Introduction to

Algorithm Design and Analysis

[16] Dynamic Programming 1

Jingwei Xu
https://cs.nju.edu.cn/ics/people/jingweixu/index.html
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

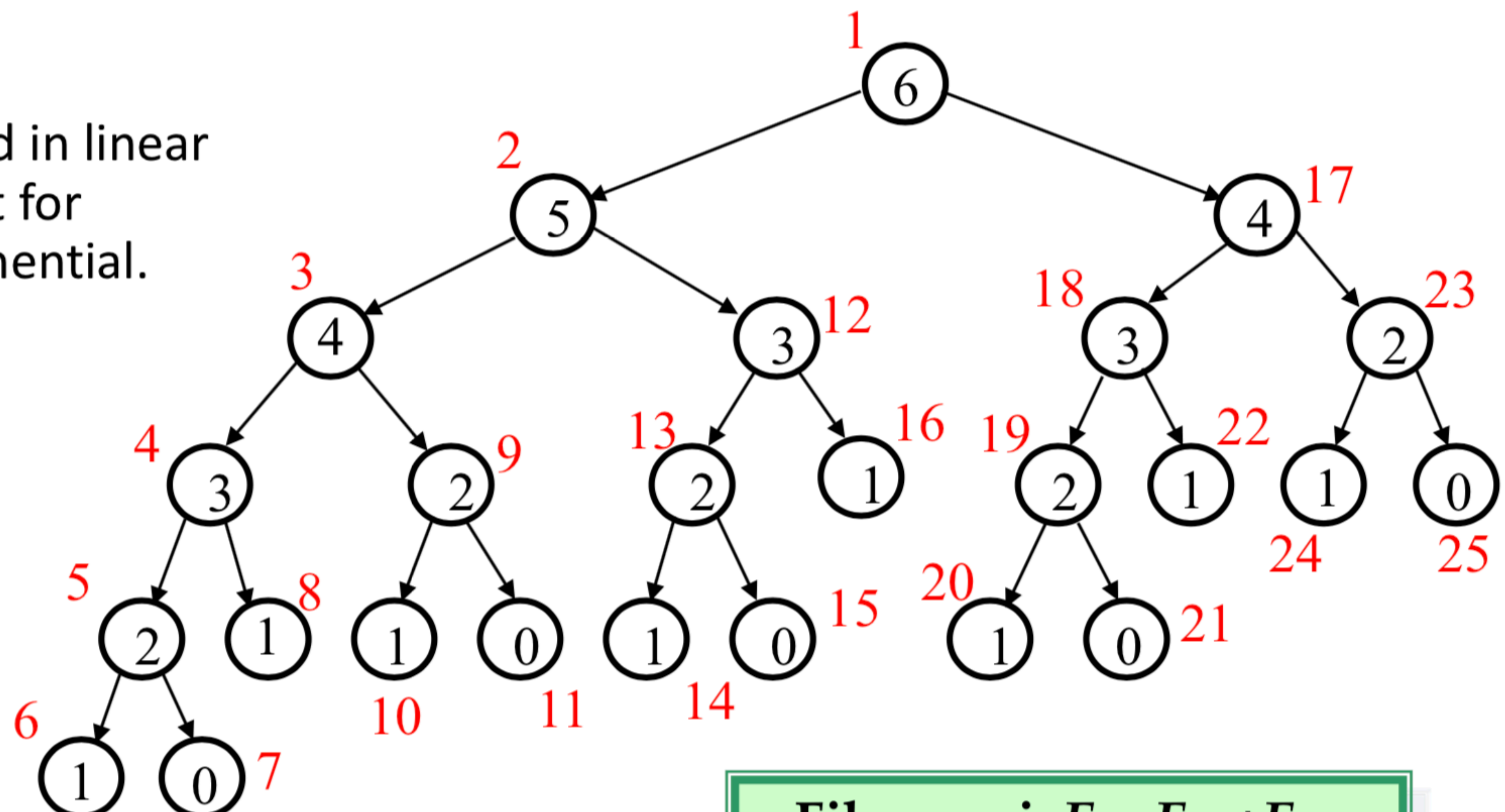
Dynamic Programming

- **Basic Idea of Dynamic Programming (DP)**
  - Smart scheduling of subproblems
- **Minimum Cost Matrix Multiplication**
  - BF1, BF2
  - A DP solution
- **Weighted Binary Search Tree**
  - 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: $F_n = F_{n\text{-}1} + F_{n\text{-}2}$

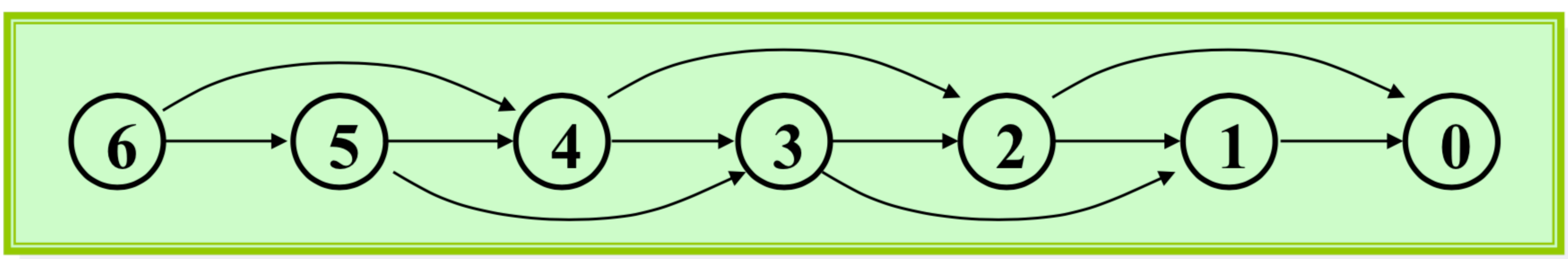
0, 1, 1, 2, 3, 5, 8, 13, 21, 35, ...

