search Tree
- Separating Sequence of Word
- Changing Coins
- Dynamic Programming Algorithms

Binary Search Tree

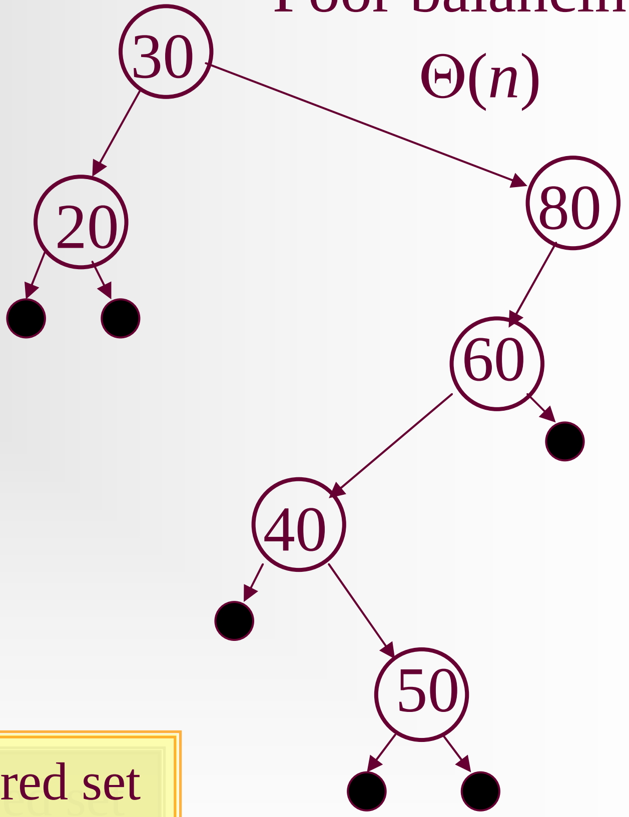
Good balancing

$\Theta(\log n)$



Poor balancing

$\Theta(n)$

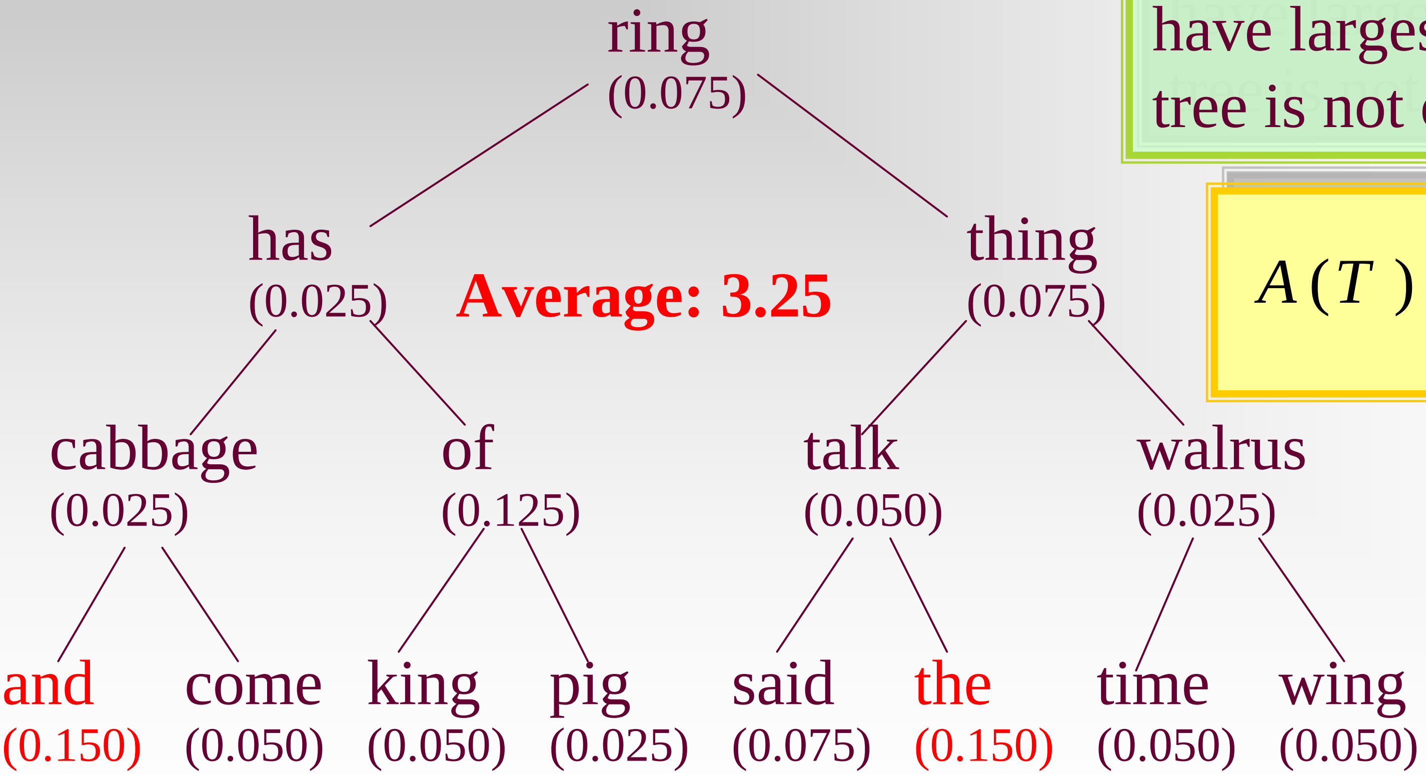


In a properly drawn tree, pushing forward to get the ordered list.

- Each node has a key, belonging to a linear ordered set
