# 计算机问题求解 – 论题2-12

- 动态规划

课程研讨

- TC第15章

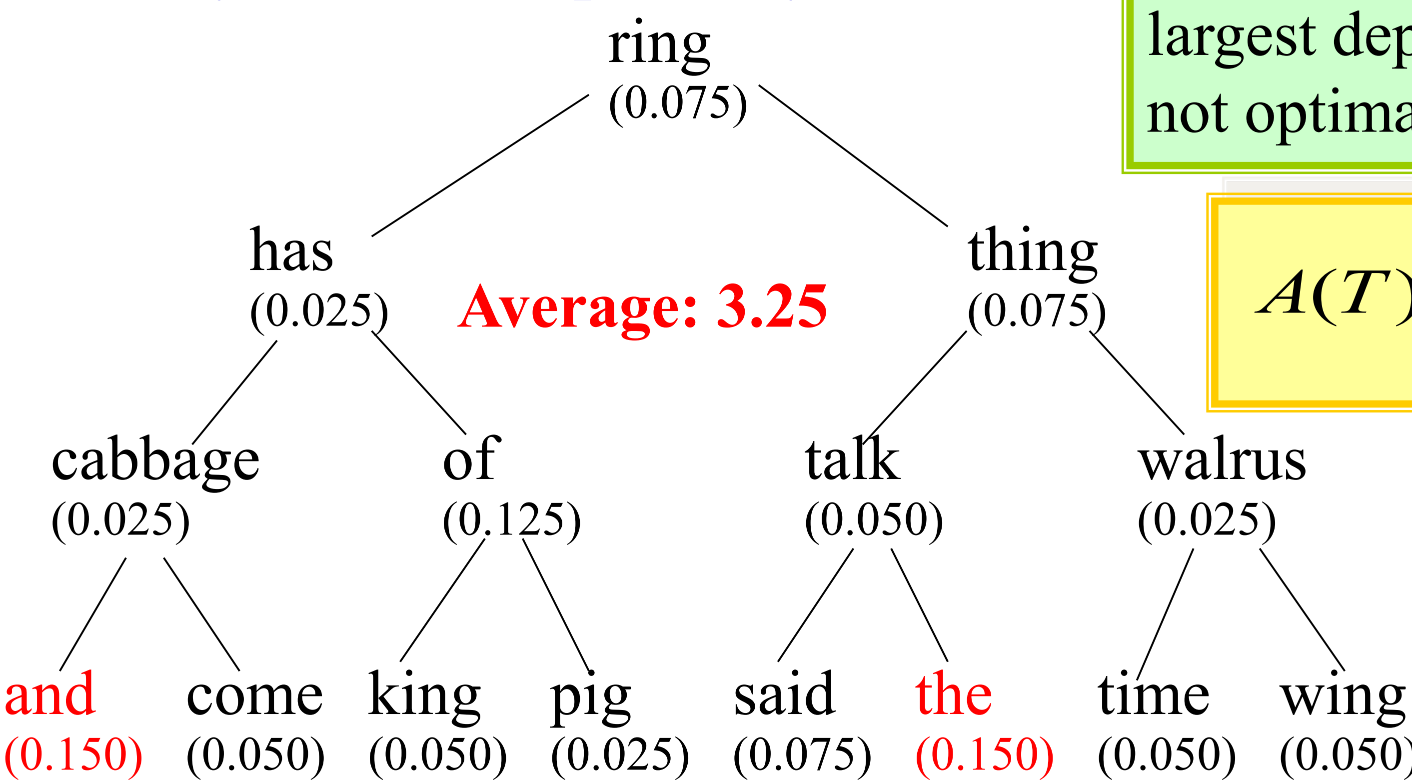
# 问题0：dynamic programming的基本概念

- 什么样的问题可以使用dynamic programming来求解？它高效的根本原因是什么？付出了什么代价？
- 你理解dynamic programming的四个步骤了吗？
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution.
  4. Construct an optimal solution from computed information.

