Design and Analysis of Algorithms
Midterm Review

Lecture 8: Soring in Linear Time, Selection Problem, and Optimal Binary Search Tree Problem

Yongxin Tong (童咏昕)

School of CSE, Beihang University

yxtong@buaa.edu.cn

Review to Part I

- In Part I, we illustrated Divide-and-Conquer using several examples:
  - Maximum Contiguous Subarray (最大子数组)
  - Counting Inversions (逆序计数)
  - Polynomial Multiplication (多项式乘法)
  - QuickSort and Partition (快速排序与划分)

Review to Part I

- In Part I, we illustrated Divide-and-Conquer using several examples:
  - Maximum Contiguous Subarray (最大子数组)
  - Counting Inversions (逆序计数)
  - Polynomial Multiplication (多项式乘法)
  - QuickSort and Partition (快速排序与划分)

Introduction to Part II

- In Part II, we illustrated sorting and searching problems using several examples:
  - Heapsort and Priority Queues (堆排序与优先队列)
  - Lower Bound for Sorting (基于比较的排序下界)
  - Sorting in Linear Time (线性时间排序)
  - Selection Problem (选择问题)
  - AVL Tree (AVL树-二叉平衡树)

Introduction to Part II

- In Part II, we illustrated sorting and searching problems using several examples:
  - Heapsort and Priority Queues (堆排序与优先队列)
  - Lower Bound for Sorting (基于比较的排序下界)
  - Sorting in Linear Time (线性时间排序)
  - Selection Problem (选择问题)
  - AVL Tree (AVL树-二叉平衡树)

Review to Part III

- In Part III, we illustrated Dynamic Programming (DP) using several examples:
  - 0-1 Knapsack (0–1背包)
  - Rod-Cutting (钢条切割)
  - Chain Matrix Multiplication (矩阵链乘法)
  - Longest Common Subsequences (最长公共子序列)
  - Minimum Edit Distance (最小编辑距离)
  - Optimal Binary Search Trees (最优二叉查找树)

Review to Part III

- In Part III, we illustrated Dynamic Programming (DP) using several examples:
  - 0-1 Knapsack (0–1背包)
  - Rod-Cutting (钢条切割)
  - Chain Matrix Multiplication (矩阵链乘法)
  - Longest Common Subsequences (最长公共子序列)
  - Minimum Edit Distance (最小编辑距离)
  - Optimal Binary Search Trees (最优二叉查找树)

Review to Divide-and-Conquer (DC)

- Divide-and-conquer (DC) is an important algorithm design paradigm.
  - **Divide**

    Dividing a given problem into two or more subproblems (ideally of approximately equal size)
  - **Conquer**

    Solving each subproblem (directly if small enough or recursively)
  - **Combine**

    Combining the solutions of the subproblems into a global solution

Review to Dynamic Programming (DP)

- Dynamic Programming (DP) is similar to Divide and Conquer (D&C)
  - They both partition a problem into smaller subproblems
- DP is preferable when the subproblems overlap, i.e., they share common subproblems
- Often DP is used for optimization problems
  - Problems that have many solutions, and we want to find the best one
- Main idea of DP
  - Analyze the structure of an optimal solution
  - Recursively define the value of an optimal solution
  - Compute the value of an optimal solution (usually bottom-up)

Comparison of DC and DP

- Commonalities:
  - Partition the problem into particular subproblems.
  - Solve the subproblems.
  - Combine the solutions to solve the original one.
- Differences:
  - DC:
    - Efficient when the subproblems are independent.
    - Not efficient when subproblems share subsubproblems.
    - Some subproblems might be solved many times.
  - DP:
    - Suitable when subproblems share subsubproblems.
    - Do each subproblem only once.
    - The result is stored in a table in case it is needed elsewhere.
    - DP trades space for time.

Outline

- Sorting in Linear Time
  - Counting Sort
- Randomized Selection Problem
  - Problem Definition
  - First solution: Selection by sorting
  - A divide-and-conquer algorithm
- Optimal Binary Search Tree Problem
  - Review of Binary Search Tree
  - Problem Definition
  - A Dynamic Programming Algorithm

Outline

- Sorting in Linear Time
  - Counting Sort

- Randomized Selection Problem
  - Problem Definition
  - First solution: Selection by sorting
  - A divide-and-conquer algorithm

- Optimal Binary Search Tree Problem
  - Review of Binary Search Tree
  - Problem Definition
  - A Dynamic Programming Algorithm

Review of Comparison-based Sorting

- All sorting algorithms seen so far are based on comparing elements
  - E.g., insertion sort, merge sort and heapsort
- Insertion sort has worst-case running time $\Theta(n^2)$, while the others have worst-case running time $\Theta(n \log n)$

Question

Can we do better?

Goal

We will prove that any comparison-based sorting algorithm has a worst-case running time $\Omega(n \log n)$.

Can we do better?

Are there sorting algorithms which are not based on comparisons? Do they beat the $\Omega(n \log n)$ lower bound?

Can we do better?

Are there sorting algorithms which are not based on comparisons? Do they beat the $\Omega(n \log n)$ lower bound?

- Counting sort (计数排序)
- Radix sort (基数排序)

Outline

- Sorting in Linear Time
  - Counting Sort
  - Radix Sort
- Randomized Selection Problem
  - Problem Definition
  - First solution: Selection by sorting
  - A divide-and-conquer algorithm
- AVL Tree
  - Binary Search Tree and AVL Tree
  - Insertion/Deletion Operations of AVL Tree
- Optimal Binary Search Tree Problem
  - Problem Definition
  - A Dynamic Programming Algorithm

Main Ideas

- Counting sort determines, for each input element x, the number of elements less than x.

Main Ideas

- Counting sort determines, for each input element *x*, the number of elements less than *x*.
- It uses this information to place element *x* directly into its position in the output array.

Main Ideas

- Counting sort determines, for each input element *x*, the number of elements less than *x*.

- It uses this information to place element *x* directly into its position in the output array.
  - For example, if 17 elements are less than *x*, then *x* belongs in output position 18.

Counting Sort

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
  $C[i] \leftarrow 0;$
end
for $j \leftarrow 1$ *to* n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$
end
for $i \leftarrow 2$ *to* k **do**
  $C[i] \leftarrow C[i] + C[i-1]; // C[i] = |\{key \leq i\}|$
end
for $j \leftarrow n$ *to* 1 **do**
  $B[C[A[j]]] \leftarrow A[j];$
  $C[A[j]] \leftarrow C[A[j]] - 1;$
end
return B;

Counting Sort

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1,2,...,k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
$\quad C[i] \leftarrow 0;$
end
for $j \leftarrow 1$ *to* n **do**
$\quad C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$
end
for $i \leftarrow 2$ *to* k **do**
$\quad C[i] \leftarrow C[i] + C[i-1]; // C[i] = |\{key \leq i\}|$
end
for $j \leftarrow n$ *to* 1 **do**
$\quad B[C[A[j]]] \leftarrow A[j];$
$\quad C[A[j]] \leftarrow C[A[j]] - 1;$
end
return B;

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C				

B					

Counting Sort

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted

```
let C[1...k] be a new array;
for i ← 1 to k do
|   C[i] ← 0;
end
for j ← 1 to n do
|   C[A[j]] ← C[A[j]] + 1;//C[i] = |{key = i}|
end
for i ← 2 to k do
|   C[i] ← C[i] + C[i − 1];//C[i] = |{key ≤ i}|
end
for j ← n to 1 do
|   B[C[A[j]]] ← A[j];
|   C[A[j]] ← C[A[j]] − 1;
end
return B;
```

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	0	0	0	0

B					

for $i \leftarrow 1$ *to* k **do**
| $C[i] \leftarrow 0;$
end

Counting Sort

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
    $C[i] \leftarrow 0;$
end
for $j \leftarrow 1$ *to* n **do**
    $C[A[j]] \leftarrow C[A[j]] + 1;//C[i] = |\{key = i\}|$
end
for $i \leftarrow 2$ *to* k **do**
    $C[i] \leftarrow C[i] + C[i-1];//C[i] = |\{key \leq i\}|$
end
for $j \leftarrow n$ *to* 1 **do**
    $B[C[A[j]]] \leftarrow A[j];$
    $C[A[j]] \leftarrow C[A[j]] - 1;$
end
return B;

Example: Counting Sort

	1	2	3	4	5
A	**4**	2	1	4	2

	1	2	3	4
C	0	0	0	**1**

B					

for $j \leftarrow 1$ *to* n **do**

$\quad C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$

end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	0	1	0	1

B					

for $j \leftarrow 1$ to n **do**
 $C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$
end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	1	0	1

