Lakehead UNIVERSITY

COMP 4433: Algorithm Design and Analysis

Dr. Y. Gu

Feb. 6, 2023 (Lecture 7)

Dynamic Programming
(Final Example)

Example 5:

Optimal Binary Search Trees

Optimal Binary Search Trees

Binary search tree is a binary tree, in which the keys in the left subtree is less than the key in the root while keys in the right subtree is greater than the key in the root, and a subtree of binary search tree is also a binary search tree.

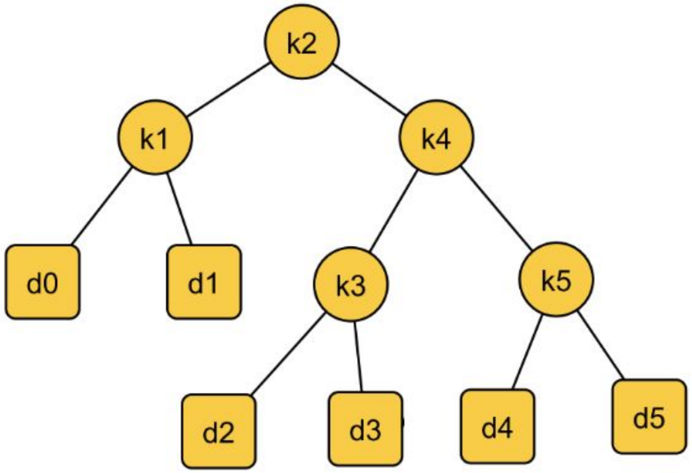
Now we consider a more general case. Suppose we have a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order (i.e., $k_1 < k_2 < \cdots < k_n$). For each key k_i, the probability a search will be on k_i is p_i. We wish to build a binary search tree for these keys such that the expected search time (the average search time) is optimal.

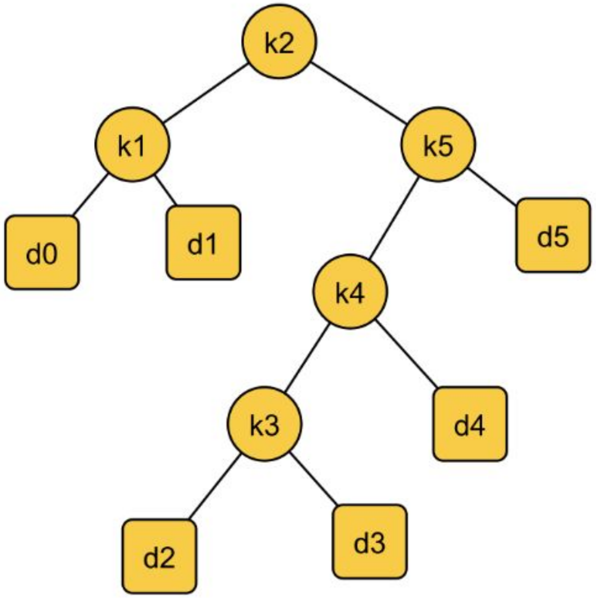
Optimal Binary Search Trees

We also need to consider the search values that are not in K. So we have n+1 dummy keys d_0, d_1, d_2 . . . d_n, where, d_i, 0<i<n, represents the values between k_i and k_{i+1}, d_0 represents the values less than k_1 and d_n represents the values greater than k_n. For each dummy key d_j, we assume the probability for searching according to it is q_j. For each dummy key d_j, we assume the probability for searching according to it is q_j. So, we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1.$$

Optimal Binary Search Trees





Optimal Binary Search Trees

Suppose we have already established the binary search tree T (in the tree, dummy keys should be leaves). Then we have the expected cost of a search in T is

$$\begin{aligned} E[\text{search cost in } T] &= \sum_{i=1}^{n} (depth_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n} (depth_T(d_i)+1)\cdot q_i \\ &= 1 + \sum_{i=1}^{n} depth_T(k_i)\cdot p_i + \sum_{i=0}^{n} depth_T(d_i)\cdot q_i , \end{aligned}$$

where *depthT* denotes a node's depth in the tree T. If the expected search cost is the smallest, then we call T an optimal binary search tree.