For your reference

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

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)



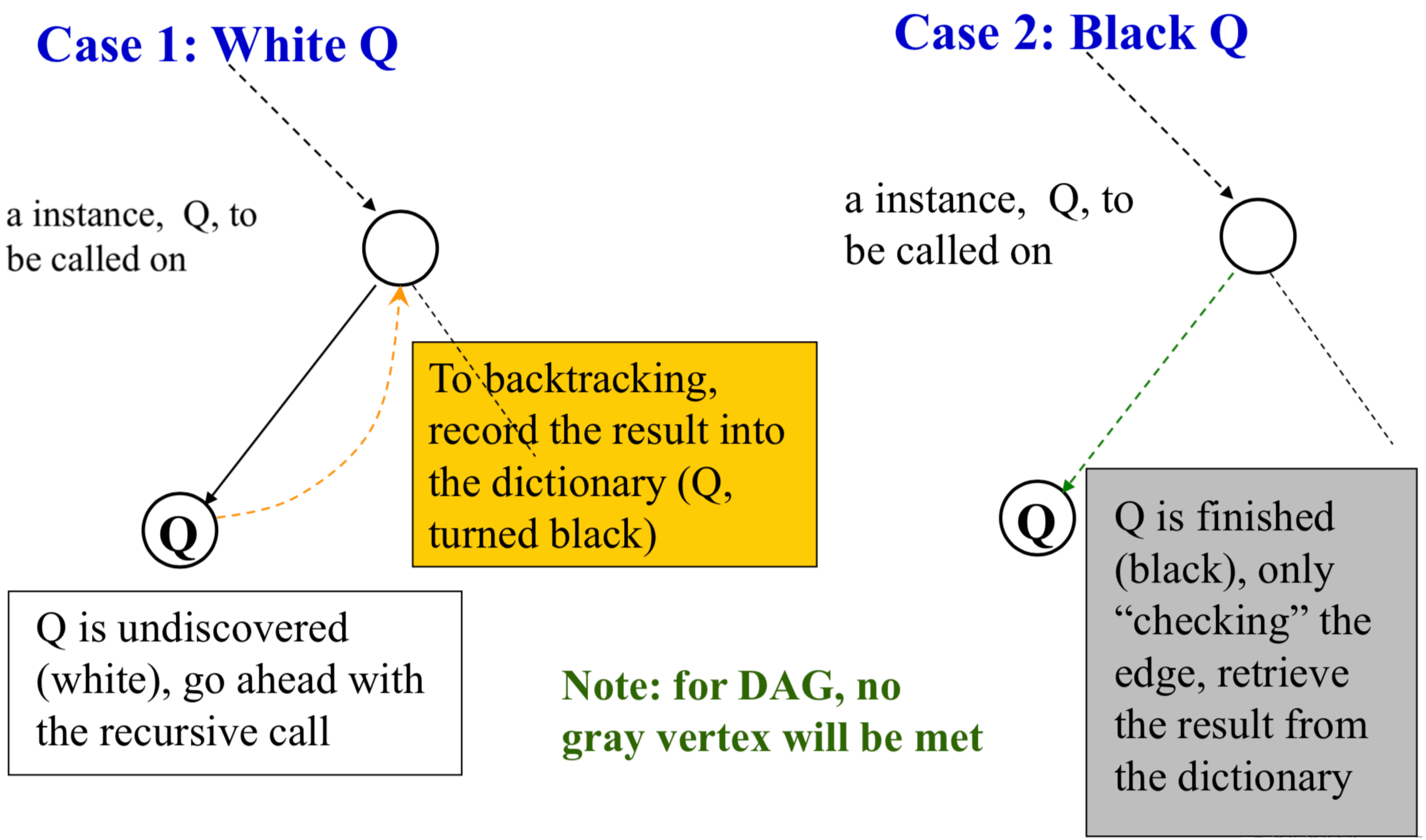
Properties of Subproblem Graph

- **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



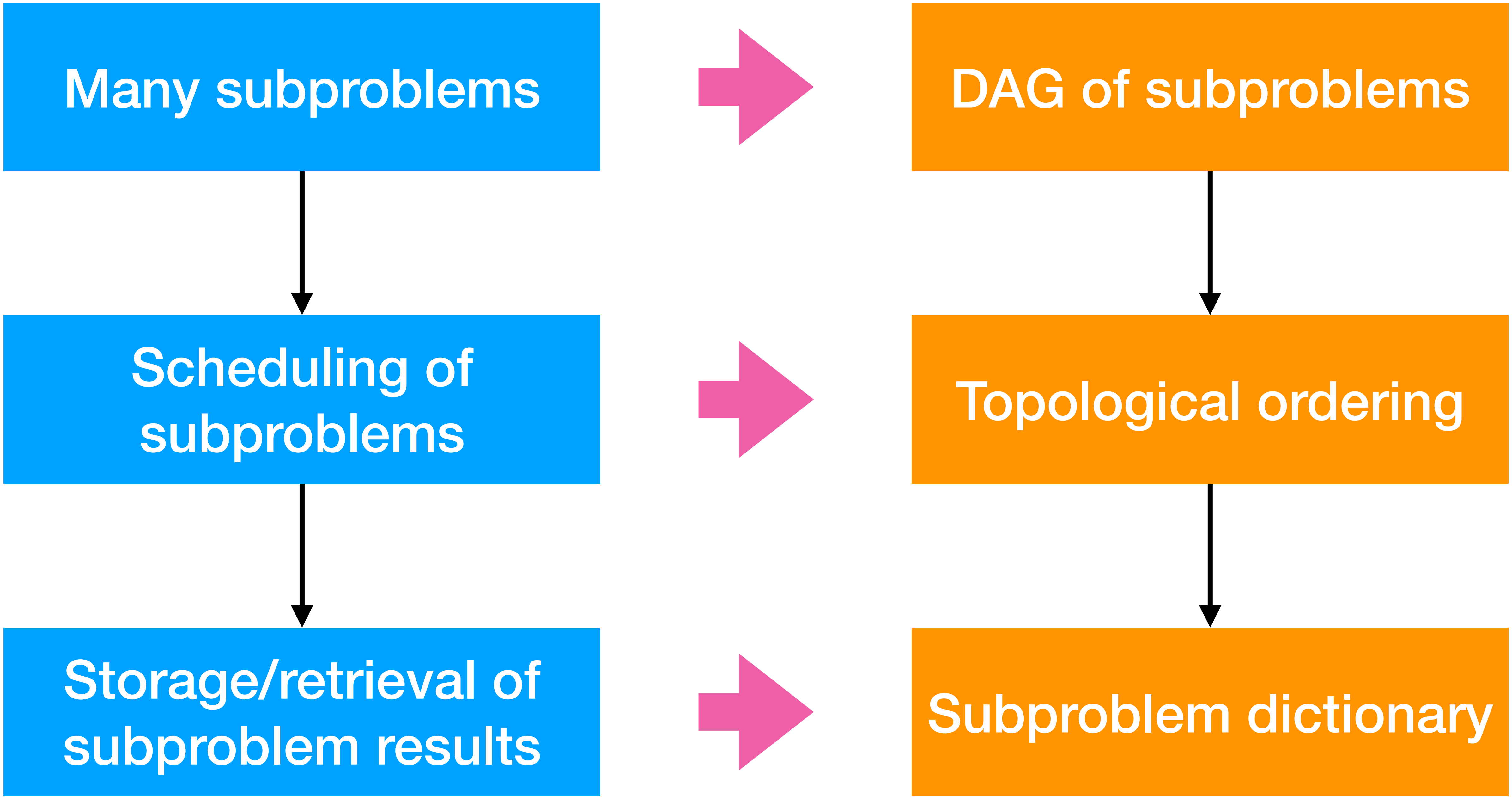
Fibonacci by DP

```
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
```

DP: New Concept Recursion



Matrix Multiplication Order Problem

- **The task:**
  - Find the product: $A_1 \times A_2 \times \ldots A_{n-1} \times A_n$
  - A_i is 2-dimensional 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_{pxq} x B_{qxr}

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

There are q multiplication

C has pxr elements as $c_{i,j}$

So, pqr multiplications altogether

Cost for Matrix Multiplication

Let C = A_{pxq} x B_{qxr}

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

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 pxr elements as $c_{i,j}$

So, pqr multiplications altogether

Looking for a Greedy Solution

- **Strategy 1: "cheapest multiplication first"**
  - Success: $A_{30x1} \times ((A_{1x40} \times A_{40x10}) \times A_{10x25})$
  - Fail: $(A_{4x1} \times A_{1x100}) \times A_{100x5}$
- **Strategy 2: "largest dimension first"**
  - Correct for the second example above
  - $A_{1x10} \times A_{10x10} \times A_{10x2}$: two results

Intuitive Solution

- **Matrices: A_1, A_2, …, A_n**
- **Dimension: dim: $d_0,d_1,d_2,\ldots,d_{n-1},d_n$, for A_i is d_{i-1} x d_i**
- **Sub-problem: seq: $s_0,s_1,s_2,\ldots,s_{k-1},s_{len}$, which means the multiplication of k matrices, with the dimensions: d_{s0} x d_{s1},d_{s1} x d_{s2},…,$d_{s[len]-1}$ x $d_{s[len]}$.**
  - 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 ith 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 $A_k \times A_{k+1}$.