B					

for $j \leftarrow 1$ *to* n **do**

$\mid \quad C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$

end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	1	0	2

B					

for $j \leftarrow 1$ *to* n **do**
$\quad C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$
end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	2	0	2

B					

for $j \leftarrow 1$ *to* n **do**
| $C[A[j]] \leftarrow C[A[j]] + 1;$ $//C[i] = |\{key = i\}|$
end

Counting Sort

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
  $C[i] \leftarrow 0;$
end
for $j \leftarrow 1$ *to* n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$
end
for $i \leftarrow 2$ *to* k **do**
  $C[i] \leftarrow C[i] + C[i-1]; // C[i] = |\{key \leq i\}|$
end
for $j \leftarrow n$ *to* 1 **do**
  $B[C[A[j]]] \leftarrow A[j];$
  $C[A[j]] \leftarrow C[A[j]] - 1;$
end
return B;

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	2	0	2

B					

C'	1	3	0	2

for $i \leftarrow 2$ *to* k **do**
| $C[i] \leftarrow C[i] + C[i-1]; // C[i] = |\{key \leq i\}|$
end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	2	0	2

B					

C'	1	3	3	2

for $i \leftarrow 2$ *to* k **do**
| $C[i] \leftarrow C[i] + C[i-1]$; $//C[i] = |\{key \leq i\}|$
end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2
B					

	1	2	3	4
C	1	2	0	2
C'	1	3	3	5

for $i \leftarrow 2$ *to* k **do**

$\quad C[i] \leftarrow C[i] + C[i-1]; // C[i] = |\{key \leq i\}|$

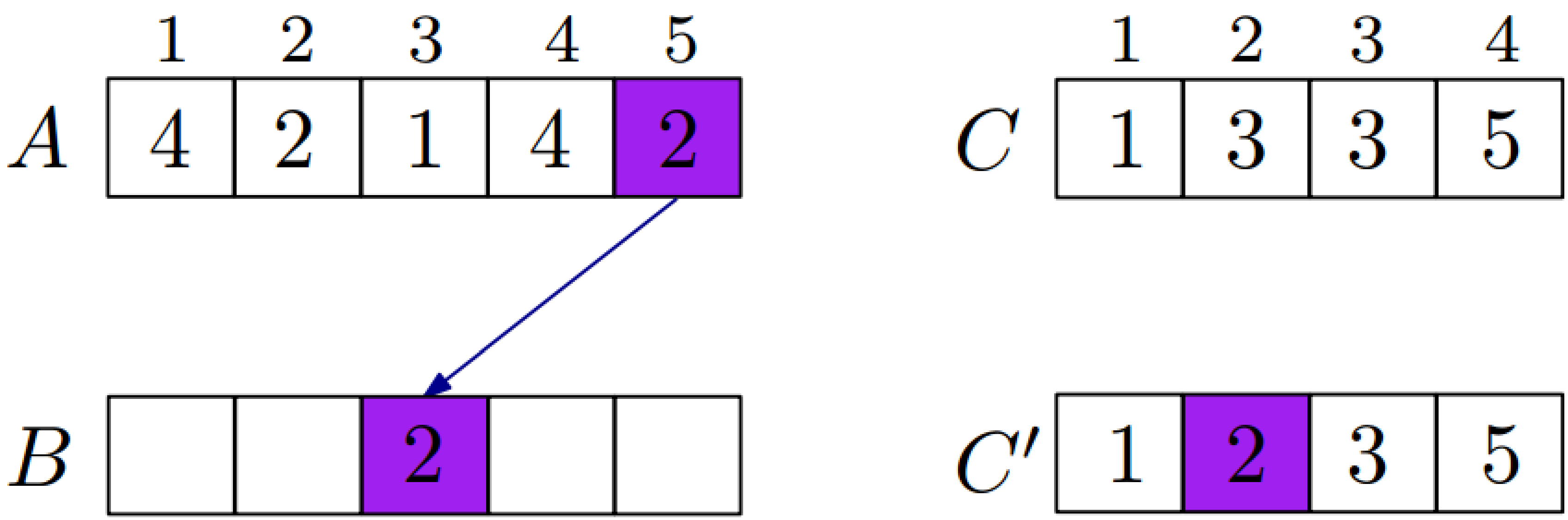
end

Counting Sort

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
  $C[i] \leftarrow 0;$
end
for $j \leftarrow 1$ *to* n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$
end
for $i \leftarrow 2$ *to* k **do**
  $C[i] \leftarrow C[i] + C[i-1]; // C[i] = |\{key \leq i\}|$
end
for $j \leftarrow n$ *to* 1 **do**
  $B[C[A[j]]] \leftarrow A[j];$
  $C[A[j]] \leftarrow C[A[j]] - 1;$
end
return B;