Optimal Binary Search Trees

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Two binary search trees are displayed in the previous slide.

The first tree has the expected search cost 2.80 and the second tree has the expected search cost 2.75, which is optimal.

Optimal Binary Search Trees

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

Optimal Binary Search Trees

To construct the tree, we can first construct a binary search tree with the n keys, then add the dummy nodes to leaves. But the number of binary search tree with n nodes is $\Theta(4^n/n^{3/2})$. So exhaustive search is not feasible. We can consider to use dynamic programming.

Dynamic Programming

Step 1: The structure of an optimal binary search tree

Suppose we have constructed an optimal binary search tree. Then each subtree must contain keys in a contiguous range $k_i, k_{i+1}, \ldots, k_j$, for some $1 \le i \le j \le n$. In addition, that subtree must also contains the leaves of dummy keys $d_{i-1}, d_i, \ldots, d_j$.

Therefore we have the optimal substructure: if an optimal binary search tree T has a subtree T′ containing keys $k_i, \ldots, k_j$, then T′ must be optimal as well for subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$. Otherwise we can replace the subtree with better expected cost and that means that T is not optimal.

Dynamic Programming

Considering the recursive method, if a subtree contains keys $k_i, \ldots, k_j$ and the root is k_r, then the left subtree contains keys $k_i, \ldots, k_{r-1}$ (and dummy keys $d_{i-1}, \ldots, d_{r-1}$) and the right subtree contains keys $k_{r+1}, \ldots, k_j$ (and dummy keys $d_r, \ldots, d_j$).

When the root is i, then the left subtree contains only d_{i-1} and when k_j is the root, its right subtree contains only d_j . We may try every possible key as the root to obtain the optimal subtree.

Dynamic Programming

Step 2: A recursive solution

We can define the values of optimal solution for subtrees as follows. For a subtree with keys k_i , . . . , k_j , define $e[i, j]$ to be the optimal expected cost of searching, where $i \geq 1$, $i - 1 \leq j \leq n$. Here we define $e[i, i - 1]$ as the subtree with d_{i-1} as a only node. So

$$e[i, i - 1] = q_{i-1}.$$

Dynamic Programming

- When $j \geq i$, we need to select a root k_r, which forms two subtrees, one with the keys $d_i, \ldots, d_{r-1}$ and another with the keys $d_{r+1}, \ldots, d_j$.
- For a tree containing keys $k_s, \ldots, k_t$, the optimal value is e[s, t].
- But when it becomes a subtree, the depth of each vertex will increase one. Therefore the the expected costs for this subtree will be

$$e[s,t] + \sum_{l=s}^{t} p_l + \sum_{l=s-1}^{t} q_l .$$

Dynamic Programming

We define

$$w(s,t) = \sum_{l=s}^{t} p_l + \sum_{l=s-1}^{t} q_l$$

Thus, if k_r is the root of an optimal subtree containing keys k_i ,..., k_j , we have

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j)) .$$

Note that $w(i,j) = w(i,r-1) + p_r + w(r+1,j)$

We have $e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j) .$

Dynamic Programming

Now we have the recursive formula for e[i, j].

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ \min_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j. \end{cases}$$

To help us to keep the track of the structure of optimal binary search tree, we define root[i, j] to be the index r for which k_r is the root of an optimal binary search tree containing keys k_i ,. . . , k_j .

Dynamic Programming

Step 3: Computing the expected search cost of an optimal BST

Similar to other dynamic programming, we need to use some tables to store the solutions for subproblems. So we define tables *e*, *w* and *root* in the following procedure. For *e* and *w* we need to define $1 \le i \le n + 1$, $0 \le j \le n$, because we need to record the values of “empty” subtrees (e.g., $e[i, i-1]$, $1 \le i \le n$).