- 广义上决定dynamic programming运行时间的要素是哪两点？
- top-down with memorization和bottom-up method哪个更快？

# 问题1：Keys with Different Frequencies

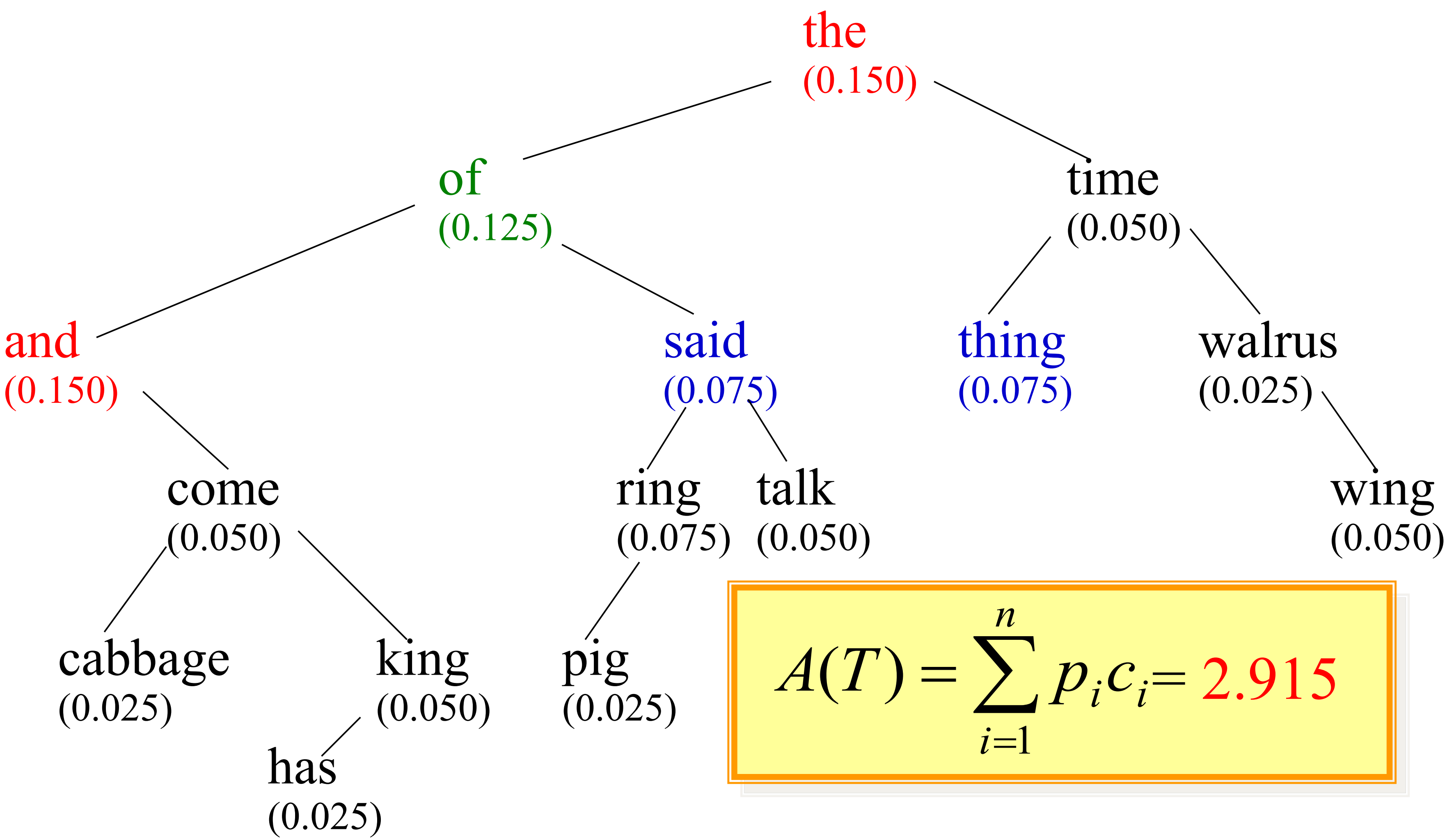
**A binary search tree perfectly balanced**