Example: Counting Sort



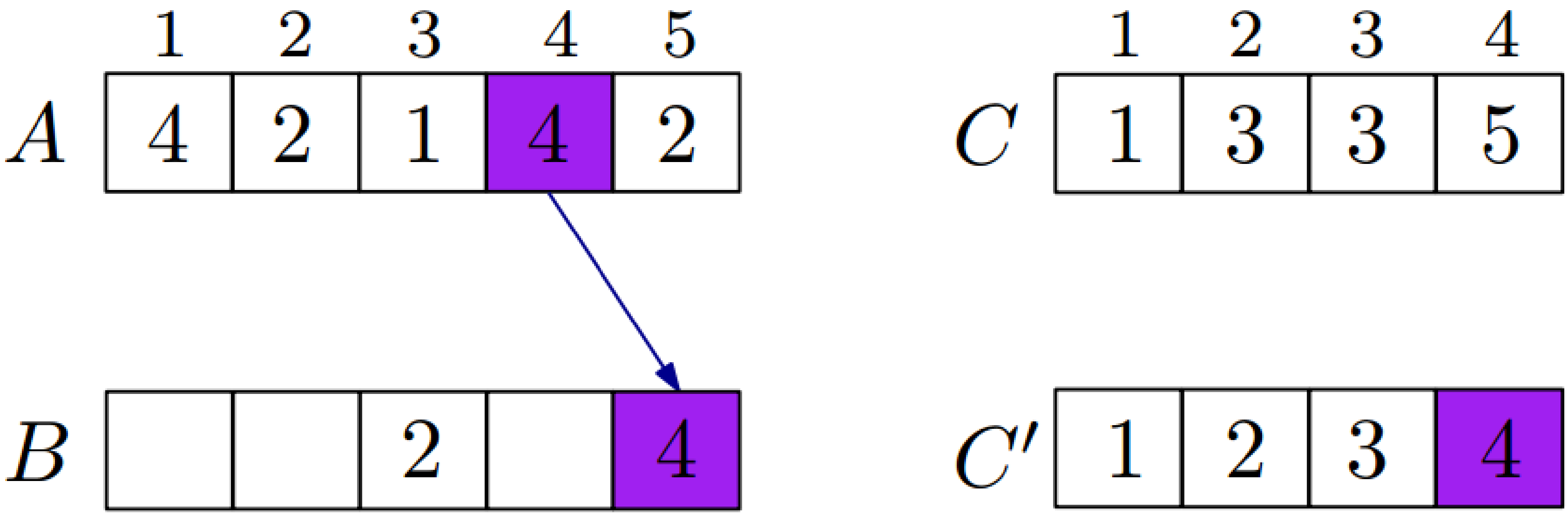
for $j \leftarrow n$ *to* 1 **do**

$\quad B[C[A[j]]] \leftarrow A[j];$

$\quad C[A[j]] \leftarrow C[A[j]] - 1;$

end

Example: Counting Sort



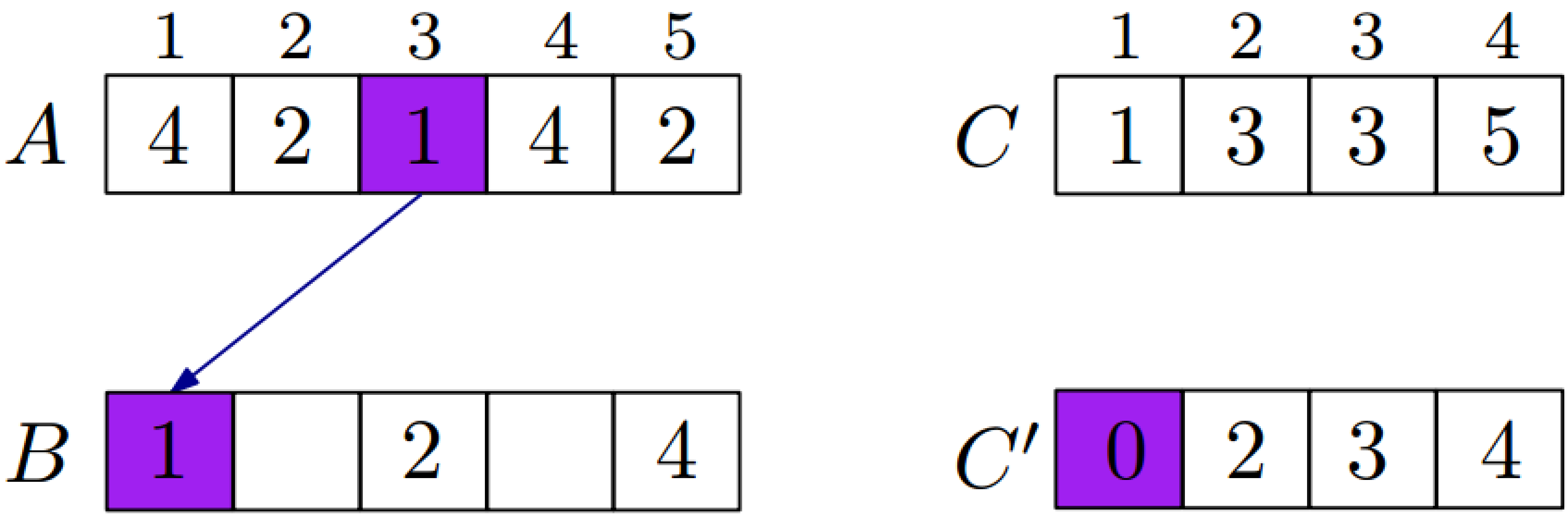
for $j \leftarrow n$ *to* 1 **do**

$\quad B[C[A[j]]] \leftarrow A[j];$

$\quad C[A[j]] \leftarrow C[A[j]] - 1;$

end

Example: Counting Sort



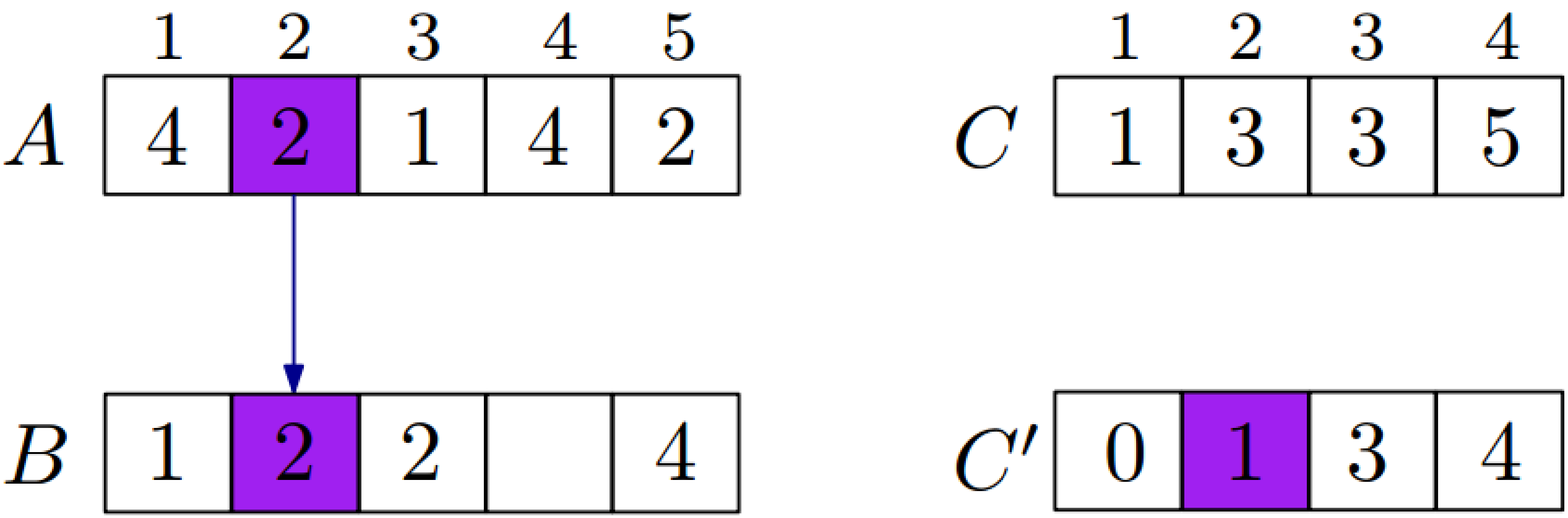
for $j \leftarrow n$ *to* 1 **do**

$B[C[A[j]]] \leftarrow A[j]$;

$C[A[j]] \leftarrow C[A[j]] - 1$;

end

Example: Counting Sort



for $j \leftarrow n$ *to* 1 **do**

  $B[C[A[j]]] \leftarrow A[j]$;

  $C[A[j]] \leftarrow C[A[j]] - 1$;

end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	3	3	5

B	1	2	2	4	4

C'	0	1	3	3

for $j \leftarrow n$ *to* 1 **do**

$\quad B[C[A[j]]] \leftarrow A[j];$

$\quad C[A[j]] \leftarrow C[A[j]] - 1;$

end

Analysis

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
| $C[i] \leftarrow 0;//O(k)$
end

return B;

Analysis

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
| $C[i] \leftarrow 0$;$//O(k)$
end
for $j \leftarrow 1$ *to* n **do**
| $C[A[j]] \leftarrow C[A[j]] + 1$;$//O(n)$
end

return B;

Analysis

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
| $C[i] \leftarrow 0;//O(k)$
end
for $j \leftarrow 1$ *to* n **do**
| $C[A[j]] \leftarrow C[A[j]] + 1;//O(n)$
end
for $i \leftarrow 2$ *to* k **do**
| $C[i] \leftarrow C[i] + C[i-1];//O(k)$
end

return B;

Analysis

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
  $C[i] \leftarrow 0;//O(k)$
end
for $j \leftarrow 1$ *to* n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1;//O(n)$
end
for $i \leftarrow 2$ *to* k **do**
  $C[i] \leftarrow C[i] + C[i-1];//O(k)$
end
for $j \leftarrow n$ *to* 1 **do**
  $B[C[A[j]]] \leftarrow A[j];$
  $C[A[j]] \leftarrow C[A[j]] - 1;//O(n)$
end
return B;

Analysis

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
| $C[i] \leftarrow 0; //O(k)$
end
for $j \leftarrow 1$ *to* n **do**
| $C[A[j]] \leftarrow C[A[j]] + 1; //O(n)$
end
for $i \leftarrow 2$ *to* k **do**
| $C[i] \leftarrow C[i] + C[i-1]; //O(k)$
end
for $j \leftarrow n$ *to* 1 **do**
| $B[C[A[j]]] \leftarrow A[j];$
| $C[A[j]] \leftarrow C[A[j]] - 1; //O(n)$
end
return B;

Total: $O(n+k)$

Running Time

If k = O(n), then counting sort takes O(n) time.

- But didn't we prove that sorting must take $\Omega(n \log n)$ time?

Running Time

If $k = O(n)$, then counting sort takes $O(n)$ time.

- But didn't we prove that sorting must take $\Omega(n \log n)$ time?
- No, actually we proved that any comparison-based sorting algorithm takes $\Omega(n \log n)$ time.

Running Time

If k = O(n), then counting sort takes O(n) time.

- But didn't we prove that sorting must take Ω(n log n) time?
- No, actually we proved that any comparison-based sorting algorithm takes Ω(n log n) time.
- Note that counting sort is not a comparison-based sorting algorithm.

Running Time

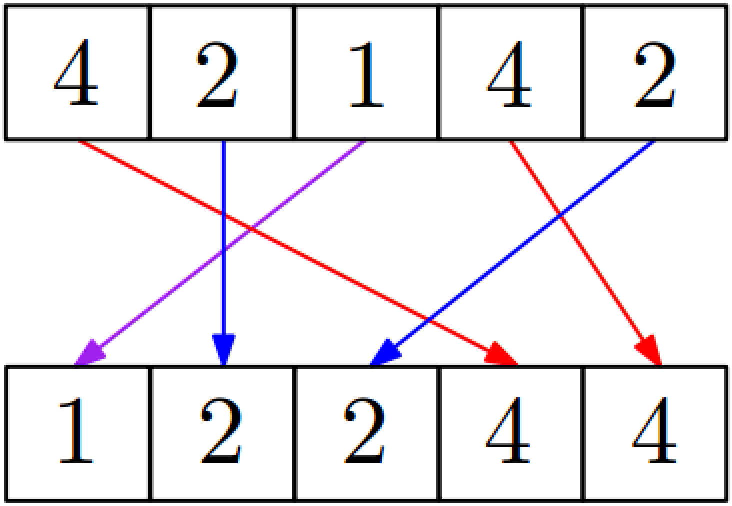
If $k = O(n)$, then counting sort takes $O(n)$ time.

