Introduction to

Algorithm Design and Analysis

[16] Dynamic Programming 1

Yu Huang

http://cs.nju.edu.cn/yuhuang
Institute of Computer Software
Nanjing University

In the last class…

- **Single-source shortest paths**
  - From BFS to the Dijkstra algorithm

- **All-pairs shortest paths**
  - BF1, BF2, BF3
 - Floyd-Warshall algorithm

Dynamic Programming

- **Basic Idea of Dynamic Programming (DP)**
  - o Smart scheduling of subproblems

- **Minimum Cost Matrix Multiplication**
  - o BF1, BF2
  - o A DP solution

- **Weighted Binary Search Tree**
  - o The "same" DP with matrix multiplication

Brute Force Recursion

The F_n can be computed in linear time easily, but the cost for recursion may be exponential.

The number of activation frames are $2F_{n+1}-1$
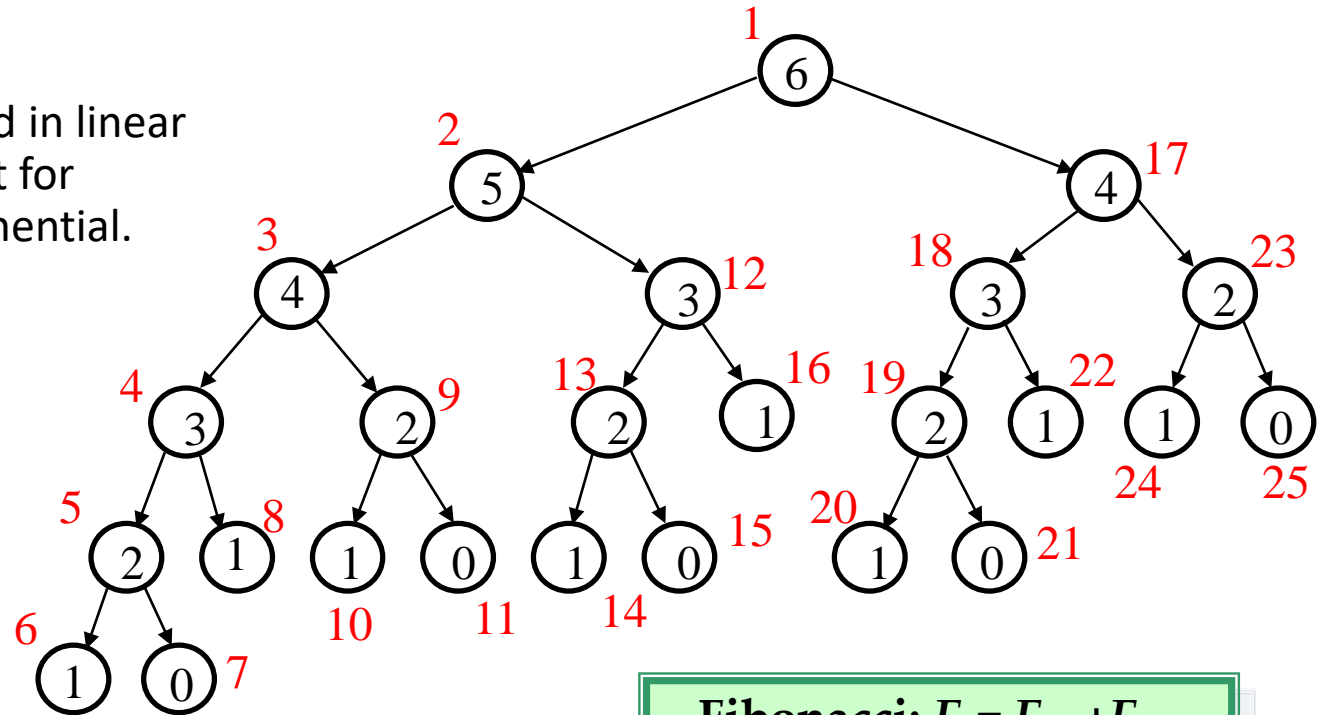

Fibonacci: $F_n = F_{n-1} + F_{n-2}$

For your reference

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 35, ...

Subproblem Graph

- **The subproblem graph for a recursive algorithm *A* of some problem is defined as:**
  - vertex: the instance of the problem
  - directed edge: I→J if and only if when *A* invoked on I, it makes a recursive call directly on instance J.

- **Portion A(*P*) of the subproblem graph for Fibonacci function:** **here is fib(6)**
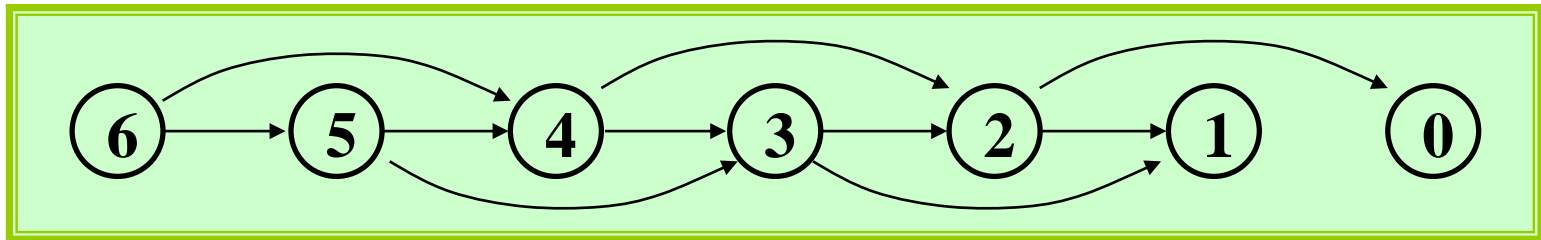
Properties of Subproblem Graph

- **If A always terminates, the subproblem graph for A is a DAG.**

  - For each path in the tree of activation frames of a particular call of A, A(P), there is a corresponding path in the subproblem graph of A connecting vertex P and a base-case vertex.