$T(n)=(n-1)T(n-1)+n$, 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

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]

DP dict

Multiplication Order

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**

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

First entry filled

Last entry filled

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

For each selected *k*, retrieving:

- least cost of $A_{i+1} \times \ldots \times A_k$.
- least cost of $A_{k+1} \times \ldots \times A_j$.

and computing:

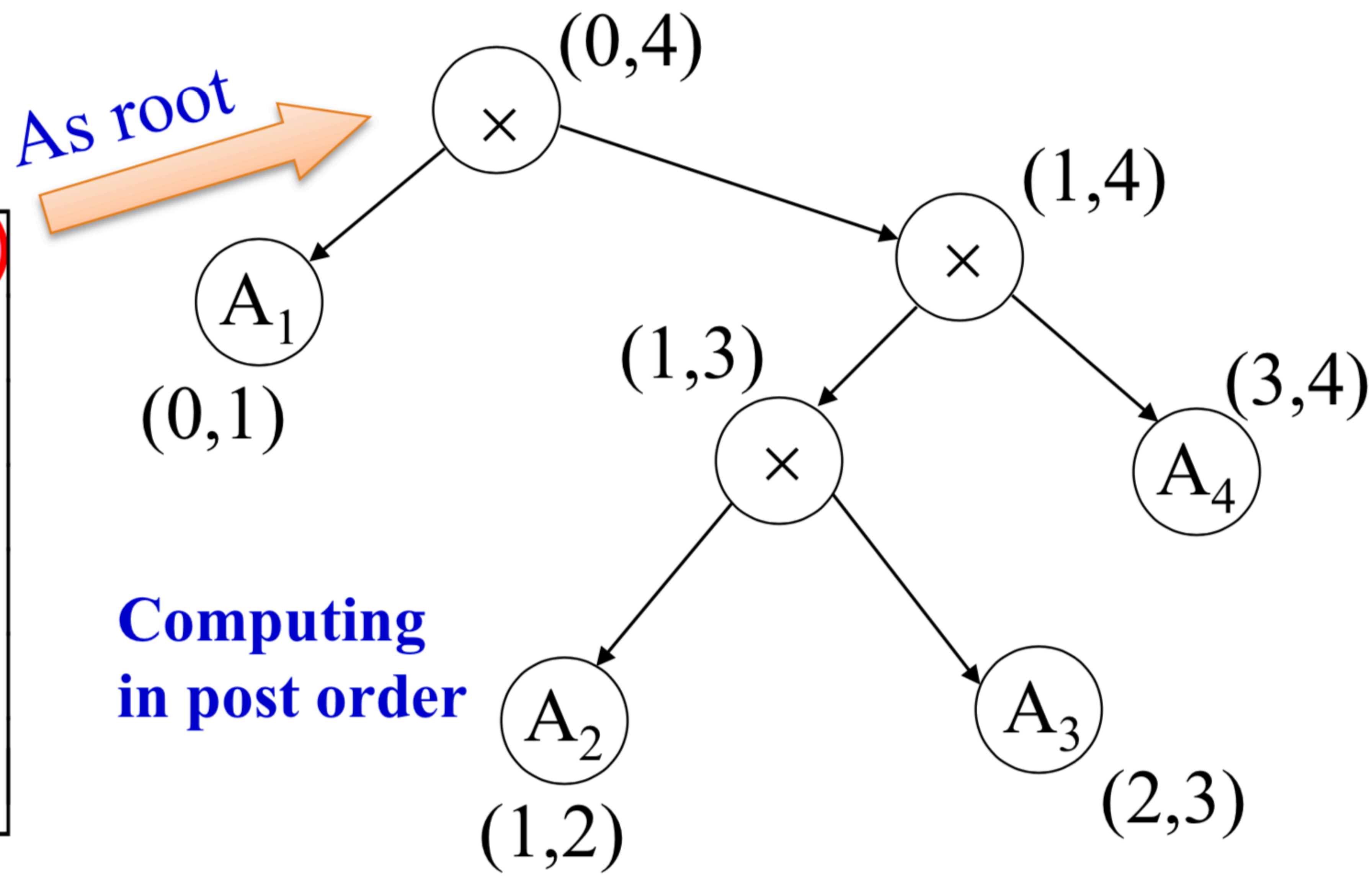
- cost of the last multiplication

Arithmetic Expression Tree

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

last **as finished**

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



Getting the optimal Order

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

```
extractOrder(low, high, last, multOrder)
    int k;
    if (high-low>1)
        k=last[low][high];
        extractOrder(low, k, last, multOrder);
        extractOrder(k, high, last, multOrder);
        multOrder[multOrderNext]=k;
        multOrderNext++;
```