- But didn't we prove that sorting must take $\Omega(n \log n)$ time?
- No, actually we proved that any comparison-based sorting algorithm takes $\Omega(n \log n)$ time.
- Note that counting sort is not a comparison-based sorting algorithm.
- In fact, it makes no comparison at all!

Stable Sorting

Counting sort is a stable sort

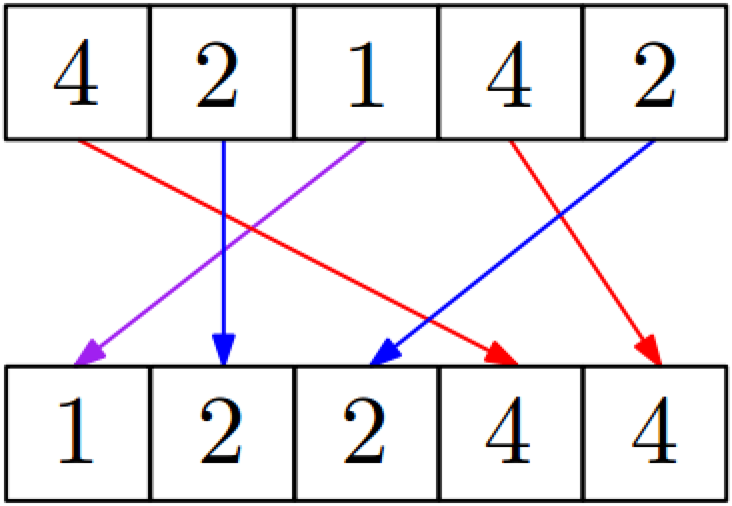
- it preserves the input order among equal elements.



Stable Sorting

Counting sort is a stable sort

- it preserves the input order among equal elements.



Exercise

What other sorts have this property?

Outline

- Sorting in Linear Time
  - Counting Sort

- Randomized Selection Problem
  - Problem Definition
  - First solution: Selection by sorting
  - A divide-and-conquer algorithm

- Optimal Binary Search Tree Problem
  - Review of Binary Search Tree
  - Problem Definition
  - A Dynamic Programming Algorithm

Linear Time Selection

Definition (Selection Problem)

Given a sequence of numbers $\langle a_1, \ldots, a_n \rangle$, and an integer i, $1 \le i \le n$, find the ith smallest element. When $i = \lceil n/2 \rceil$, it is called the median problem.

Linear Time Selection

Definition (Selection Problem)

Given a sequence of numbers $\langle a_1, \ldots, a_n \rangle$, and an integer i, $1 \le i \le n$, find the ith smallest element. When $i = \lceil n/2 \rceil$, it is called the median problem.

Example

Given $\langle 1, 8, 23, 10, 19, 33, 100 \rangle$, the 4th smallest element is 19.

Linear Time Selection

Definition (Selection Problem)

Given a sequence of numbers $\langle a_1, \ldots, a_n \rangle$, and an integer i, $1 \le i \le n$, find the ith smallest element. When $i = \lceil n/2 \rceil$, it is called the median problem.

Example

Given $\langle 1, 8, 23, 10, 19, 33, 100 \rangle$, the 4th smallest element is 19.

Question

How do you solve this problem?

Outline

- Sorting in Linear Time
  - Counting Sort

- Randomized Selection Problem
  - Problem Definition
  - First solution: Selection by sorting
  - A divide-and-conquer algorithm

- Optimal Binary Search Tree Problem
  - Review of Binary Search Tree
  - Problem Definition
  - A Dynamic Programming Algorithm

First Solution: Selection by Sorting

- Sort the elements in ascending order with any algorithm of complexity O(n log n).

First Solution: Selection by Sorting

- Sort the elements in ascending order with any algorithm of complexity O(n log n).
- Return the *i*th element of the sorted array.

First Solution: Selection by Sorting

- Sort the elements in ascending order with any algorithm of complexity O(n log n).
- Return the *i*th element of the sorted array.

The complexity of this solution is O(   )

First Solution: Selection by Sorting

- Sort the elements in ascending order with any algorithm of complexity O(n log n).
- Return the *i*th element of the sorted array.

The complexity of this solution is O(n log n)

First Solution: Selection by Sorting

- Sort the elements in ascending order with any algorithm of complexity O(n log n).
- Return the *i*th element of the sorted array.

The complexity of this solution is O(n log n)

Question

Can we do better?

First Solution: Selection by Sorting

- Sort the elements in ascending order with any algorithm of complexity O(n log n).
- Return the *i*th element of the sorted array.

The complexity of this solution is O(n log n)

Question

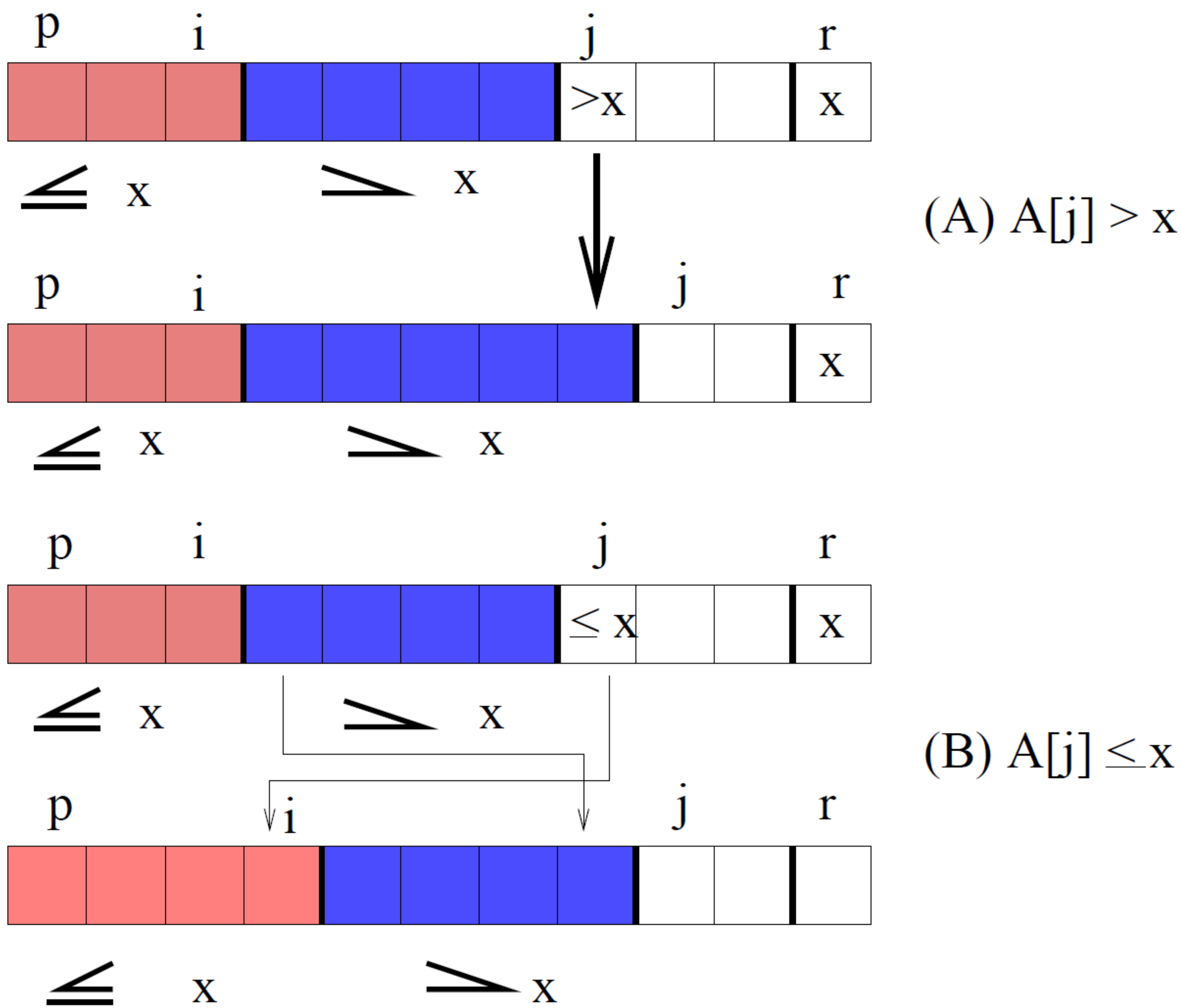
Can we do better?