  - The subproblem graph can be viewed as a dependency graph of subtasks to be solved.

- **A top-level recursive computation traverse the entire subproblem graph in some memoryless style.**

Basic Idea of DP

- **Smart recursion**
  - Compute each subproblem **only once**

- **Basic process of a "smart" recursion**
  - Find a reverse topological order for the subproblem graph
    - In most cases, the order can be determined by particular knowledge of the problem.
    - General method based on DFS is available
  - Scheduling the subproblems according to the reverse topological order
  - Record the subproblem solutions in a dictionary

Recursion by DP

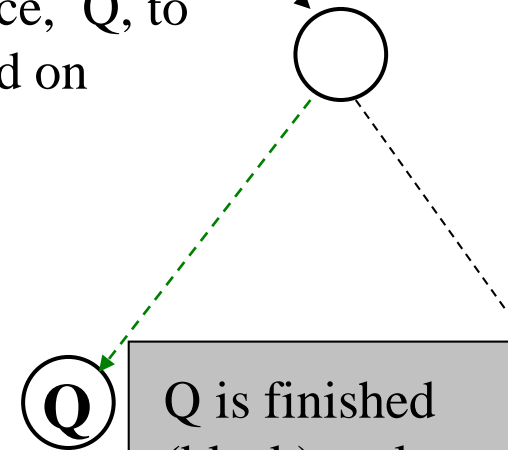## Case 1: White Q

a instance, Q, to be called on
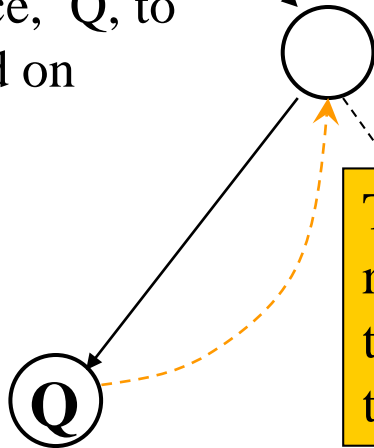
To backtracking, record the result into the dictionary (Q, turned black)

Q is undiscovered (white), go ahead with the recursive call

Note: for DAG, no gray vertex will be met

Case 2: Black Q

a instance, Q, to be called on

Q is finished (black), only "checking" the edge, retrieve the result from the dictionary

Fibonacci by DP

fibDPwrap(*n***)**

  Dict soln=create(*n***);**

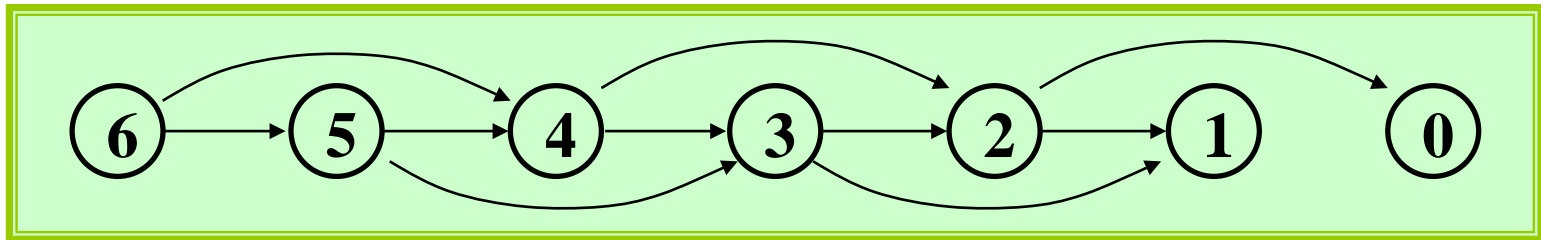**return fibDP(soln,***n***)**

This is the wrapper, which will contain processing existing in original recursive algorithm wrapper.

fibDP(soln,*k***)**
  int fib, f1, f2;
  if (*k*<**2) fib=***k***;**
  else
    if (member(soln, ***k*-1)==false)**
      f1=fibDP(soln, ***k*-1);**
    else
      f1= retrieve(soln, k-1);
    if (member(soln, ***k*-2)==false)**
      f2=fibDP(soln, ***k*-2);**
    else
      f2= retrieve(soln, k-2);
    fib=f1+f2;
  store(soln, ***k*, fib);**
return fib

Fibonacci by DP

- ## Subproblems
  - soln[0], soln[1], …, soln[n]

- ## Topology order to the subproblems
  - soln[0] = 0; soln[1] = 1;
  - for{i=2; i≤n; i++}{
        soln[i] = soln[i-1] + soln[i-2];
    } // end for