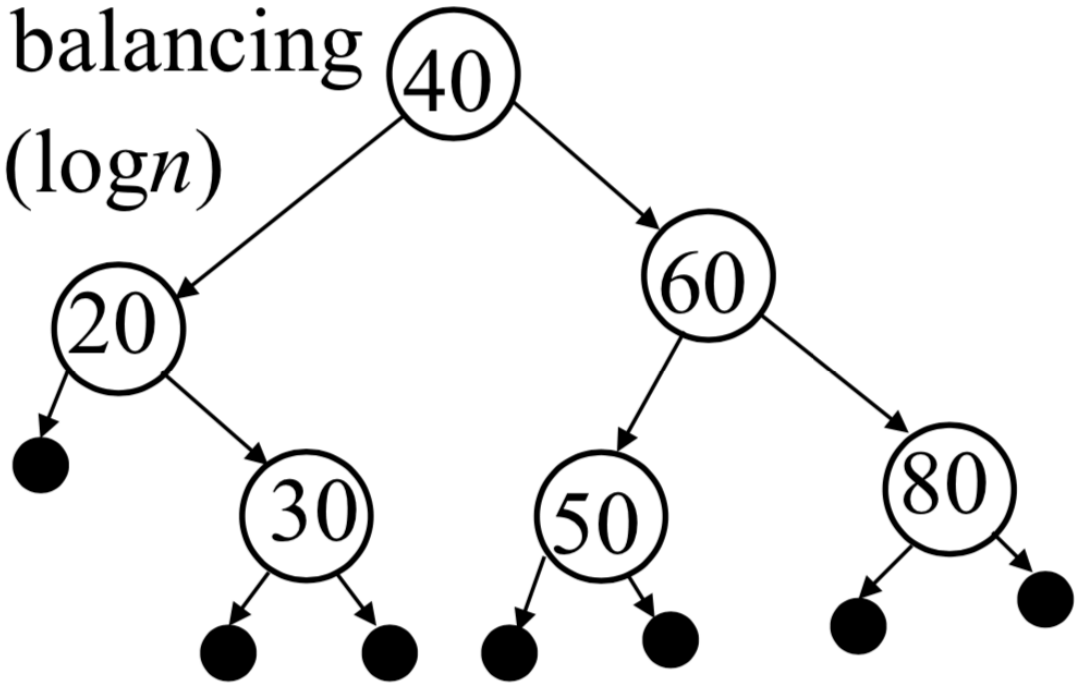
Just a post-order traversal

initialized in the wrapper

Binary Search Tree

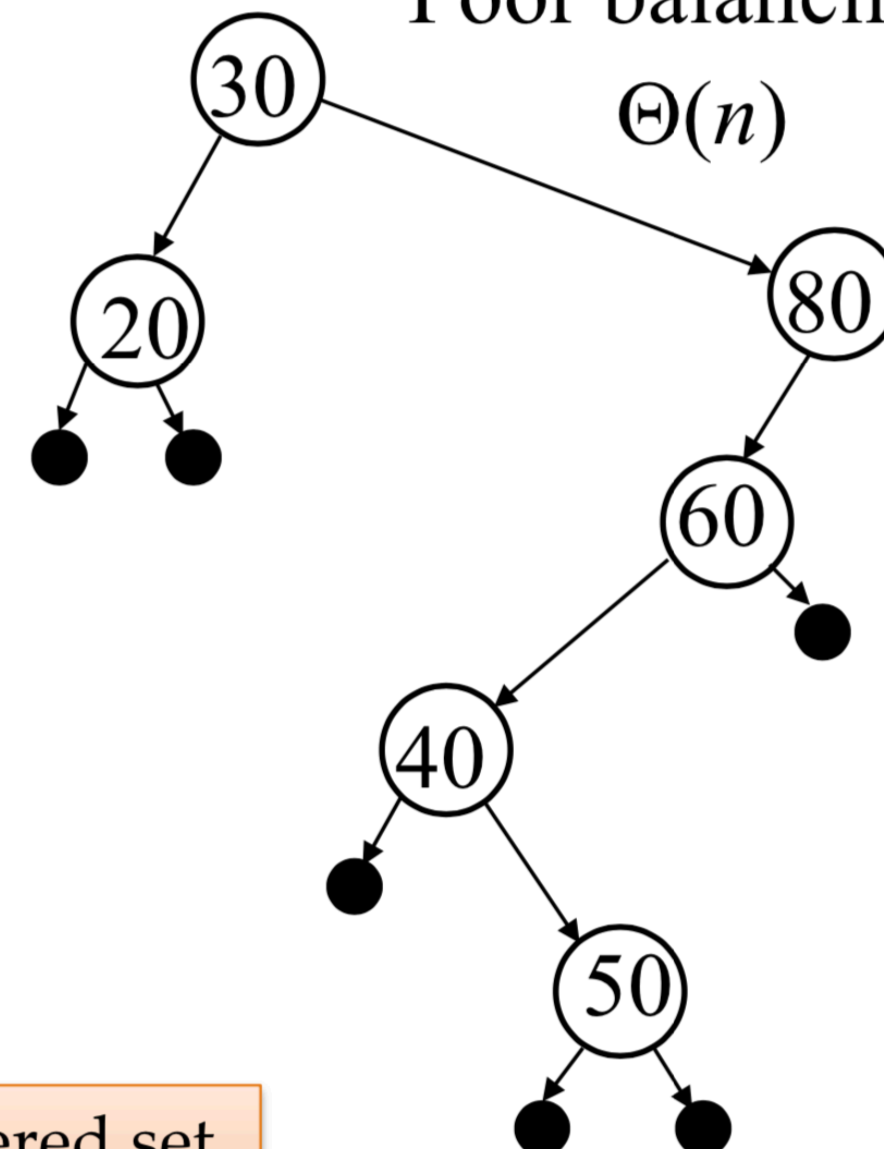
Good balancing