- An inorder traversal produces a sorted list of the keys

Keys with Different Frequencies

A binary search tree perfectly balanced



ring (0.075)

has (0.025)

thing (0.075)

cabbage (0.025)

of (0.125)

talk (0.050)

walrus (0.025)

and (0.150)

come (0.050)

king (0.050)

pig (0.025)

said (0.075)

the (0.150)

time (0.050)

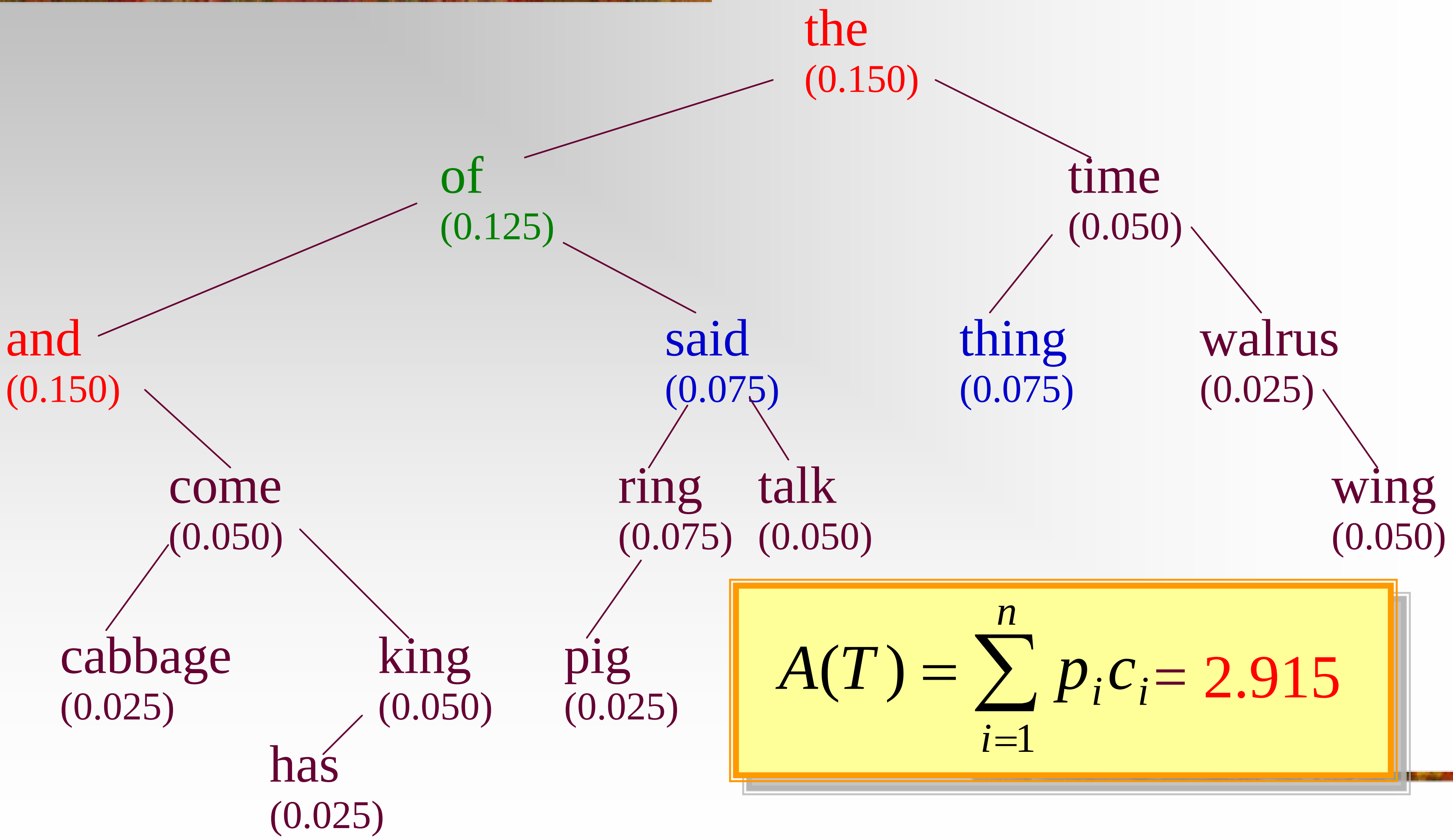
wing (0.050)