DP: New Concept Recursion

Many subproblems	DAG of subproblems
Scheduling of subproblems	Topological ordering
Storage/retrieval of subproblem results	Subproblem dictionary
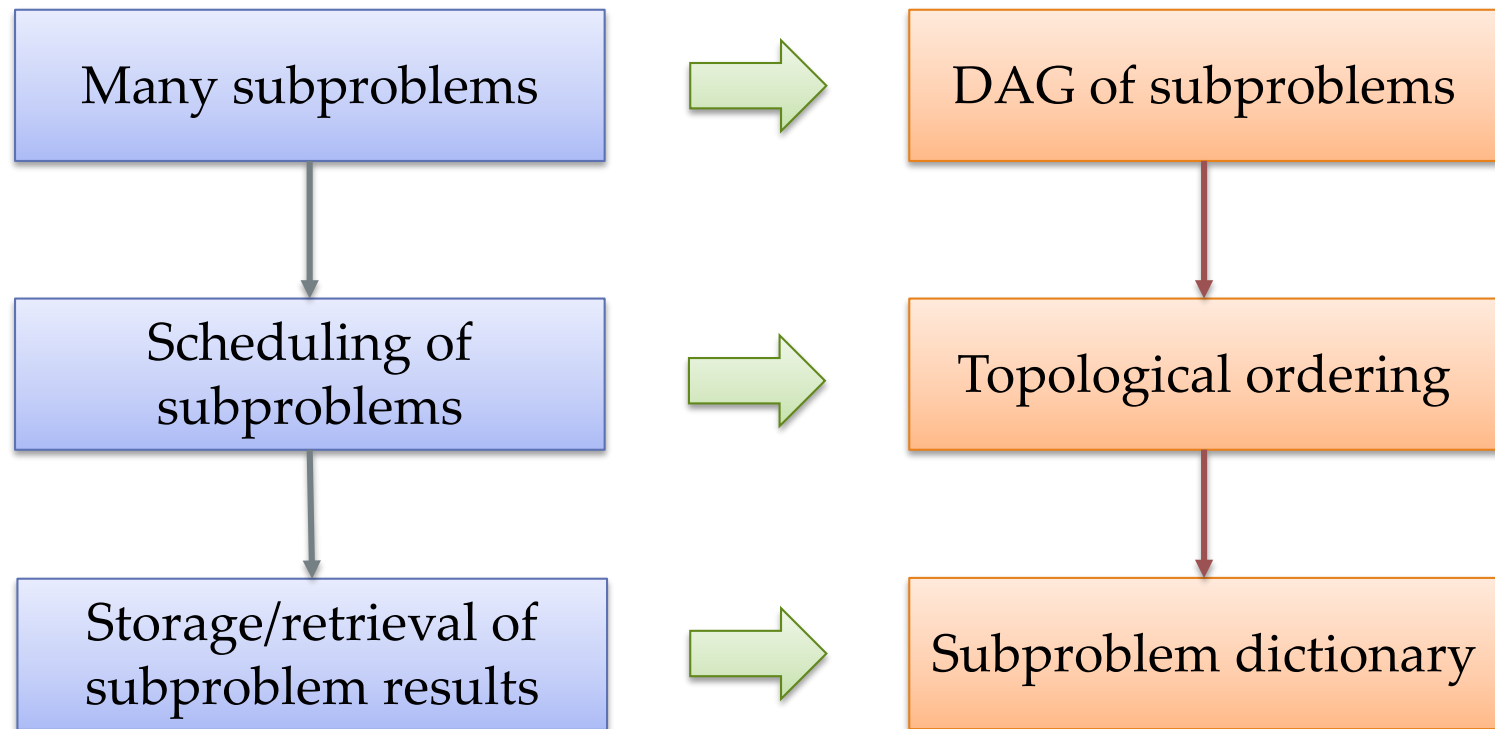
Matrix Multiplication Order Problem

- **The task:**

  Find the product: $A_1 \times A_2 \times \ldots \times A_{n-1} \times A_n$

  A_i is 2-dimentional array of different legal sizes

- **The issues:**
  - Matrix multiplication is associative
  - Different computing order results in great difference in the number of operations

- **The problem:**
  - Which is the best computing order

Cost for Matrix Multiplication

Let $C = A_{p \times q} \times B$

$$c_{i,j} = \sum_{k=1}^{q} a_{ik} b_{kj}$$ There are q multiplication

An example: $A_1 \times A_2 \times A_3 \times A_4$
        30×1 1×40 40×10 10×25

$((A_1 \times A_2) \times A_3) \times A_4$: 20700 multiplications

$A_1 \times (A_2 \times (A_3 \times A_4))$: 11750

$(A_1 \times A_2) \times (A_3 \times A_4)$: 41200

$A_1 \times ((A_2 \times A_3) \times A_4)$: 1400

C has $p \times r$ elements as $c_{i,j}$

So, pqr multiplications altogether

Looking for a Greedy Solution

- **Strategy 1: "cheapest multiplication first"**
  - Success: $A_{30\times1}\times((A_{1\times40}\times A_{40\times10})\times A_{10\times25}$
  - Fail: $(A_{4\times1}\times A_{1\times100})\times A_{100\times5}$

- **Strategy 2: "largest dimension first"**

  - Correct for the second example above

  - Fail: $A_{1\times10}\times A_{10\times10}\times A_{10\times2}$

Intuitive Solution

- **Matrices: A_1, A_2, …, A_n**

- **Dimension: dim: d_0, d_1, d_2, …, d_{n-1}, d_n, for A_i is $d_{i-1} \times d_i$**

- **Sub-problem: seq: s_0, s_1, s_2, …, s_{len-1}, s_{len}, which means the multiplication of k matrices, with the dimensions: $d_{s0} \times d_{s1}$, $d_{s1} \times d_{s2}$, …**
  - Note: the original problem is: seq=$(0,1,2,…,n)$

Intuitive Solution

mmTry1(dim, len, seq)
  if (len<3) bestCost=0
  else
    bestCost=∞;
    for (i=1; i≤len-1; i++)
      c=cost of multiplication at position seq[i];
      newSeq=seq with *i*th element deleted;
      b=mmTry1(Dim, len-1, newSeq);
      bestCost=min(bestCost, b+c);
  return bestCost

> **Recursion on index sequence:**
> **(seq): 0, 1, 2, …, *n*** (len=n)
> with the kth matrix is A_k ($k \neq 0$) of the size $d_{k-1} \times d_k$,
> and the kth($k<n$) multiplication is $A_k \times A_{k+1}$.

$$T(n)=(n-1)T(n-1)+n, \quad \text{in } \Theta((n-1)!)$$

Subproblem Graph

- **key issue**
  - How can a subproblem be denoted using a **concise identifier**?
  - For mmTry1, the difficulty originates from the varied intervals in each newSeq.