$\Theta(\log n)$



Poor balancing

$\Theta(n)$

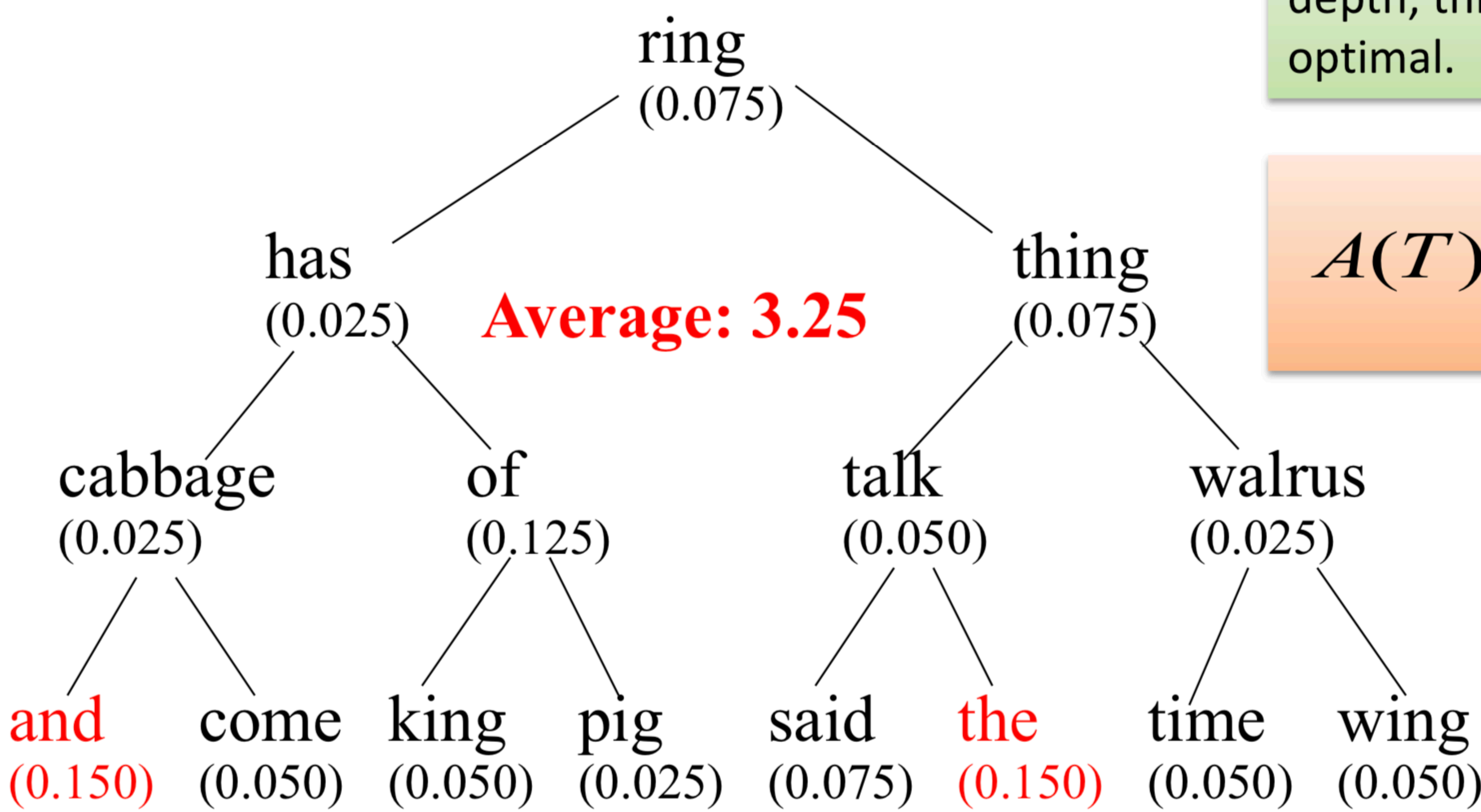


In a properly drawn tree, pushing forward to get the ordered list.

