Today:
– Optimal Binary Search

COSC 581, Algorithms

January 30, 2014

Many of these slides are adapted from several online sources

Reading Assignments

- Today's class:
  - Chapter 15.5
- Reading assignment for next class:
  - Chapter 22, 24.0, 24.2, 24.3
- Announcement: Exam 1 is on Tues, Feb. 18
  - Will cover everything up through dynamic programming

Optimal Binary Search Trees

- **Problem Statement:**
  - Given sequence $K = k_1 < k_2 < \cdots < k_n$ of *n* sorted keys, with a search probability p_i for each key k_i.
  - Also given $n + 1$ "dummy keys" $d_0, d_1, \ldots, d_n$representing searches not in k_i
    - In particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n, and for $i = 1, 2, \ldots, n - 1$, the dummy key d_i represents all values between k_i and k_{i+1}.
    - The dummy keys are leaves (external nodes), and the data keys are internal nodes.
    - For each dummy key d_i, we have search probability q_i

  - We want to build a binary search tree (BST) with minimum expected search cost.
  - Actual cost = # of items examined.
  - For key k_i, cost = $\text{depth}_T(k_i)$+1,

    where $\text{depth}_T(k_i)$ = depth of k_i in BST *T* .

We add 1 because root is at depth 0

Expected Search Cost

Since every search is either successful or not, the probabilities sum to 1:

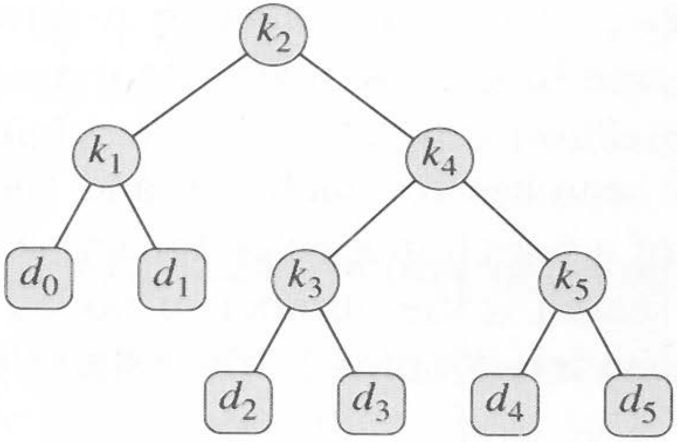
$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

$$\text{E[search cost in T]} = \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i$$

$$= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i$$

Example – Expected Search Cost

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

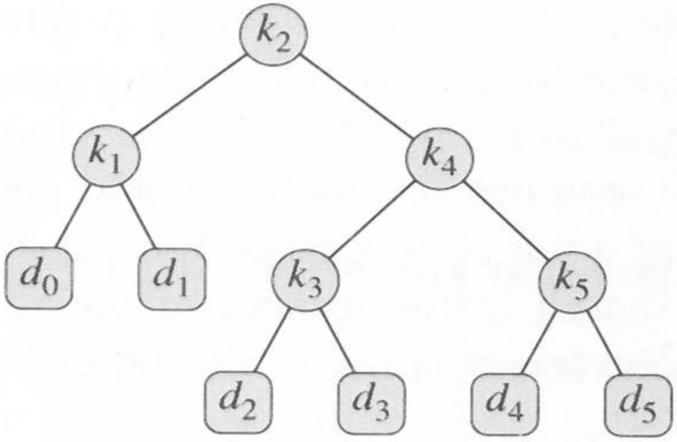


cost: 2.80

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

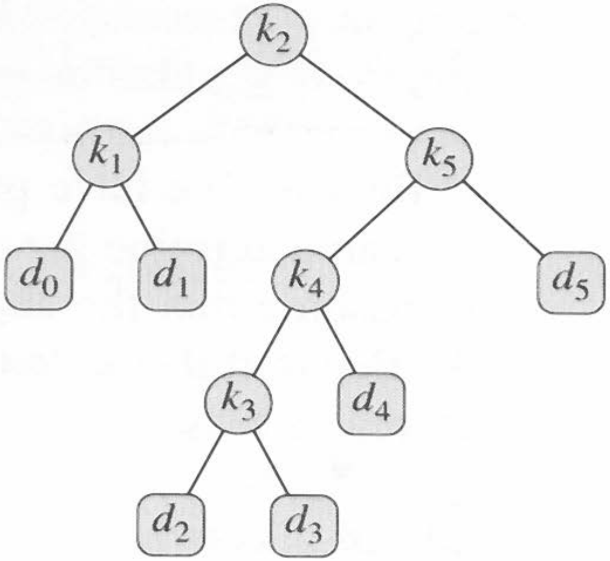
Example – Expected Search Cost

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



cost: 2.80

But there's a better solution:



cost: 2.75
optimal!!