- **If we look at the last (contrast to the first) multiplication, the two (not one) resulted subproblems are both contiguous subsequences, which can be uniquely determined by the pair:**

  <head-index, tail-index>

Improved Recursion

mmTry2(dim, low, high)
   if (high-low= =1) bestCost=0
   else
      bestCost=∞;
      for (k=low+1; k≤high-1; k++)
        a=mmTry2(dim, low, k);
        b=mmTry2(dim, k, high);
        c=cost of multiplication at position *k*;
        bestCost=min(bestCost, a+b+c);
   return bestCost

Only one matrix

with dimensions: dim[low], dim[k], and dim[high]

Still in $\Omega(2^n)$!

Smart Recursion by DP

- **DFS can traverse the subproblem graph in time O(n^3)**
    - At most $n^2/2$ vertices, as <i,j>, $0 \le i < j \le n$.
    - At most $2n$ edges leaving a vertex

```
mmTry2DP(dim, low, high, cost)
   ……
   for (k=low+1; k≤high-1; k++)
      if (member(low,k)==false) a=mmTry2(dim, low, k);
         else a=retrieve(cost, low, k);
      if (member(k,high)==false) b=mmTry2(dim, k, high);
         else b=retrieve(cost, k, high);
      ……
   store(cost, low, high, bestCost);
   return bestCost
```

Corresponding to the recursive procedure of DFS

Order of Computation
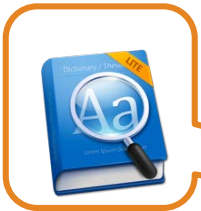
- **Dependency between subproblems**

DP dict

matrixOrder(n, cost, last)
- for (low=n-1; low≥1; low--)
- for (high=low+1; high≤n; high++)

Compute solution of subproblem (low, high) and store it in cost[low][high] and last[low][high]

- **return cost[0][n]**

Multiplication Order

- Input: array **dim** $=(d_0, d_1, \ldots, d_n)$, the dimension of the matrices.

- Output: array **multOrder**, of which the ith entry is the index of the ith multiplication in an optimum sequence.
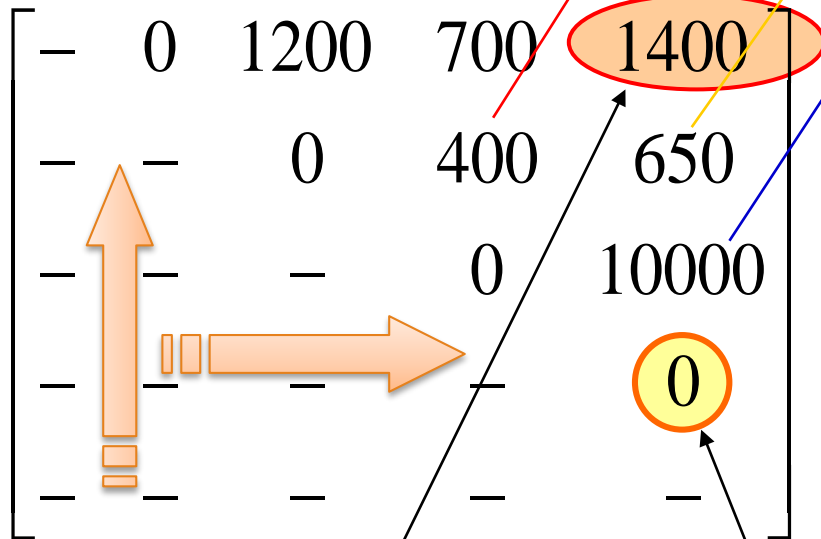
Using the stored results

```
float matrixOrder(int[] dim, int n, int[]
   multOrder)
<initialization of last,cost,bestcost,bestlast…>
  for (low=n-1; low≥1; low--)
    for (high=low+1; high≤n; high++)
      if (high-low==1) <base case>
      else bestcost=∞;

      for (k=low+1; k≤high-1; k++)
        a=cost[low][k];
        b=cost[k][high]
        c=multCost(dim[low], dim[k],
   dim[high]);
        if (a+b+c<bestCost)
          bestCost=a+b+c; bestLast=k;
    cost[low][high]=bestCost;
    last[low][high]=bestLast;
  extrctOrderWrap(n, last, multOrder)
  return cost[0][n]
```

An Example

- **Input: d_0=30, d_1=1, d_2=40, d_3=10, d_4=25**

cost **as finished**

$$\begin{bmatrix} - & 0 & 1200 & 700 & 1400 \\ - & - & 0 & 400 & 650 \\ - & - & - & 0 & 10000 \\ - & - & - & - & 0 \\ - & - & - & - & - \end{bmatrix}$$

Note: *cost*[i][j] is the least cost of $A_{i+1} \times A_{i+2} \times \dots A_j$.