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

Key with Different Frequencies

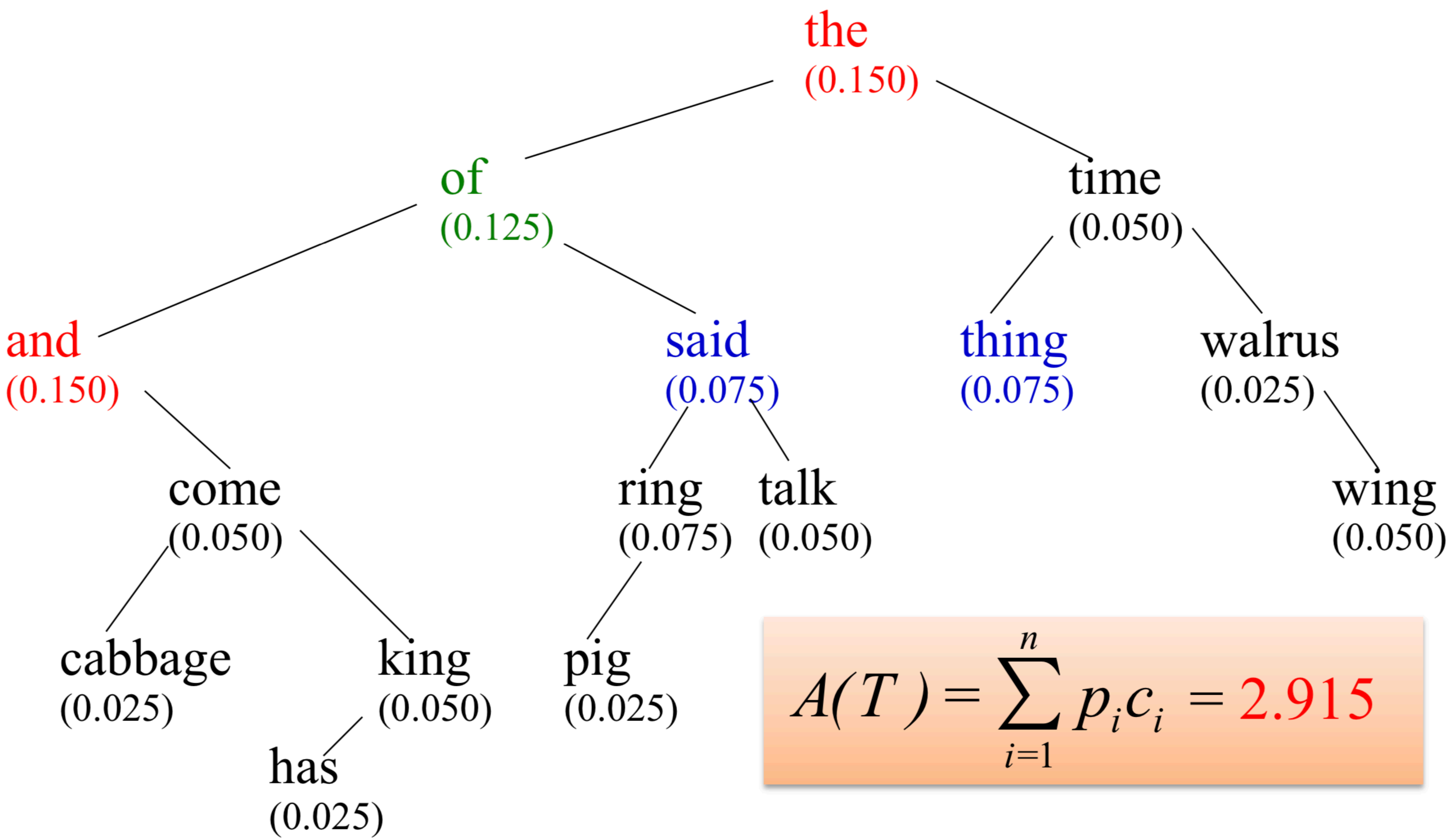
A binary search tree perfectly balanced



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

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

Unbalanced but Improved

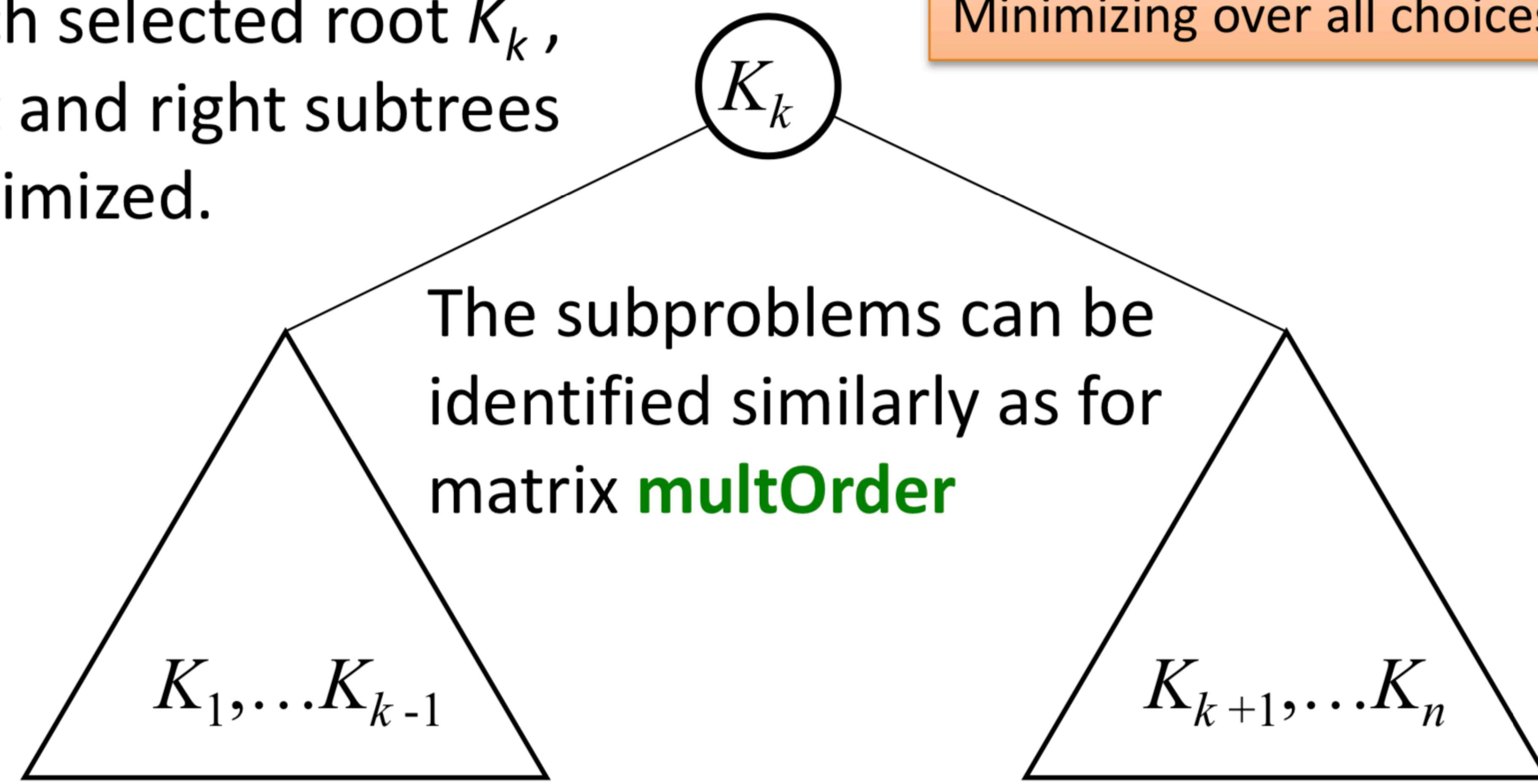


$$A(T) = \sum_{i=1}^{n} p_i c_i = 2.915$$

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.