Observations

- **Observations:**
  - Optimal BST may not have smallest height.
  - Optimal BST may not have highest-probability key at root.

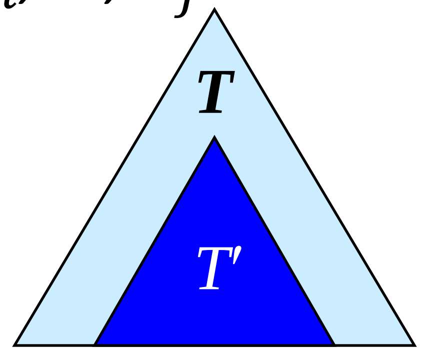
Brute Force?

- Build by exhaustive checking?
  - Construct each n-node BST.
  - For each,
    assign keys and compute expected search cost.
  - Pick best

- But there are $\Omega(4^n/n^{3/2})$ different BSTs with n nodes!

Step 1: Optimal Substructure

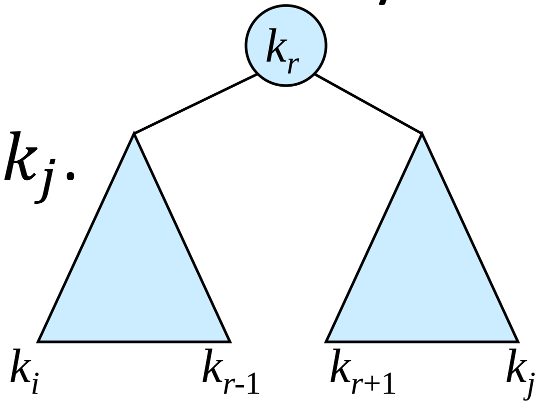
- Any subtree of a BST contains keys in a contiguous range $k_i, \ldots, k_j$ for some $1 \le i \le j \le n$.



- If T is an optimal BST and
  T contains subtree T' with keys $k_i, \ldots, k_j$,
  then T' must be an optimal BST for keys $k_i, \ldots, k_j$.
- **Proof:** Cut and paste.

Optimal Substructure

- One of the keys in k_i, …, k_j, say k_r, where $i \le r \le j$, must be the root of an optimal subtree for these keys.
- Left subtree of k_r contains $k_i,\ldots,k_{r-1}$.
- Right subtree of k_r contains k_{r+1} , …, k_j.



- To find an optimal BST:
  - Examine all candidate roots k_r , for $i \le r \le j$
  - Determine all optimal BSTs containing $k_i,\ldots,k_{r-1}$ and containing $k_{r+1},\ldots, k_j$

Step 2: Recursive Solution

- Find optimal BST for $k_i, \ldots, k_j$, where $i \geq 1, j \leq n, j \geq i{-}1$. When $j = i{-}1$, the tree is empty.
- Define $e[i, j]$ = expected search cost of optimal BST for $k_i, \ldots, k_j$.

- If $j = i - 1$, then $e[i, j]$ = q_{i-1}.
- If $j \geq i$,
  - Select a root k_r for some $i \leq r \leq j$.
  - Recursively make optimal BSTs
    - for $k_i, \ldots, k_{r-1}$ as the left subtree, and
    - for $k_{r+1}, \ldots, k_j$ as the right subtree.

Step 2: Recursive Solution

- When the optimal subtree becomes a subtree of a node:
  - Depth of every node in optimal subtree goes up by 1.
  - Expected search cost increases by sum of probabilities of subtree: $$\boxed{w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l}$$

- If k_r is the root of an optimal BST for $k_i, \ldots, k_j$:

$$\begin{aligned} e[i,j] &= p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j)) \\ &= e[i,r-1] + e[r+1,j] + w(i,j). \end{aligned}$$

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

- But, we don't know k_r. Hence,

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

Step 3: Computing an Optimal Solution

For each subproblem (i, j), store:

- Expected search cost in a table $e[1\,..\,n+1\,,0\,..\,n]$
  - Will use only entries $e[i, j\,]$, where $j \geq i{-}1$.
- $\mathrm{root}[i, j]$ = root of subtree with keys $k_i, \ldots, k_j$, for $1 \leq i \leq j \leq n$.
- $w[1..n+1, 0..n]$ = sum of probabilities:
  - $w[i, i{-}1] \;=\; q_{i-1}$ for $1 \leq i \leq n+1$ *(base case)*
  - $w[i, j\,] \;=\; w[i, j-1] \;+\; p_j + q_j$ for $1 \leq i \leq j \leq \; n$

Note: *w* is stored for purposes of efficiency. Rather than compute $w(i, j)$ from scratch every time we compute $e[i, j]$, we store these values in a table instead.

Step 3: Computing the expected search cost of an optimal binary search tree

```
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
```

Consider all trees with l keys.

Fix the first key.

Fix the last key

Determine the root of the optimal (sub)tree

Time?

Step 3: Computing the expected search cost of an optimal binary search tree

```
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
```

Consider all trees with l keys.

Fix the first key.

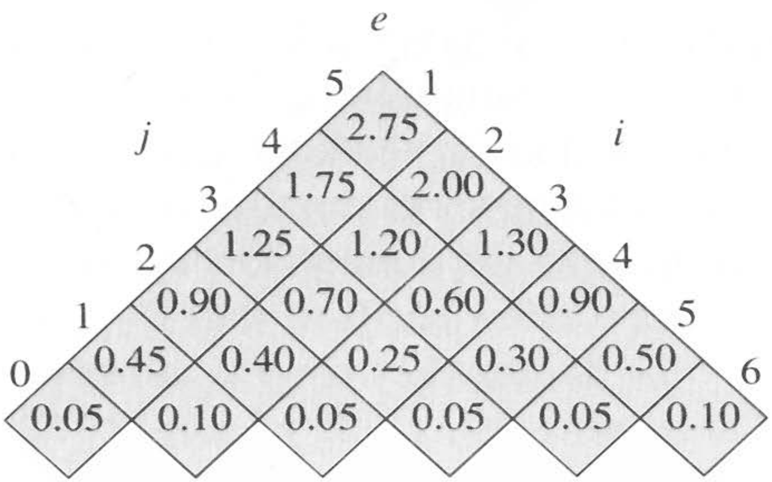
Fix the last key

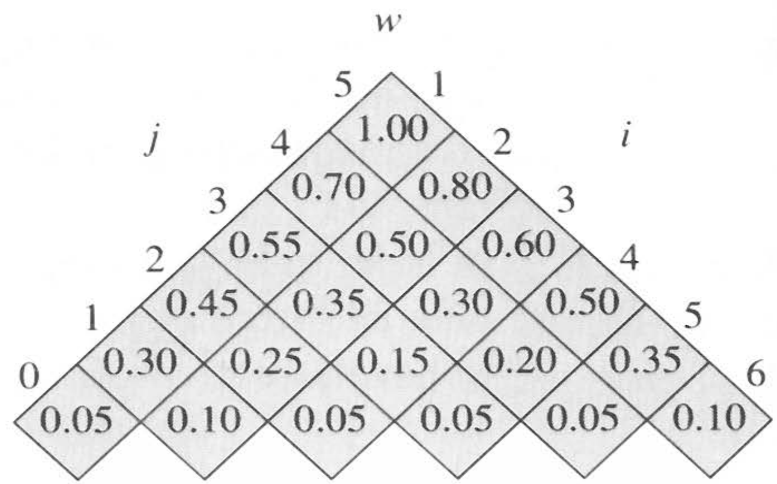
Determine the root of the optimal (sub)tree

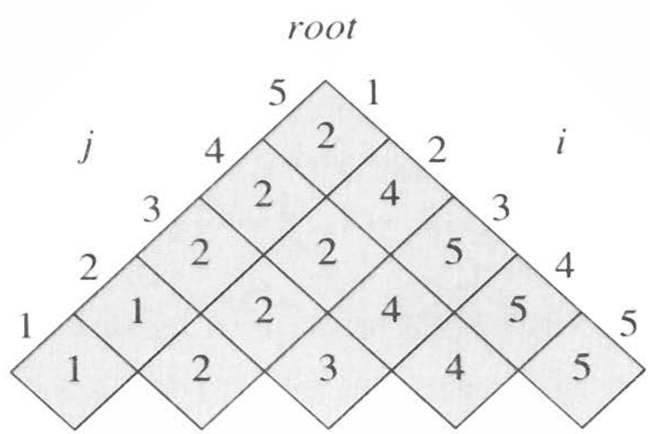
Time = $O(n^3)$

Table $e[i,j]$, $w[i,j]$, and $root[i,j]$ computed by OPTIMAL-BST on an example key distribution:

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







Dynamic Programming Exercise

- Consider an exam with n questions. For each $i = 1, \dots, n$, question i has integral point value $v_i > 0$ and requires $m_i > 0$ minutes to solve. Suppose further that no partial credit is awarded.
- The ultimate goal would be to come up with an algorithm which, given $v_1, v_2, \dots, v_n, m_1, m_2, \dots, m_n$, and V, computes the minimum number of minutes required to earn at least V points on the exam. (For example, you might use this algorithm to determine how quickly you can get an A on the exam.)
- Let $M(i, v)$ denote the minimum number of minutes needed to earn v points when you are restricted to selecting from questions 1 through i. Complete the following recurrence expression for $M(i, v)$ (i.e., fill in the blank). The base cases are supplied for you.

$$M(i,v) = \begin{cases} 0 & \text{for all } i, \text{ if } v \leq 0 \\ \infty & \text{if } i = 0 \text{ and } v > 0 \\ \underline{\hspace{10cm}} & \text{otherwise} \end{cases}$$

Another DP Exercise

[In this problem, we define meanings for upper-case S_k and *W,* as well as for lower-case s_k and *w*. Keep in mind that the meanings of these variables are case-sensitive – i.e., s_k is not the same thing as S_k, and *W* does not have the same meaning as *w*. These are defined clearly below.]

The 0-1 Knapsack problem is as follows. You are given a knapsack with maximum capacity *W* and a set *S* consisting of n items, $\{s_1, s_2, \ldots, s_n\}$. Each item s_i has some weight w_i and benefit value b_i. Here, all w_i, b_i, and *W* are integer values. The problem is to pack the knapsack so as to achieve the maximum total value of the packed items. Each item has to be either entirely accepted, or entirely rejected; no partial items are allowed.

1. In a brute force approach, we would search all possible combinations of items and find the best one. How many possible combinations would we have to search using this brute force approach? (State your answer in big-O notation.)

Another DP Exercise

Let us now consider a dynamic programming approach to this problem. We will define subproblems as follows. Let S_k be the subset of items $\{s_1, s_2, \ldots, s_k\}$. We want to find the optimal solution for S_k, which would be the subset of items in S_k that achieves the maximum total value. To make this work properly, we need to define another parameter *w*, which is the exact weight for each subset of items. Thus, we now define a subproblem to be to compute *B*[*k*, *w*], which represents the value of the best subset of S_k that has total weight of exactly *w*.

We can now define a recursive solution for *B*[*k*, *w*] that considers 2 choices – either item s_k is part of the solution or it isn't. (We also must include a base case.) What is this recursive solution ?

- $$B[k,w] = \begin{cases} \underline{\hspace{10cm}} & \text{if } w_k > w \\ \underline{\hspace{10cm}} & \text{otherwise} \end{cases}$$

Another DP Exercise

Let us now consider a dynamic programming approach to this problem. We will define subproblems as follows. Let S_k be the subset of items $\{s_1, s_2, \ldots, s_k\}$. We want to find the optimal solution for S_k, which would be the subset of items in S_k that achieves the maximum total value. To make this work properly, we need to define another parameter w, which is the exact weight for each subset of items. Thus, we now define a subproblem to be to compute $B[k, w]$, which represents the value of the best subset of S_k that has total weight of exactly w.

We can now define a recursive solution for $B[k, w]$ that considers 2 choices – either item s_k is part of the solution or it isn't. (We also must include a base case.) What is this recursive solution ?

In filling in this $B[k, w]$ table, what are the indices over which k runs?

In filling in this $B[k, w]$ table, what are the indices over which w runs?

How much work has to be done for each subproblem (stated in Θ notation)?

What is the runtime of the DP alg. that implements this recursive solution?

Reading Assignments

- Reading assignment for next class:
  - Chapter 22, 24.0, 24.2, 24.3

- Announcement: Exam 1 is on Tues, Feb. 18
  - Will cover everything up through dynamic programming