For each selected k, retrieving:
- least cost of $A_{i+1} \times \dots \times A_k$.
- least cost of $A_{k+1} \times \dots \times A_j$.
and computing:
- cost of the last multiplication

First entry filled

Last entry filled
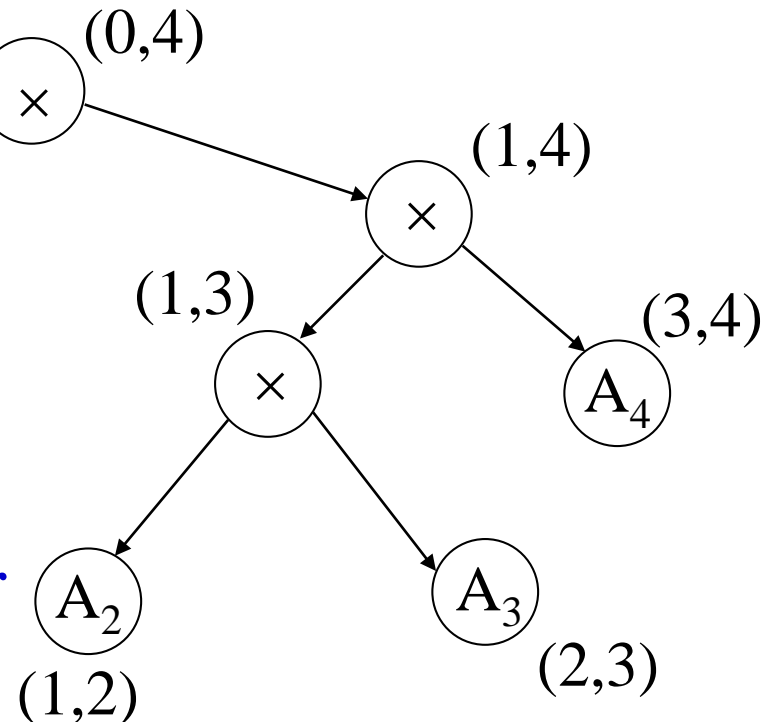
Arithmetic Expression Tree

- **Example input: $d_0=30$, $d_1=1$, $d_2=40$, $d_3=10$, $d_4=25$**

last **as finished**

As root

$$\begin{bmatrix} - & -1 & 1 & 1 & 1 \\ - & - & -1 & 2 & 3 \\ - & - & - & -1 & 3 \\ - & - & - & - & -1 \\ - & - & - & - & - \end{bmatrix}$$

Computing in post order

(0,4) $\times$

(1,4) $\times$

A$_1$ (0,1)

(3,4) A$_4$

(1,3) $\times$

A$_2$ (1,2)

A$_3$ (2,3)

Getting the Optimal Order

- **The core procedure is extractOrder, which fills the multiOrder array for subproblem (low,high), using informations in *last* array.**

```
extractOrder(low, high, last, multOrder)
    int k;
    if (high-low>1)
        k=last[low][high];               Just a post-order traversal
        extractOrder(low, k, last, multOrder);
        extractOrder(k, high, last, multOrder);
        multOrder[multOrderNext]=k;
        multOrderNext++;
```
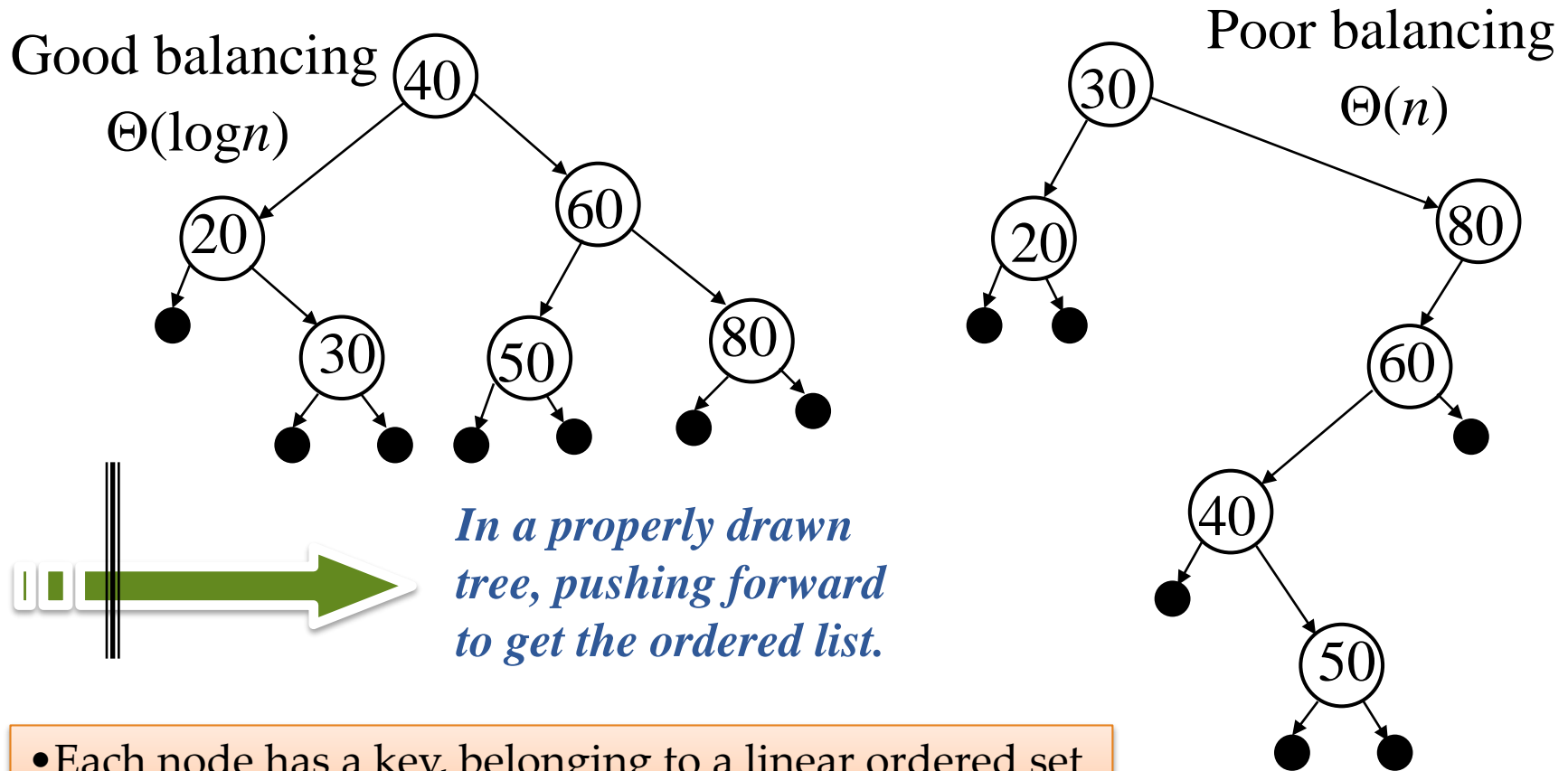
initialized in the wrapper