Answer: YES, but we need to recall Partition(A,p,r) used in Quicksort!

Outline

- Sorting in Linear Time
  - Counting Sort
- Randomized Selection Problem
  - Problem Definition
  - First solution: Selection by sorting
  - A divide-and-conquer algorithm
- Optimal Binary Search Tree Problem
  - Review of Binary Search Tree
  - Problem Definition
  - A Dynamic Programming Algorithm

Review of Randomized-Partition (A,p,r)



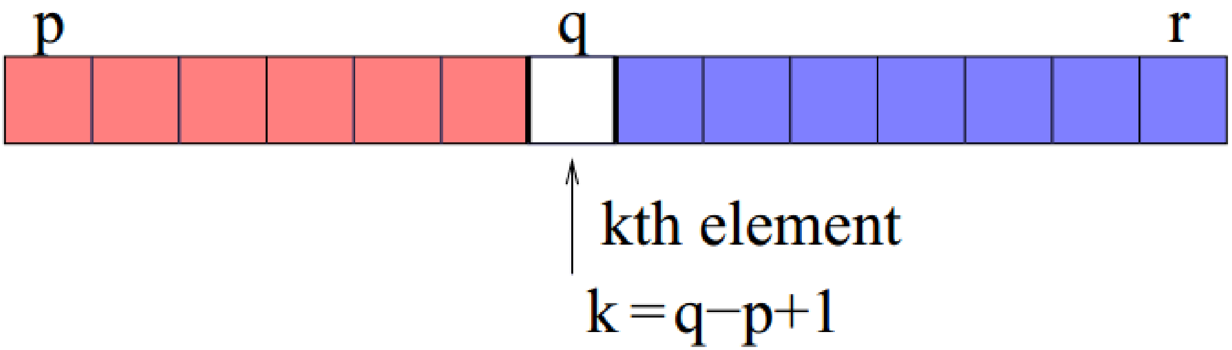
Randomized-Select(A,p,r,i), $1 \le i \le r-p+1$

Problem: Select the *i*th smallest element in A[p..r], where
$1 \le i \le r-p+1$

Randomized-Select(A,p,r,i), 1≤i≤r−p+1

Problem: Select the *i*th smallest element in A[p..r], where 1 ≤ i ≤ r−p+1

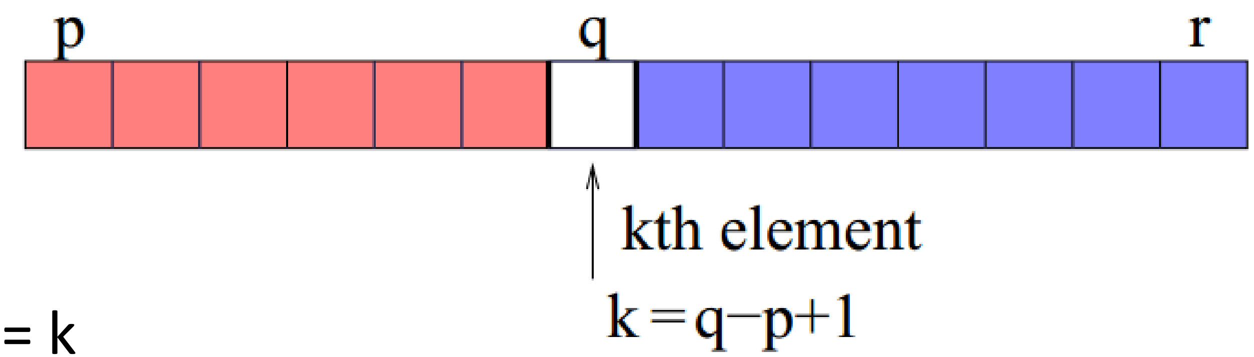
Solution: Apply Randomized-Partition(A, p, r), getting



Randomized-Select(A,p,r,i), $1 \le i \le r-p+1$

Problem: Select the *i*th smallest element in A[p..r], where $1 \le i \le r-p+1$

Solution: Apply Randomized-Partition(A, p, r), getting

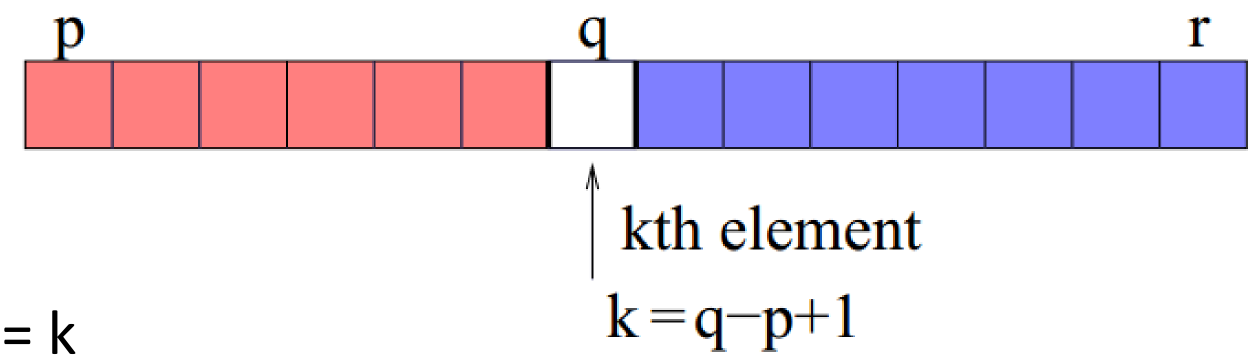


- i = k
  - pivot is the solution

Randomized-Select(A,p,r,i), 1≤i≤r−p+1

Problem: Select the *i*th smallest element in A[p..r], where 1 ≤ i ≤ r−p+1

Solution: Apply Randomized-Partition(A, p, r), getting

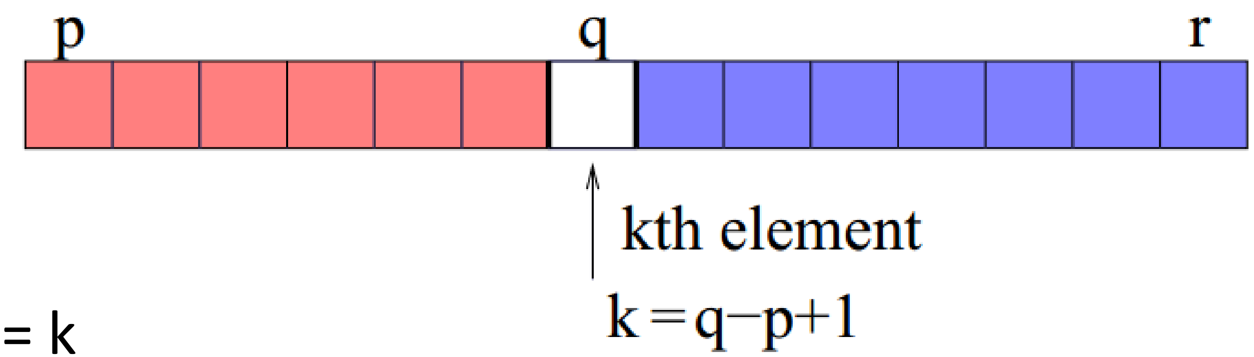


- i = k
  - pivot is the solution
- i < k
  - the *i*th smallest element in A[p..r] must be the *i*th smallest element in A[p..q−1]

Randomized-Select(A,p,r,i), $1 \le i \le r-p+1$

Problem: Select the *i*th smallest element in A[p..r], where $1 \le i \le r-p+1$

Solution: Apply Randomized-Partition(A, p, r), getting



- $i = k$
  - pivot is the solution
- $i < k$
  - the *i*th smallest element in A[p..r] must be the *i*th smallest element in A[p..q−1]
- $i > k$
  - the *i*th smallest element in A[p..r] must be the (*i* − *k*)th smallest element in A[q+1..r]

If necessary, recursively call the same procedure to the subarray

Randomized-Select(A,p,r,i), 1≤i≤r−p+1

Randomized-Select(A,p,r,i)

Input: An array $\boldsymbol{A}$,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$,the $\boldsymbol{i}$th smallest element that we want to select

Output: The ith smallest element $\boldsymbol{A}[\boldsymbol{i}]$

Randomized-Select(A,p,r,i), 1≤i≤r−p+1

Randomized-Select(A,p,r,i)