Average: 3.25

Since the keys with largest frequencies have largest depth, this tree is not optimal.

$$A(T) = \sum_{i=1}^{n} p_i c_i$$

Improved for a Better Average



the
(0.150)

of
(0.125)

time
(0.050)

and
(0.150)

said
(0.075)

thing
(0.075)

walrus
(0.025)

come
(0.050)

ring
(0.075)

talk
(0.050)

wing
(0.050)

cabbage
(0.025)

king
(0.050)

pig
(0.025)

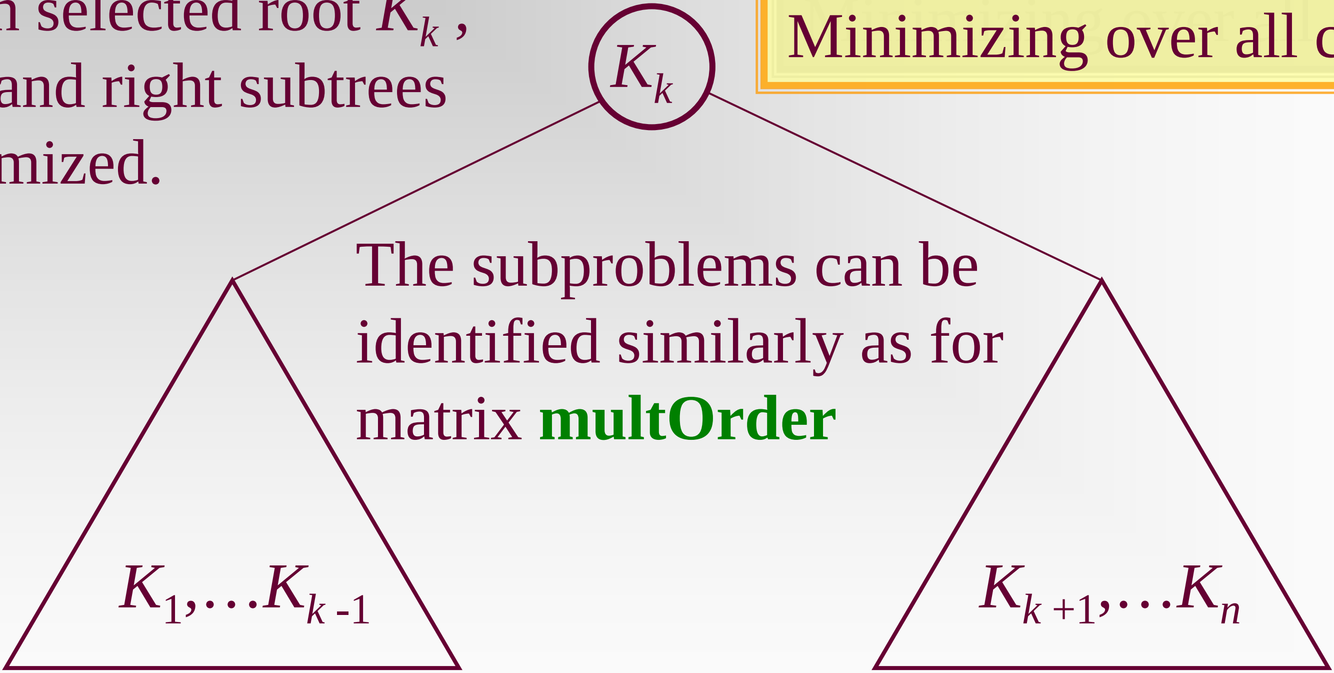
has
(0.025)