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




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

# 矩阵连乘的问题

- 需要完成的任务:

  求乘积: $A_1 \times A_2 \times \ldots \times A_{n-1} \times A_n$

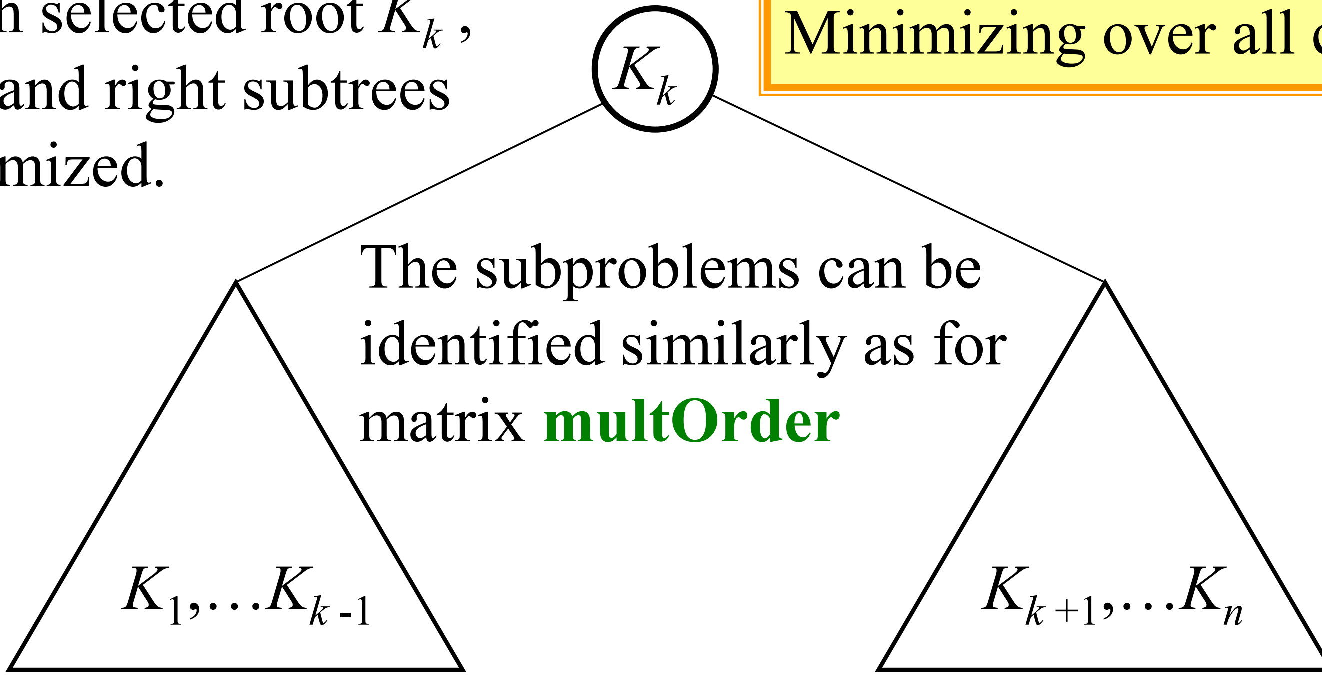
  $A_i$ 是二维矩阵，一般不是方阵，大小符合乘法规定的要求。

- 为什么会成为问题:
  - 矩阵乘法满足结合律，因此我们可以任意指定运算顺序;
  - 而不同的计算顺序代价差别很大。
- 优化问题: **什么样的次序计算代价最小?**