Input: An array $\boldsymbol{A}$,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$,the $\boldsymbol{i}$th smallest element that we want to select

Output: The ith smallest element $\boldsymbol{A}[\boldsymbol{i}]$

if p *is equal to* r **then**

  | **return** $A[p]$;

end

Randomized-Select(A,p,r,i), 1≤i≤r−p+1

Randomized-Select(A,p,r,i)

Input: An array $\boldsymbol{A}$,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$,the $\boldsymbol{i}$th smallest element that we want to select

Output: The ith smallest element $\boldsymbol{A}[\boldsymbol{i}]$

if p *is equal to* r **then**

  return $A[p]$;

end

$q \leftarrow$Randomized-Partition(A, p, r);

Randomized-Select(A,p,r,i), 1≤i≤r−p+1

Randomized-Select(A,p,r,i)

Input: An array $\boldsymbol{A}$,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$,the $\boldsymbol{i}$th smallest element that we want to select

Output: The ith smallest element $\boldsymbol{A}[\boldsymbol{i}]$

if p *is equal to* r **then**

  return $A[p]$;

end

$q \leftarrow$Randomized-Partition(A, p, r);

$k \leftarrow q - p + 1$;

if $i \leftarrow k$ **then**

Randomized-Select(A,p,r,i), 1≤i≤r−p+1

Randomized-Select(A,p,r,i)

Input: An array $\boldsymbol{A}$,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$,the $\boldsymbol{i}$th smallest element that we want to select

Output: The ith smallest element $\boldsymbol{A}[\boldsymbol{i}]$

if p *is equal to* r **then**
  | **return** $A[p]$;
end
$q \leftarrow$Randomized-Partition(A, p, r);
$k \leftarrow q - p + 1$;
if $i \leftarrow k$ **then**
  | **return** $A[q]$;//The pivot is the answer
end

Randomized-Select(A,p,r,i), 1≤i≤r−p+1

Randomized-Select(A,p,r,i)

Input: An array $\boldsymbol{A}$,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$,the $\boldsymbol{i}$th smallest element that we want to select

Output: The ith smallest element $\boldsymbol{A}[\boldsymbol{i}]$

```
if p is equal to r then
|  return A[p];
end
q ←Randomized-Partition(A, p, r);
k ← q − p + 1;
if i ← k then
|  return A[q];//The pivot is the answer
end
else if i < k then
|  return Randomized-Select(A, p, q − 1, i);
end
```

Randomized-Select(A,p,r,i), 1≤i≤r−p+1

Randomized-Select(A,p,r,i)

Input: An array $\boldsymbol{A}$,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$,the $\boldsymbol{i}$th smallest element that we want to select

Output: The ith smallest element $\boldsymbol{A}[\boldsymbol{i}]$

if p *is equal to* r **then**
  | **return** $A[p]$;
end
$q \leftarrow$Randomized-Partition(A, p, r);
$k \leftarrow q - p + 1$;
if $i \leftarrow k$ **then**
  | **return** $A[q]$;//The pivot is the answer
end
else if $i < k$ **then**
  | **return** Randomized-Select$(A, p, q - 1, i)$;
end
else
  | **return** Randomized-Select$(A, q + 1, r, i - k)$;
end

Randomized-Select(A,p,r,i), 1≤i≤r−p+1

Randomized-Select(A,p,r,i)

```
Input: An array A,the range of index p,r,the ith smallest element that
       we want to select
Output: The ith smallest element A[i]
if p is equal to r then
 |  return A[p];
end
q ←Randomized-Partition(A, p, r);
k ← q − p + 1;
if i ← k then
 |  return A[q];//The pivot is the answer
end
else if i < k then
 |  return Randomized-Select(A, p, q − 1, i);
end
else
 |  return Randomized-Select(A, q + 1, r, i − k);
end
```

To find the ith smallest element in A[1..n], call Randomized-Select(A, 1, n, i)

Randomized Selection - Example

- Find the 8th smallest element of the following list of numbers:
  - 8 25 2 14 3 20 15 13 12 11 7 5 9 10 16 18 1 23 26 21

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i-k*)th smallest element in A[q+1..r]

p																			r
8	25	2	14	3	20	15	13	12	11	7	5	9	10	16	18	1	23	26	21

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i-k*)th smallest element in A[q+1..r]

p																			r
8	**25**	**2**	**14**	**3**	**20**	**15**	**13**	**12**	**11**	**7**	**5**	**9**	**10**	**16**	**18**	**1**	**23**	**26**	**21**

$$i = 8, p = 1, r = 20$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i*-*k*)th smallest element in A[q+1..r]

p																			r
8	**25**	**2**	**14**	**3**	**20**	**15**	**13**	**12**	**11**	**7**	**5**	**9**	**10**	**16**	**18**	**1**	**23**	**26**	**21**

$$i = 8, p = 1, r = 20$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i*-*k*)th smallest element in A[q+1..r]

p															q				r
8	2	14	3	15	13	12	11	7	5	9	10	16	18	1	20	25	23	26	21

$$i = 8, p = 1, r = 20$$
$$q = 16, \mathrm{k} = 16$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i*-*k*)th smallest element in A[q+1..r]

p															q				r
8	2	14	3	15	13	12	11	7	5	9	10	16	18	1	20	25	23	26	21

$$i = 8, p = 1, r = 20$$
$$q = 16, \mathrm{k} = 16$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i-k*)th smallest element in A[q+1..r]

p														r					
8	2	14	3	15	13	12	11	7	5	9	10	16	18	1	20	25	23	26	21

$$i = 8, p = 1, r = 15$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i*-*k*)th smallest element in A[q+1..r]

p														r					
8	2	14	3	15	13	12	11	7	5	9	10	16	18	1	20	25	23	26	21

$$i = 8, p = 1, r = 15$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i*-*k*)th smallest element in A[q+1..r]

p						q								r					
8	2	3	7	5	1	9	11	14	15	13	10	16	18	12	20	25	23	26	21

$$i = 8, p = 1, r = 15$$
$$q = 7, k = 7$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i-k*)th smallest element in A[q+1..r]

p						q								r					
8	2	3	7	5	1	9	11	14	15	13	10	16	18	12	20	25	23	26	21

$$i = 8, p = 1, r = 15$$
$$q = 7, k = 7$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i-k*)th smallest element in A[q+1..r]

							p							r					
8	2	3	7	5	1	9	**11**	**14**	**15**	**13**	**10**	**16**	**18**	**12**	20	25	23	26	21

$$i = 1, p = 8, r = 15$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i*-*k*)th smallest element in A[q+1..r]

							p							r					
8	2	3	7	5	1	9	**11**	**14**	**15**	**13**	**10**	**16**	**18**	**12**	20	25	23	26	21

$$i = 1, p = 8, r = 15$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i*-*k*)th smallest element in A[q+1..r]

							p			q				r					
8	2	3	7	5	1	9	**11**	**12**	**10**	**13**	**15**	**14**	**18**	**16**	20	25	23	26	21

$$i = 1, p = 8, r = 15$$
$$q = 11, k = 4$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i*-*k*)th smallest element in A[q+1..r]

							p			q				r					
8	2	3	7	5	1	9	**11**	**12**	**10**	**13**	**15**	**14**	**18**	**16**	20	25	23	26	21

$$i = 1, p = 8, r = 15$$
$$q = 11, k = 4$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i-k*)th smallest element in A[q+1..r]

							p		r										
8	2	3	7	5	1	9	**11**	**12**	**10**	13	15	14	18	16	20	25	23	26	21

$$i = 1, p = 8, r = 10$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i*-*k*)th smallest element in A[q+1..r]

							p		r										
8	2	3	7	5	1	9	**11**	**12**	**10**	13	15	14	18	16	20	25	23	26	21

$$i = 1, p = 8, r = 10$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i-k*)th smallest element in A[q+1..r]

							p,q		r										
8	2	3	7	5	1	9	10	12	11	13	15	14	18	16	20	25	23	26	21

$$i = 1, p = 8, r = 10$$
$$q = 8, k = 1$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i-k*)th smallest element in A[q+1..r]

							p,q		r										
8	2	3	7	5	1	9	10	**12**	**11**	13	15	14	18	16	20	25	23	26	21

$$i = 1, p = 8, r = 10$$
$$q = 8, k = 1$$

Randomized Selection - Example

- Select the *i*th smallest element in A[p..r], pivot is A[q], k = q-p+1.
  - i = k : pivot is the solution
  - i < k : the *i*th smallest element in A[p..q-1]
  - i > k : the (*i-k*)th smallest element in A[q+1..r]

							p,q		r										
8	2	3	7	5	1	9	10	12	11	13	15	14	18	16	20	25	23	26	21

$$i = 1, p = 8, r = 10$$
$$q = 8, k = 1$$

10 is the 8th smallest element of the array.