Analysis of matrixOrder

- **Main body: 3 layer of loops**
  - Time: the innermost processing costs constant, which is executed $\Theta(n^3)$ times.
  - Space: extra space for *cost* and *last*, both in $\Theta(n^2)$
- **Order extracting**
  - There are $2n$-1 nodes in the arithmetic-expression tree. For each node, extractOrder is called once. Since non-recursive cost for extractOrder is constant, so, the complexity of extractOrder is in $\Theta(n)$

Binary Search Tree

Good balancing

$\Theta(\log n)$

Poor balancing

$\Theta(n)$
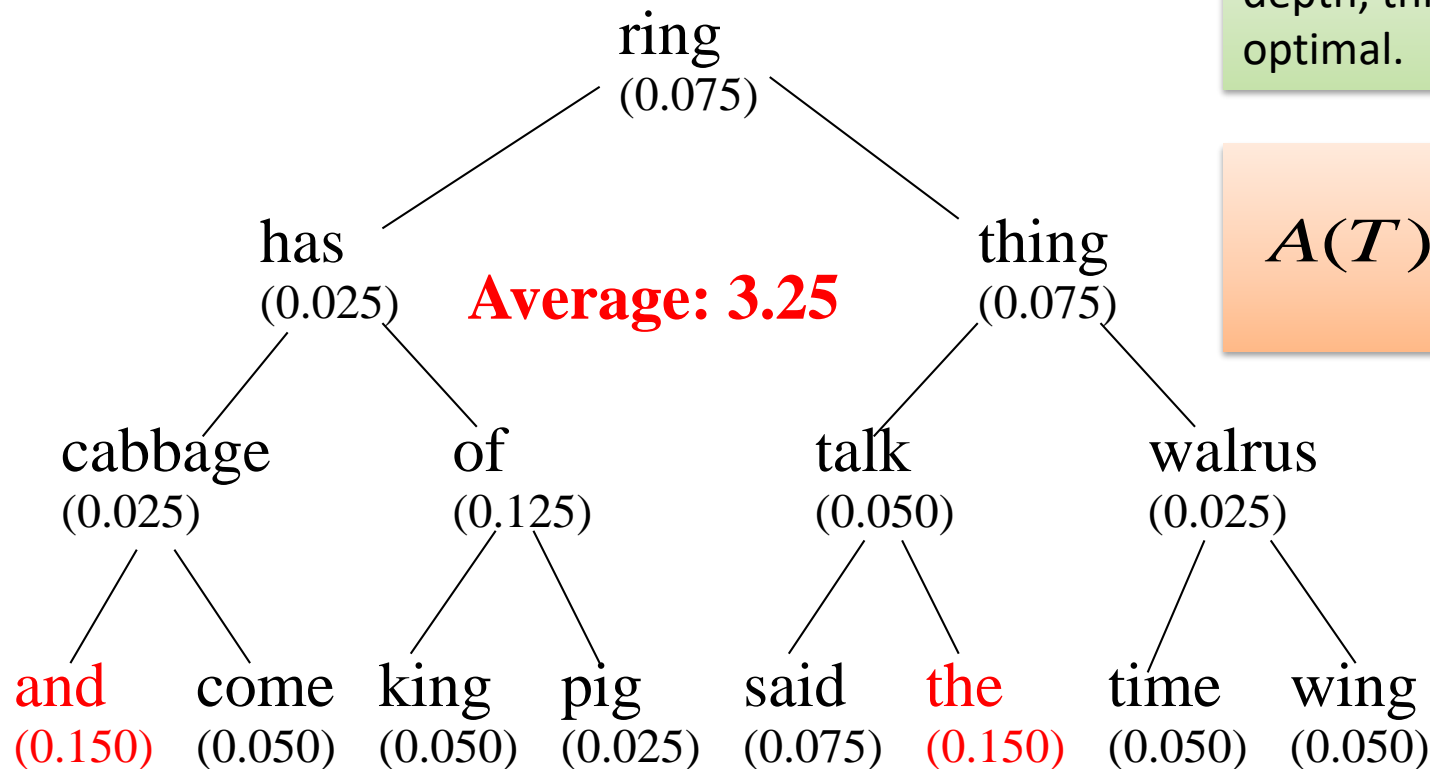
In a properly drawn tree, pushing forward to get the ordered list.