$$A(T) = \sum_{i=1}^{n} p_i c_i = 2.915$$

Plan of Optimal Binary Tree

The problem is decomposes by the choices of the root. Minimizing over all choices

For each selected root K_k , the left and right subtrees are optimized.



The subproblems can be identified similarly as for matrix **multOrder**

Subproblems as left and right subtrees

Problem Rephrased

- Subproblem identification
  - The keys are in sorted order.
  - Each subproblem can be identified as a pair of index (low, high)
- Expected solution of the subproblem
  - For each key K_i, a weight p_i is associated.

    Note: p_i is the probability that the key is searched for.
  - The subproblem (low, high) is to find the binary search tree with *minimum weighted retrieval cost*.

Minimum Weighted Retrieval Cost

- A(low, high, r) is the minimum weighted retrieval cost for subproblem (low, high) when K_r is chosen as the root of its binary search tree.
- A(low, high) is the minimum weighted retrieval cost for subproblem (low, high) over all choices of the root key.
- p(low, high), equal to $p_{\text{low}}+p_{\text{low+1}}+\ldots+p_{\text{high}}$, is the weight of the subproblem (low, high).

Note: p(low, high) is the probability that the key searched for is in this interval .

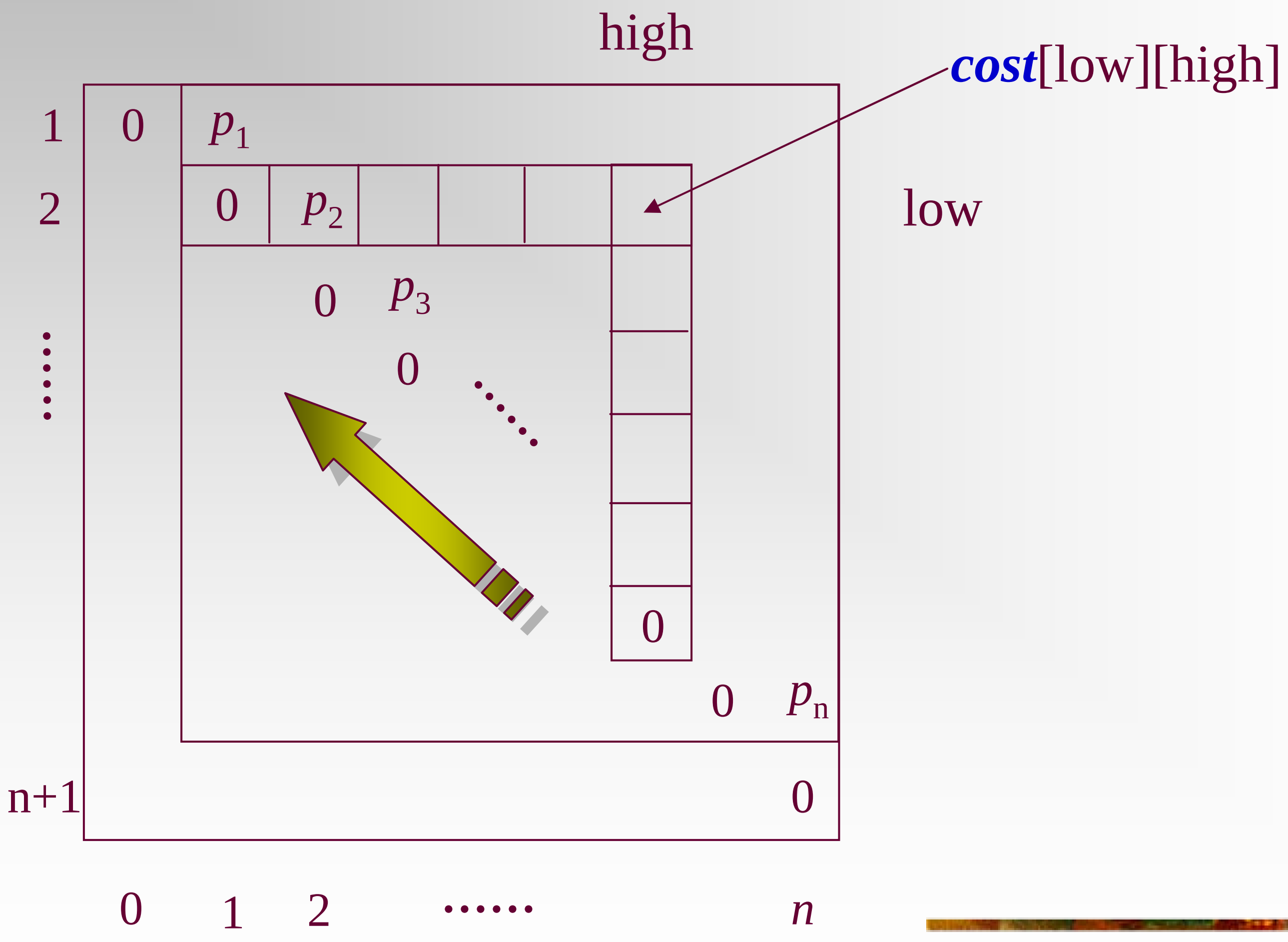
Integrating Solutions of Subproblem

- Weighted retrieval cost of a subtree
  - Let T is a particular tree containing $K_{\text{low}}, \ldots, K_{\text{high}}$, the weighted retrieval cost of T is W, with T being a whole tree. Then, as a subtree with the root at level 1, the weighted retrieval cost of T will be: $\boldsymbol{W+p(\text{low, high})}$
- So, the recursive relations:
  - $A(\text{low, high}, r)$
    $= p_r+p(\text{low}, r\text{-}1)+A(\text{low}, r\text{-}1)+p(r+1, \text{high})+A(r+1, \text{high})$