Randomized Quicksort vs Randomized Selection

Question

Why does Randomized Selection take $O(n)$ time while Randomized Quicksort takes $O(n \log n)$ time?

Randomized Quicksort vs Randomized Selection

Question

Why does Randomized Selection take $O(n)$ time while Randomized Quicksort takes $O(n \log n)$ time?

Answer:

- Randomized Selection needs to work on only 1 of the two subproblems.

Randomized Quicksort vs Randomized Selection

Question

Why does Randomized Selection take $O(n)$ time while Randomized Quicksort takes $O(n \log n)$ time?

Answer:

- Randomized Selection needs to work on only 1 of the two subproblems.
- Randomized Quicksort needs to work on both of the two subproblems.

Outline

- Sorting in Linear Time
  - Counting Sort
- Randomized Selection Problem
  - Problem Definition
  - First solution: Selection by sorting
  - A divide-and-conquer algorithm
- Optimal Binary Search Tree Problem
  - Review of Binary Search Tree
  - Problem Definition
  - A Dynamic Programming Algorithm

Binary Search Trees(BST)

A BST on a set S of n integers is a binary tree T satisfying all the following requirements:

Binary Search Trees(BST)

A BST on a set S of n integers is a binary tree T satisfying all the following requirements:

- T has n nodes.

Binary Search Trees(BST)

A BST on a set S of n integers is a binary tree T satisfying all the following requirements:

- T has n nodes.
- Each node u in T stores a distinct integer in S, which is called the key of u.
- For every internal u, it holds that:

Binary Search Trees(BST)

A BST on a set S of n integers is a binary tree T satisfying all the following requirements:

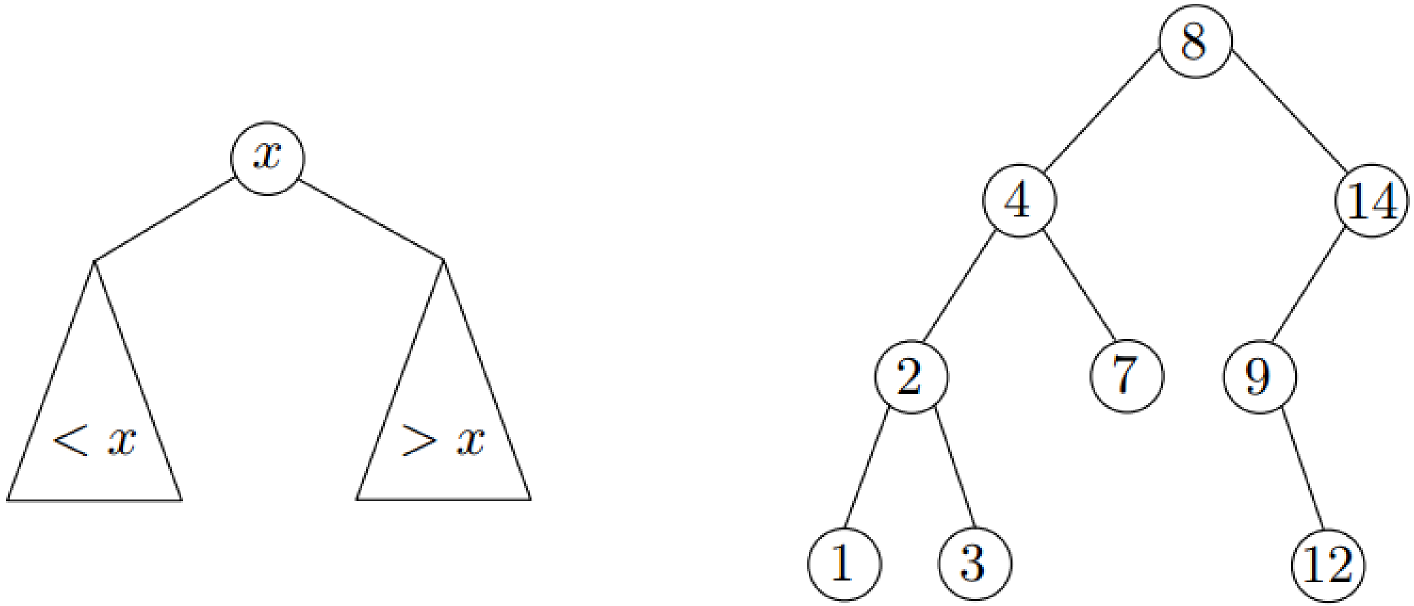
- T has n nodes.
- Each node u in T stores a distinct integer in S, which is called the key of u.
- For every internal u, it holds that:
  - The key of u is larger than all the keys in the left subtree of u.

Binary Search Trees(BST)

A BST on a set S of n integers is a binary tree T satisfying all the following requirements:

- T has n nodes.
- Each node u in T stores a distinct integer in S, which is called the key of u.
- For every internal u, it holds that:
  - The key of u is larger than all the keys in the left subtree of u.
  - The key of u is smaller than all the keys in the right subtree of u.

Binary Search Trees(BST)



Binary-search-tree property

For every node x

- All keys in its left subtree are smaller than the key value in x
- All keys in its right subtree are larger than the key value in x

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  = max(children height) +1

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  = max(children height) +1
- Leaves: height = 0

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  = max(children height) +1
- Leaves: height = 0
- Tree height = root height

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  = max(children height) +1
- Leaves: height = 0
- Tree height = root height
- Empty tree: height = −1

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

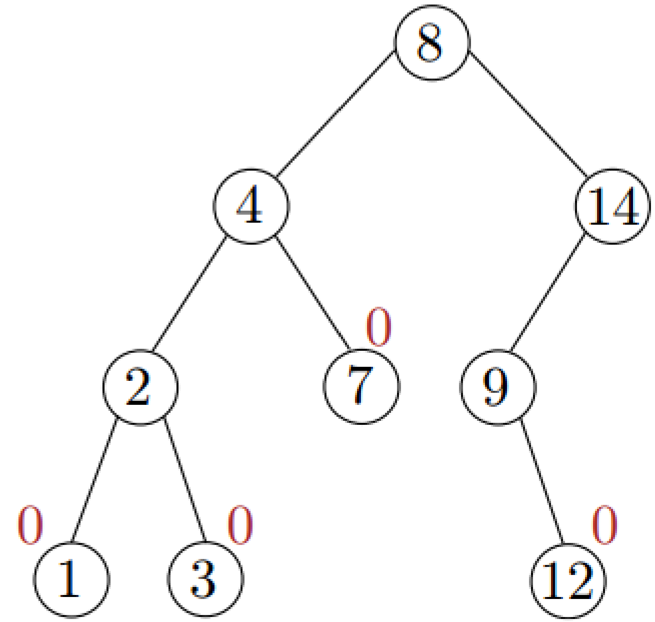
- Node height
  = max(children height) +1
- Leaves: height = 0
- Tree height = root height
- Empty tree: height = −1

- Tree operations typically take O(height) time

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  = max(children height) +1
- Leaves: height = 0
- Tree height = root height
- Empty tree: height = −1

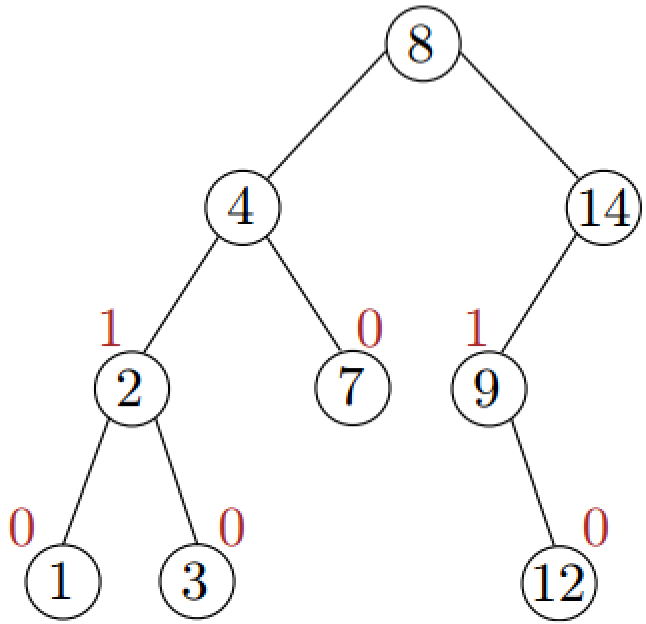


- Tree operations typically take O(height) time

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height = max(children height) +1
- Leaves: height = 0
- Tree height = root height
- Empty tree: height = −1