# 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 .

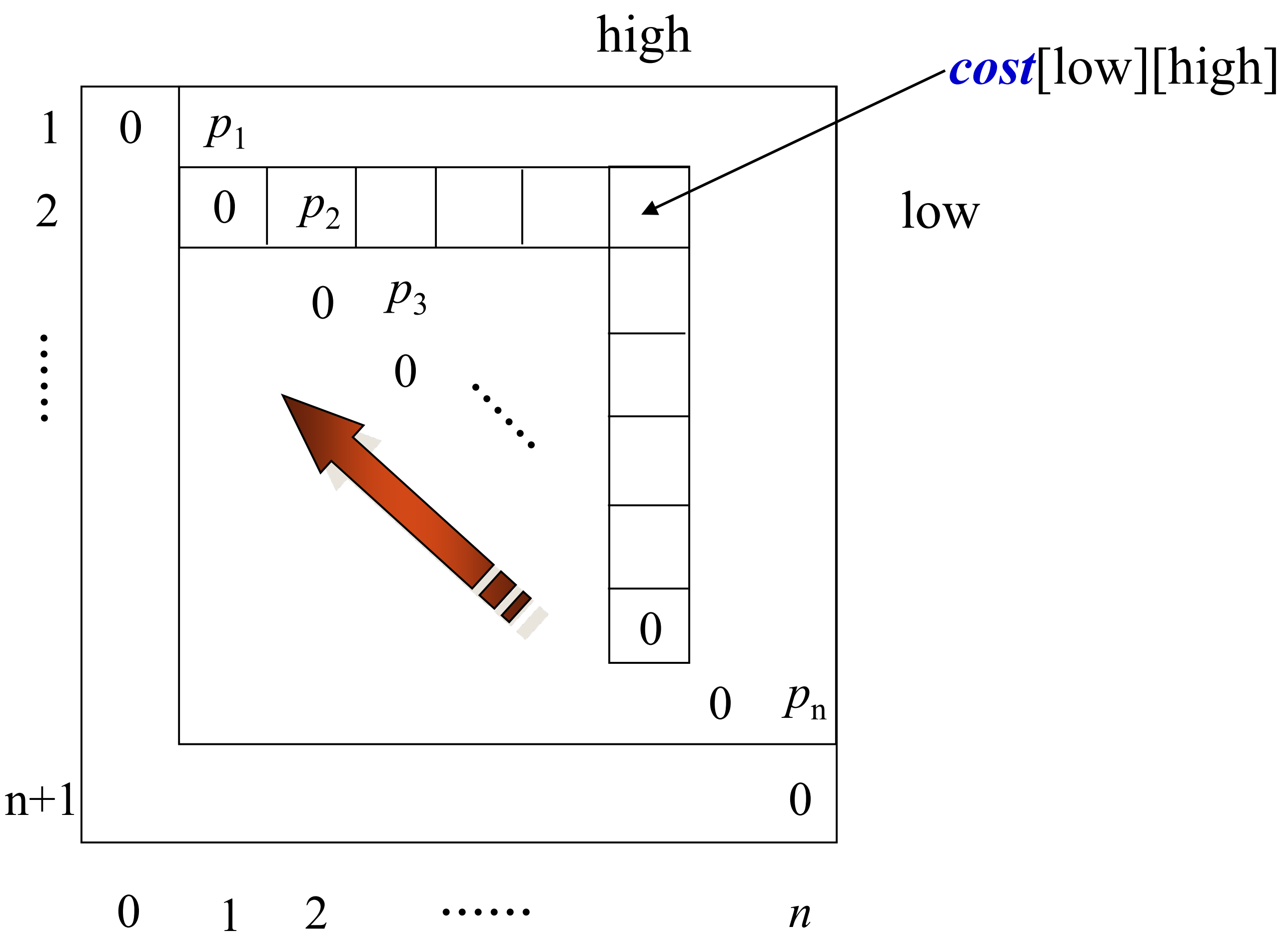
# Integrating Solutions of Subproblem

- Weighted retrieval cost of a subtree
  - Let $T$ is a particular tree containing $K_{low}, \ldots, K_{high}$, 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: ***W+p*(low, high)**
- So, the recursive relations:
  - $A(\text{low}, \text{high}, r)$
    $= p_r + p(\text{low}, r\text{-}1) + A(\text{low}, r\text{-}1) + p(r+1, \text{high}) + A(r+1, \text{high})$
    $= p(\text{low}, \text{high}) + A(\text{low}, r\text{-}1) + A(r+1, \text{high})$
  - $A(\text{low}, \text{high}) = \min\{A(\text{low}, \text{high}, r) \mid \text{low} \le r \le \text{high}\}$