Dynamic Programming

```
OPTIMAL-BST(p, q, n)
let e[1..n+1, 0..n], w[1..n+1, 0..n],
        and root[1..n, 1..n] be new tables
for i = 1 to n + 1
    e[i, i-1] = q_{i-1}
    w[i, i-1] = q_{i-1}
for l = 1 to n
    for i = 1 to n - l + 1
        j = i + l - 1
        e[i, j] = ∞
        w[i, j] = w[i, j-1] + p_j + q_j
        for r = i to j
            t = e[i, r-1] + e[r+1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```

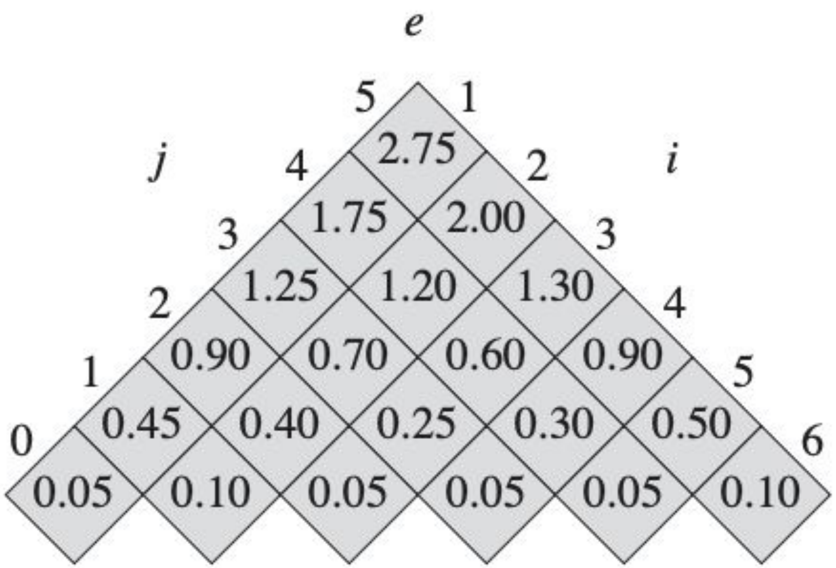
Running Time Discussion

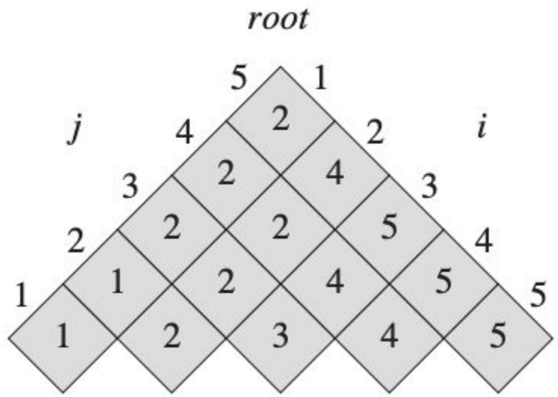
The Optimal-BST procedure takes $\Theta(n^3)$ time.

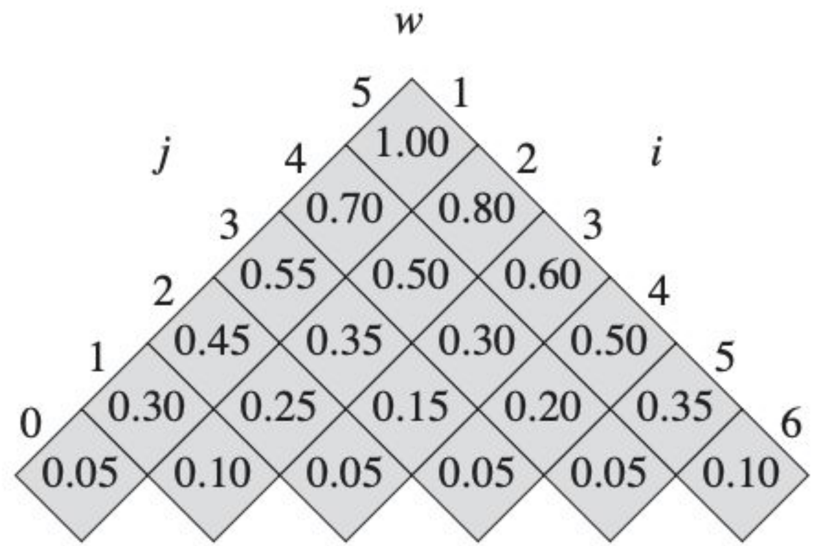
Because the main costs are the three nested for loops, each loop index takes at most n values, the running time is $O(n^3)$.

On the other hand, we can also see that the procedure takes $\Omega(n^3)$ time.

Example


e
5 1
j i
2.75
4 2
1.75 2.00
3 3
1.25 1.20 1.30
2 4
0.90 0.70 0.60 0.90
1 5
0.45 0.40 0.25 0.30 0.50
0 6
0.05 0.10 0.05 0.05 0.05 0.10


root
5 1
j i
2
4 2
2 4
3 3
2 2 5
2 4
1 2 4 5
1 5
1 2 3 4 5


w
5 1
j i
1.00
4 2
0.70 0.80
3 3
0.55 0.50 0.60
2 4
0.45 0.35 0.30 0.50
1 5
0.30 0.25 0.15 0.20 0.35
0 6
0.05 0.10 0.05 0.05 0.05 0.10

Greedy Algorithm

Introduction

- The main idea of greedy algorithm is look some optimal solution locally and then try to extend globally. Usually the greedy algorithm is efficient.
- The greedy algorithm may not achieve optimal solution for the problem.
- We shall arrive at the greedy algorithm by first considering a dynamic programming approach and then showing that we can always make greedy choices to arrive at an optimal solution.

An Activity-Selection Problem

Suppose we have a set $S = \{a_1, a_2, . . . , a_n\}$ of *n* proposed activities that wish to use a resource (for example, a_i are presentations, which need to use one classroom).

Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the time internal $[s_i, f_i)$. Activity a_i and a_j are compatible if $[s_i, f_i) \cap [s_j , f_j) = \varnothing$, that is, if $s_i \geq f_j$ or $s_j \geq f_i$.

In the activity-selection problem, we wish to select a maximum-size subset of mutually compatible activities.

We assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \leq f_2 \leq \cdot \cdot \cdot \leq f_{n-1} \leq f_n.$$

An Activity-Selection Problem

Example: Suppose the activity set S is as follows.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Then the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. But it is not the largest subset. The subsets $\{a_1, a_4, a_8, a_{11}\}$ or $\{a_2, a_4, a_9, a_{11}\}$ are largest subsets.

An Activity-Selection Problem

We first try to find some recursive method for the optimal subproblems.

Let S_{ij} denote the subset of activities that start after activity a_i finishes and end before a_j starts, and suppose such a maximum set is A_{ij} .

Let $a_k \in A_{ij}$ be an activity, then we claim that $A_{ik} = S_{ik} \cap A_{ij}$ must be an optimal solution of S_{ik}. Otherwise we will be able to improve A_{ij} and A_{ij} would not be optimal. Similarly, $A_{kj} = S_{kj} \cap A_{ij}$ is also optimal.

Therefore, $A_{ij} = A_{ij} \cup \{a_k\} \cup A_{kj}$ and $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$.

An Activity-Selection Problem

Let c[i, j] denote the size of optimal solution for the set S_{ij} , then we have the following formula, i.e. c[i, j]=$|A_{ij}|$, then

$$c[i,j] \quad = \quad \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

- From the above formula, we can develop a dynamic programming.
- We want to use a simpler method to solve the problem with “greedy choice”.

After Class

- After class:

Part III Chapter 12, Part IV Chapter 15.5