    $= p(\text{low, high})+A(\text{low}, r\text{-}1)+A(r+1, \text{high})$
  - $A(\text{low, high}) = \min\{A(\text{low, high}, r) \mid \text{low} \leq r \leq \text{high}\}$

Avoiding Repeated Work by Storing

- Array *cost*: *cost*[low][high] gives the minimum weighted search cost of subproblem (low,high).
- Array *root*: *root*[low][high] gives the best choice of root for subproblem (low,high)
- The *cost*[low][high] depends upon subproblems with higher first index(row number) and lower second index(column number)

Computation of the Array *cost*



Optimal BST: DP Algorithm

```
bestChoice(prob, cost, r
    if (high<low)
        bestCost=0;
        bestRoot=-1;
    else
        bestCost=∞;
    for (r=low; r≤high; r++)
        rCost=p(low,high)+cost[low][r-1]+cost[r+1][high];
        if (rCost<bestCost)
            bestCost=rCost;
            bestRoot=r;
        cost[low][high]=bestCost;
        root[low][high]=bestRoot;
    return
```

```
optimalBST(prob,n,cost,root)
    for (low=n+1; low≥1; low--)
        for (high=low-1; high≤n; high++)
            bestChoice(prob,cost,root,low,high)
    return cost
```

in $\Theta(n^3)$

Separating Sequence of Words

- Word-length $w_1, w_2, \ldots, w_n$ and line-width: W
- Basic constraint: if $w_\mathrm{i}, w_{\mathrm{i+1}}, \ldots, w_\mathrm{j}$ are in one line, then $w_\mathrm{i}+w_{\mathrm{i+1}}+ \ldots+w_\mathrm{j}\leq W$
- Penalty for one line: some function of X. X is:
  - 0 for the last line in a paragraph, and
  - $W-(w_\mathrm{i}+w_{\mathrm{i+1}}+ \ldots+w_\mathrm{j})$ for other lines
- The problem
  - how to separate a sequence of words(forming a paragraph) into lines, making the penalty of the paragraph, which is the sum of the penalties of individual lines, minimized.