# 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, root, low, high)
    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 $\Theta(n^3)$**

# 问题2：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 an 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 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**"

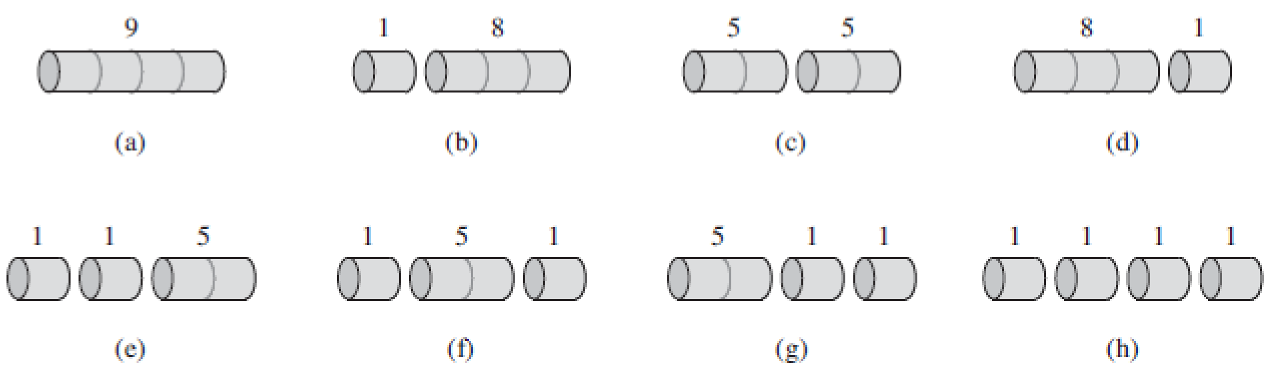
In DP version, "**Storing**" inserted

# 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 $\boldsymbol{\Theta(n)}$.

# 问题2：dynamic programming的实例

- 你能说明求解rod cutting的四个步骤吗？
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution.
  4. Construct an optimal solution from computed information.




| length $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| price $p_i$ | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 | 30 |

# 问题3：Longest Common Subsequence

- 你能说明求解longest common subsequence的四个步骤吗？
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution.
  4. Construct an optimal solution from computed information.

$$S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$$

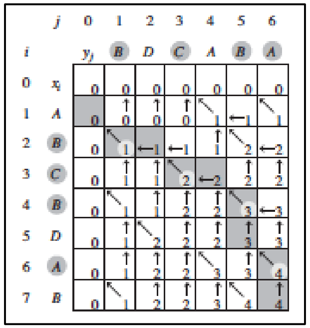
$$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$$

GTCGTCGGAAGCCGGCCGAA

# 问题3：Longest Common Subsequence

- 你能说明求解longest common subsequence的四个步骤吗？
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution.
  4. Construct an optimal solution from computed information.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$



$$S_1 = \texttt{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$$

$$S_2 = \texttt{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$$

GTCGTCGGAAGCCGGCCGAA

# 问题4：Longest palindrome subsequence

- 你能说明求解longest palindrome subsequence的四个步骤吗？
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution.
  4. Construct an optimal solution from computed information.

A ***palindrome*** is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).

Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`.

# 问题4：Longest palindrome subsequence

- 你能说明求解longest palindrome subsequence的四个步骤吗？
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution.
  4. Construct an optimal solution from computed information.

```
longest(i,j)= j-i+1 if j-i<=0,
              2+longest(i+1,j-1) if x[i]==x[j]
              max(longest(i+1,j),longest(i,j-1)) otherwise
```

A *palindrome* is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).

Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`.

# 问题5：Edit distance

- 你能说明求解edit distance的四个步骤吗？
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution.
  4. Construct an optimal solution from computed information.

**Insertion** of a single symbol. If $a = uv$, then inserting the symbol $x$ produces $uxv$. This can also be denoted $\varepsilon \rightarrow x$, using $\varepsilon$ to denote the empty string.

**Deletion** of a single symbol changes $uxv$ to $uv$ ($x \rightarrow \varepsilon$).

**Substitution** of a single symbol $x$ for a symbol $y \neq x$ changes $uxv$ to $uyv$ ($x \rightarrow y$).

The Levenshtein distance between "kitten" and "sitting" is 3. The minimal edit script that transforms the former into the latter is:

1. **k**itten → **s**itten (substitution of "s" for "k")
2. sitt**e**n → sitt**i**n (substitution of "i" for "e")
3. sittin → sittin**g** (insertion of "g" at the end).

# 问题5： Edit distance

- 你能说明求解edit distance的四个步骤吗？
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution.
  4. Construct an optimal solution from computed information.

$$d_{ij} = \begin{cases} d_{i-1,j-1} & \text{for } a_j = b_i \\ \min \begin{cases} d_{i-1,j} + w_{\text{del}}(b_i) \\ d_{i,j-1} + w_{\text{ins}}(a_j) \\ d_{i-1,j-1} + w_{\text{sub}}(a_j, b_i) \end{cases} & \text{for } a_j \neq b_i \end{cases} \quad \text{for } 1 \leq i \leq m, 1 \leq j \leq n.$$

**Insertion** of a single symbol. If $a = uv$, then inserting the symbol $x$ produces $uxv$. This can also be denoted $\varepsilon \to x$, using $\varepsilon$ to denote the empty string.

**Deletion** of a single symbol changes $uxv$ to $uv$ ($x \to \varepsilon$).

**Substitution** of a single symbol $x$ for a symbol $y \neq x$ changes $uxv$ to $uyv$ ($x \to y$).

The Levenshtein distance between "kitten" and "sitting" is 3. The minimal edit script that transforms the former into the latter is:

1. **k**itten → **s**itten (substitution of "s" for "k")
2. sitt**e**n → sitt**i**n (substitution of "i" for "e")
3. sittin → sittin**g** (insertion of "g" at the end).

# 问题6： Subset Sum

- Given a set $A=\{s_1,s_2,\ldots, s_n\}$, where $s_i$ (for $i=1,2,\ldots, n$) is a natural number, and a natural number $S$, determine whether there is a subset of $A$ totaling exactly $S$. Design a dynamic programming algorithm for solving the problem.

# Decomposition the Problem

- Suppose subset $A_i \in A$ is a solution of the problem and $s_j \in A_i$, then we have
$$\sum(A_i - \{s_j\}) = S - s_j$$
- Thus, the problem can be divided into several stages, in each of which one element is found.
- States: all the possible values of subset sum in each stages.

# Basic Idea

- Using a two-dimension boolean table $T$, in which $T[i, j]$=**true** if and only if there is a subset of the first $i$ items of $A$ totaling exactly $j$.

- Initialization
  - For each elements in $T[n+1][S]$, set as false
- Main loop to calculate each value

# Main loop

```
for (i = 0; i <= n +1; i ++)
   if (A[i] == S ) return true;
   else if (A[i] < S) T[i, A[i]] = true;
   for (j = 0; j <= S + 1; j ++)
      if (T[i -1, j] )
      {
        T[i, j] = true;
        if ((T[i, j] + A[i]) == S) return true;
        else if ((t = (T[i, j] + A[i])) < S) T[i, t] = true;
      }
return false;
```

Time $O(nS)$   Space $O(nS)$

# 问题7：dynamic programming的实例 (续)

- unweighted longest simple path为什么不具有最优子结构？
- unweighted shortest simple path为什么不存在这个问题？