- 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
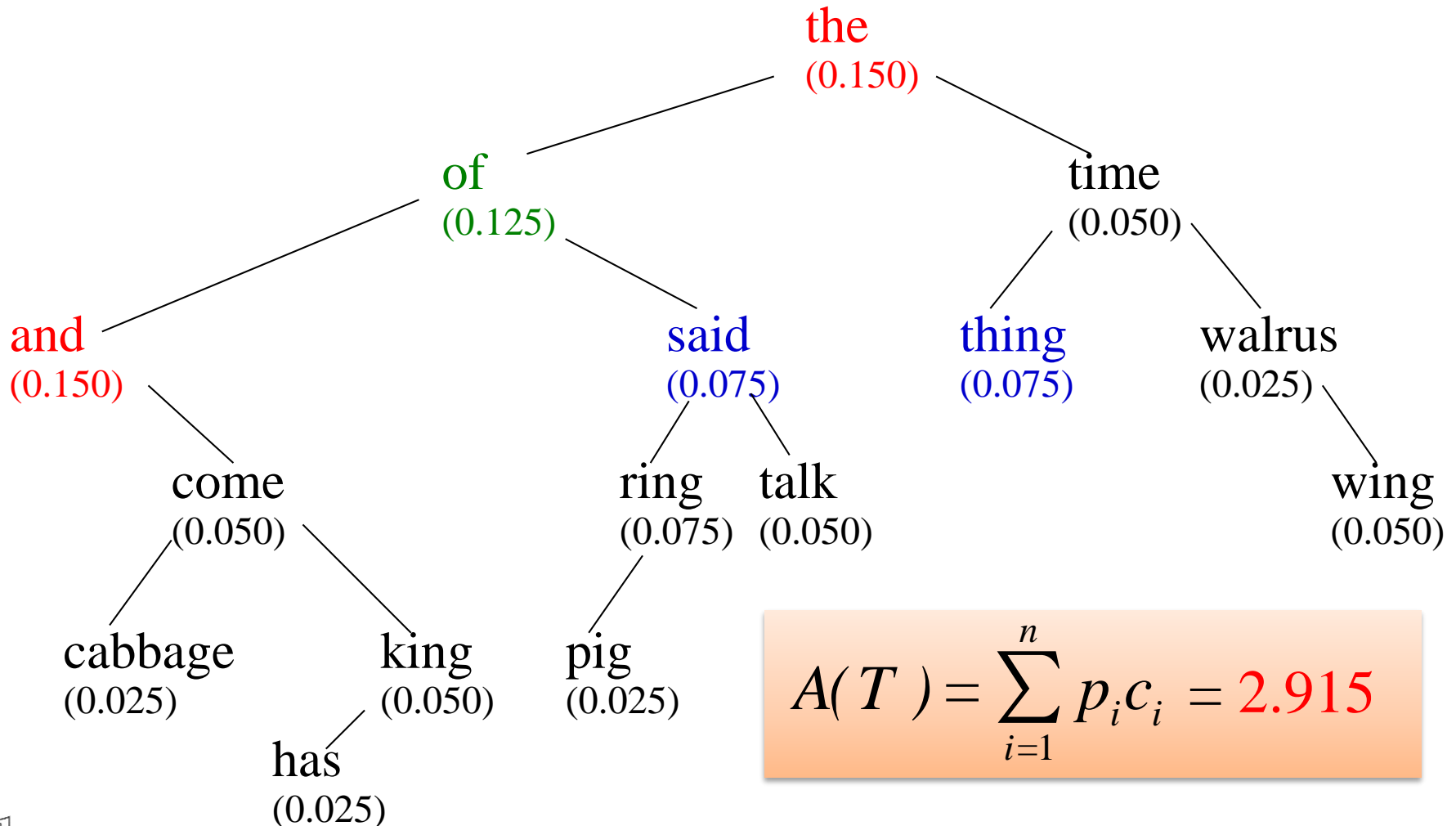
A binary search tree perfectly balanced

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

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

Average: 3.25

```
                    ring
                   (0.075)

        has                      thing
      (0.025)                   (0.075)

 cabbage     of          talk        walrus
 (0.025)   (0.125)     (0.050)       (0.025)

and  come  king  pig  said  the   time   wing
(0.150)(0.050)(0.050)(0.025)(0.075)(0.150)(0.050)(0.050)
```

Unbalanced but Improved

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)

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

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.

K_k

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

$K_1,\ldots K_{k-1}$

$K_{k+1},\ldots K_n$
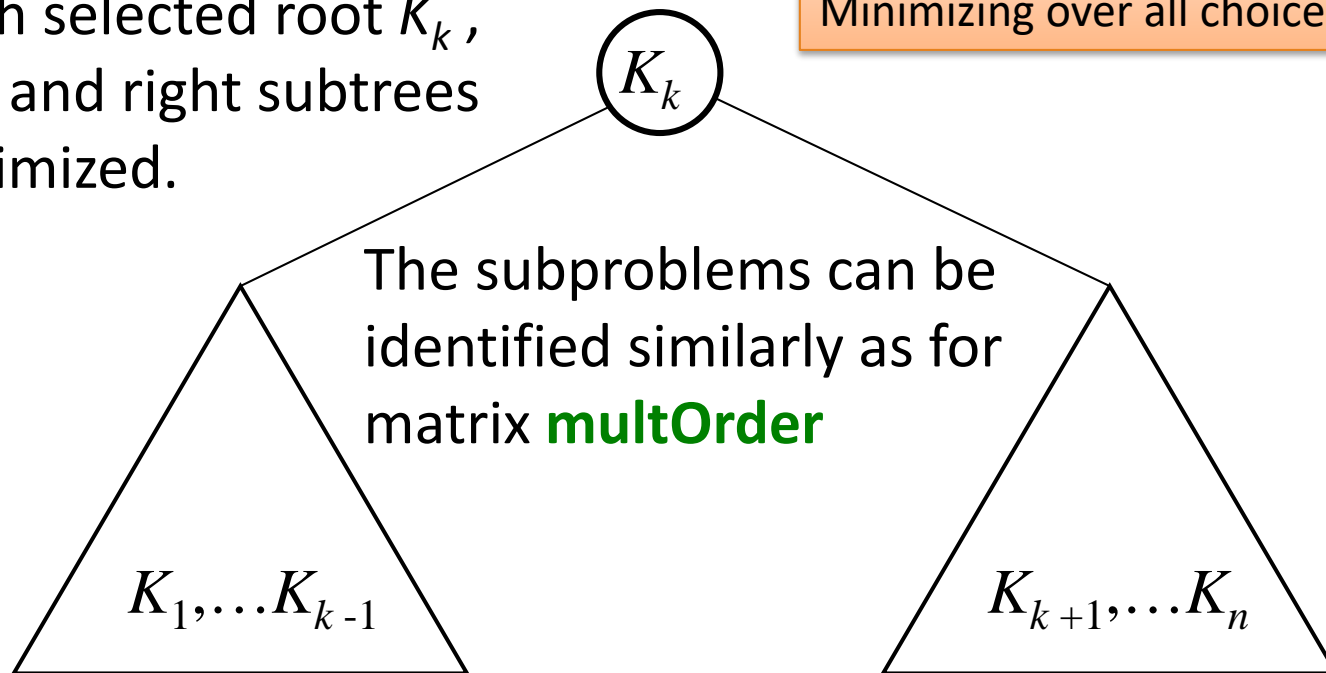
Subproblems as left and right subtrees

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.