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 Kr 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 $p_{low} + p_{low+1} + \ldots + p_{high}$, is the weight of the subproblem (low, high).
  - Note: p(low, high) is the probability that the key searched for is in this interval.

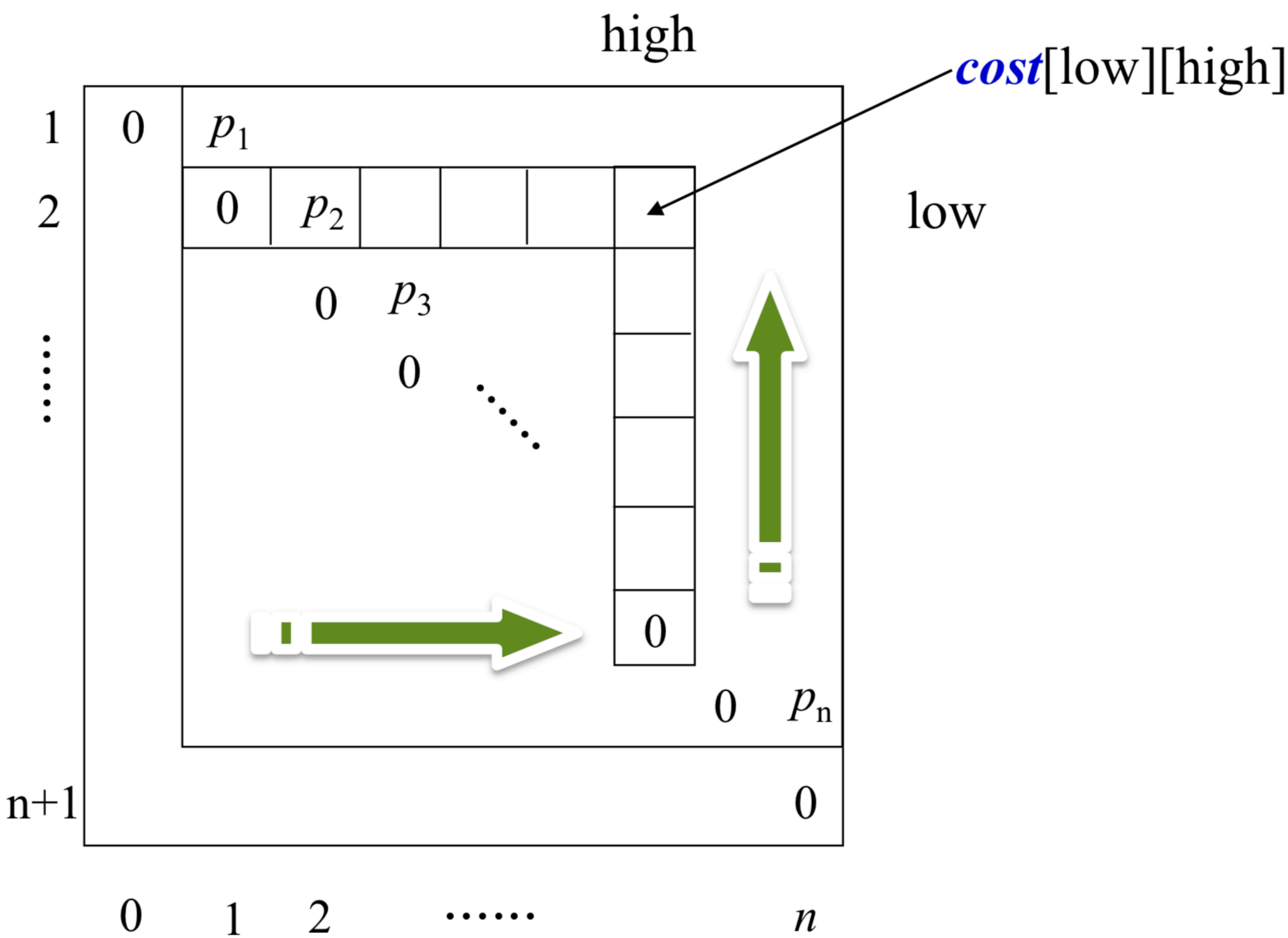
Subproblem Solutions

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

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 $\Theta(n^3)$

Thank you!
Q & A