Solution by Greedy Strategy

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 a edge to about j –i other vertices, so, the number of edges is in $\Theta(n^3)$

Simpler Identification of subproblem

- If a subproblem concludes the paragraph, then (k,n) can be simplified as (k). There are about k subproblems like this.
- 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)

Breaking Sequence into lines

```
lineBreak($w,W,i,n,L$)
    **if** ($w_i + w_{i+1} + \ldots + w_n \leq W$)
        <Put all words on line $L$, set penalty to 0>
    **else**
        **for** ($k=1$; $w_i+\ldots+w_{i+k-1} \leq W$; $k$++)
            $X=W-(w_i+\ldots+w_{i+k-1})$;
            kPenalty=lineCost($X$)+lineBreak($w,W, i+k, n, L+1$)
            <Set penalty always to the minimum kPenalty>
            <Updating $k_{min}$, which records the $k$ that produced
                                    the minimum penalty>
        <Put words $i$ through $i+k_{min}$-1 on line $L$>
    **return** penalty
```

In *DP* version this is replaced by “**Recursion or Retrieve**”

In DP version, “**Storing**” inserted

Analysis of lineBreak

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

Making Change: Revisited

- Problem: with certain systems of coinage, how to pay a given amount using the smallest possible number of coins

- We have known that the greedy strategy fails sometimes

Subproblems

- Suppose the currency we are using has available coins of *n* different denotations, and a coin of denomination *i* has d_i units. The amount to be paid is *N*.
- One subproblem can be represented as [*i*,*j*], for which the result is the minimum number of coins required to pay an amount of *j* units, using only coins of denominations 1 to *i*.
- The solution of the problem of making change is the result of 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: $c[i\text{-}1, j]$
  - One coins of denomination i is used: $1+c[i, j\text{-}d_i]$
- So, $c[i,j] = \min\,(c[i\text{-}1, j],1+c[i, j\text{-}d_i])$

Data Structure

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

an example

$$
\begin{array}{c|ccccccccc|}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
d_1=1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
d_2=4 & 0 & 1 & 2 & 3 & 1 & 2 & 3 & 4 & 2 \\
d_3=6 & 0 & 1 & 2 & 3 & 1 & 2 & 1 & 2 & 2
\end{array}
$$

direction of computation

The Procedure

```
int coinChange(int N, int n, int[] coin)
    int denomination[]=[$d_1,d_2,…,d_n$];
    for (i=1; i≤n; i++)
        coin[i,0]=0;
    for (i=1; i≤n; i++)
        for (j=1; i≤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

In the last class…

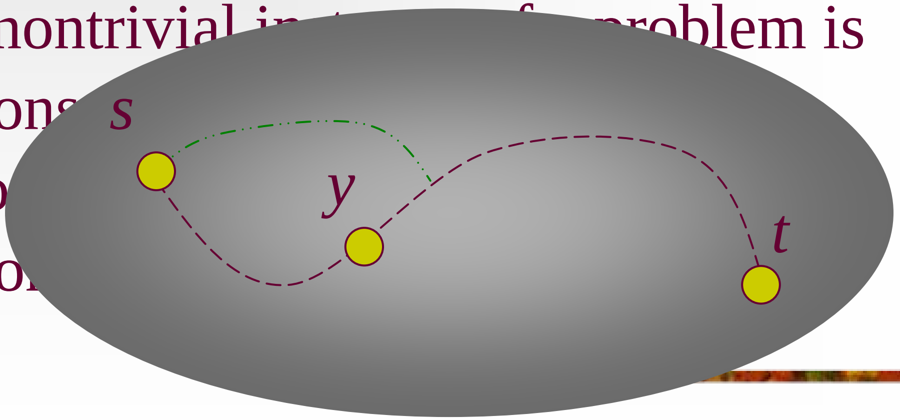
- Recursion and Subproblem Graph
- Basic Idea of Dynamic Programming
- Least Cost of Matrix Multiplication
- Extracting Optimal Multiplication Order

In the last class…

- Recursion and Subproblem Graph
- Basic Idea of Dynamic Programming
- Least Cost of Matrix Multiplication
- Extracting Optimal Multiplication Order

Principle of Optimality

- ***Given an optimal sequence of decisions, each subsequence must be optimal by itself.***
    - Positive example: shortest path
    - Counterexample: longest (simple) path
- Usually, dynamic programming may be used where the principle of optimality applies.
- So, the optimal solution to any nontrivial in f problem is a combination of optimal solution However, it is not usually obvio ***relevant*** to the instance under co



Home Assignments

- pp.477-
  - 10.10
  - 10.11
  - 10.12
  - 10.14
  - 10.15