Subproblem Solutions

- **Weighted retrieval cost of a subtree**
  - T contains $K_{\text{low}}, \ldots, K_{\text{high}}$, and the weighted retrieval cost of T is W, with T being a whole tree.
  - As a subtree with the root at level 1, the weighted retrieval cost of T will be: $\textbf{W+}\boldsymbol{p}\textbf{(low, high)}$

- **So, the recursive relations are:**
  - $A(\text{low, high, } r)$
    - $= p_r + p(\text{low, } r\text{-1}) + A(\text{low, } r\text{-1}) + p(r\text{+1, high}) + A(r\text{+1, high})$
    - $= p(\text{low, high}) + A(\text{low, } r\text{-1}) + A(r\text{+1, high})$
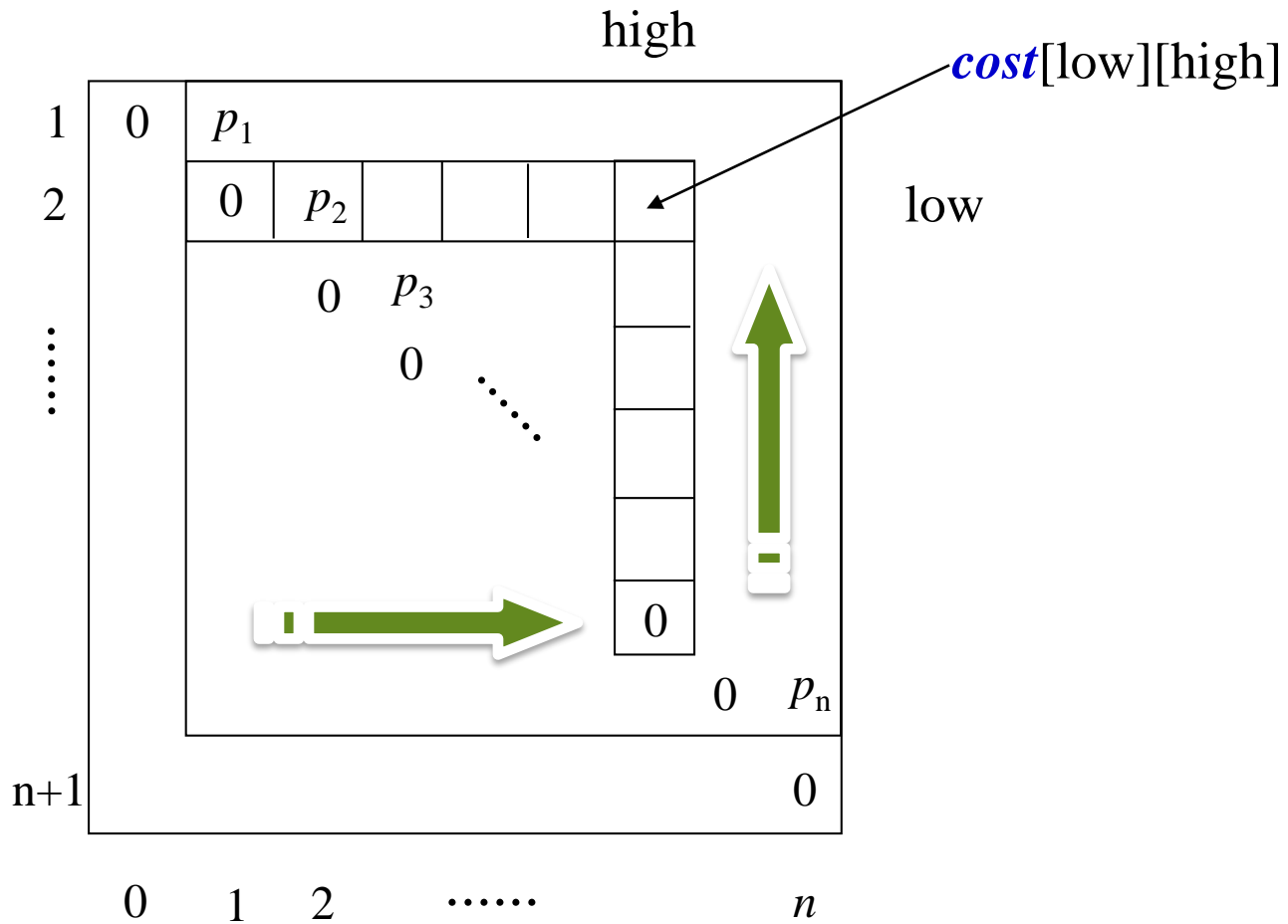 - $A(\text{low, high}) = \min\{A(\text{low, high, } r) \mid \text{low} \leq r \leq \text{high}\}$

Using DP

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

- **Array *root***
  - *root*[low][high] gives the best choice of root for subproblem (low,high)

Array cost[]

Optimal BST by DP

```
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 Θ(n³)

Thank you!

Q & A

Yu Huang

http://cs.nju.edu.cn/yuhuang