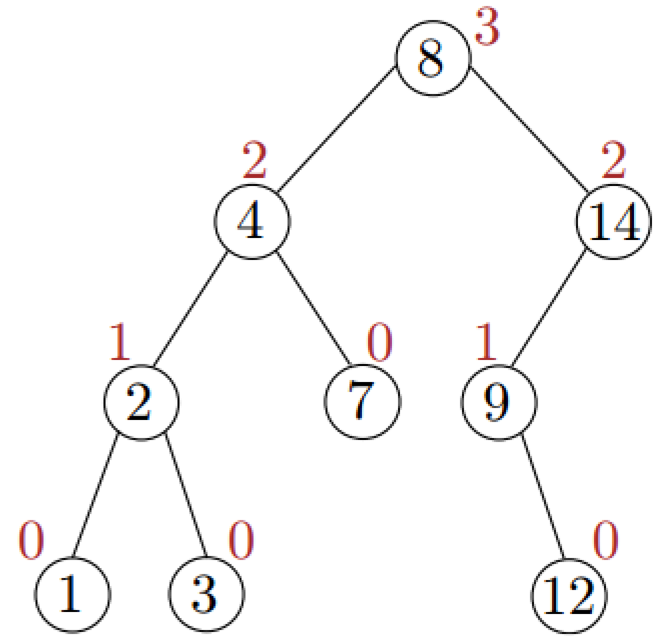


- Tree operations typically take O(height) time

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height = max(children height) +1
- Leaves: height = 0
- Tree height = root height
- Empty tree: height = −1



- Tree operations typically take O(height) time

Balanced Binary Search Tree

- A binary tree T is balanced if the following holds on every internal node u of T :

Balanced Binary Search Tree

- A binary tree T is balanced if the following holds on every internal node u of T :
  - The height of the left subtree of u differs from that of the right subtree of u by at most 1.

Balanced Binary Search Tree

- A binary tree T is balanced if the following holds on every internal node u of T :
  - The height of the left subtree of u differs from that of the right subtree of u by at most 1.
- If u violates the above requirement, we say that u is imbalanced.

Balanced Binary Search Tree

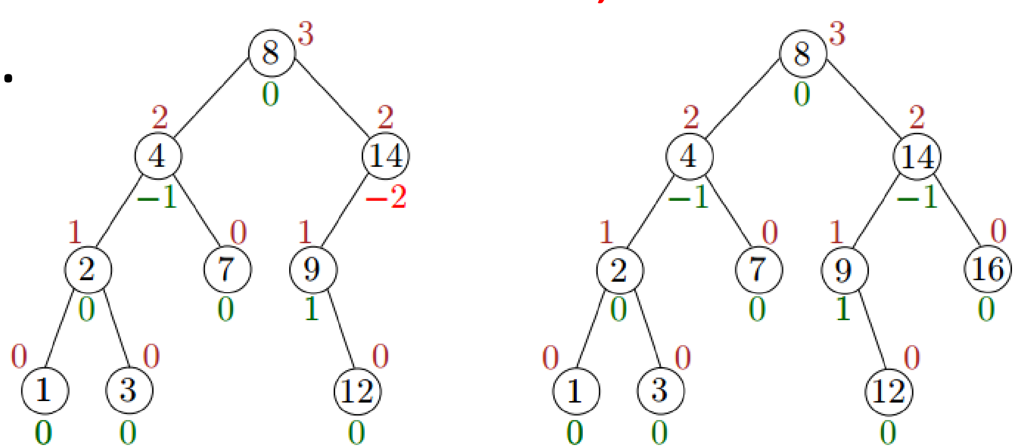
- A binary tree T is balanced if the following holds on every internal node u of T :
  - The height of the left subtree of u differs from that of the right subtree of u by at most 1.
- If u violates the above requirement, we say that u is imbalanced.
- The balance factor of a node is the height of its right subtree minus the height of its left subtree.

Balanced Binary Search Tree

- A binary tree T is balanced if the following holds on every internal node u of T :
  - The height of the left subtree of u differs from that of the right subtree of u by at most 1.
- If u violates the above requirement, we say that u is imbalanced.
- The balance factor of a node is the height of its right subtree minus the height of its left subtree.
- A node with balance factor 1, 0 or –1 is considered balanced.

Balanced Binary Search Tree

- A binary tree T is balanced if the following holds on every internal node u of T :
  - The height of the left subtree of u differs from that of the right subtree of u by at most 1.
- If u violates the above requirement, we say that u is imbalanced.
- The balance factor of a node is the height of its right subtree minus the height of its left subtree.
- A node with balance factor 1, 0 or –1 is considered balanced.



AVL Tree [Adelson-Velskii & Landis, 1962]

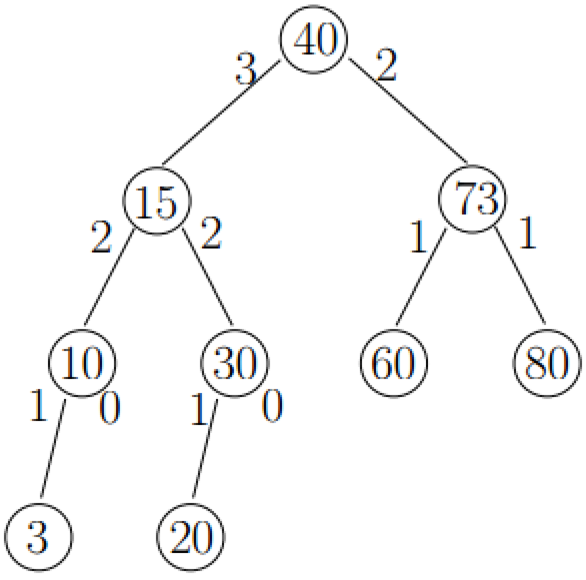
An **AVL**-tree on a set S of n integers is a balanced binary search tree T where the following holds on every internal node u

- u stores the heights of its left and right subtrees.

AVL Tree [Adelson-Velskii & Landis, 1962]

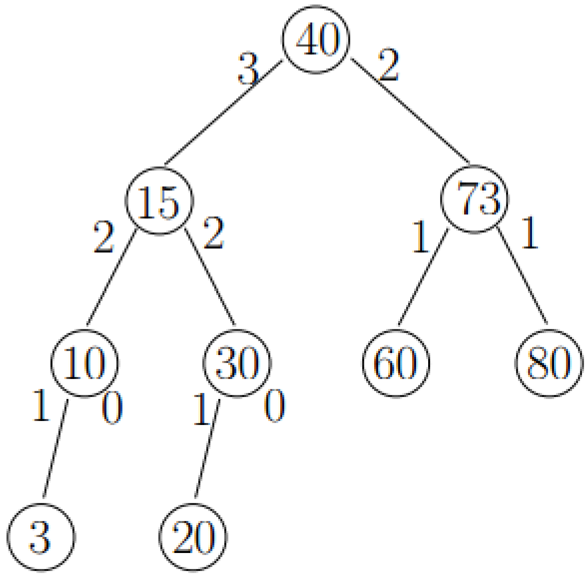
An **AVL**-tree on a set S of n integers is a balanced binary search tree T where the following holds on every internal node u

- u stores the heights of its left and right subtrees.



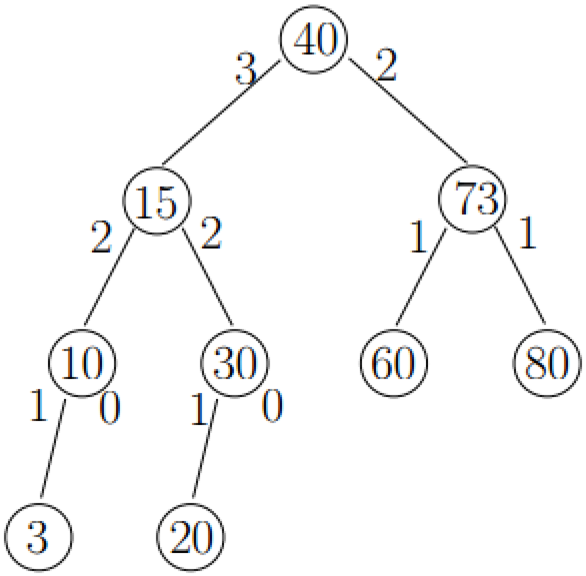
AVL Tree［Adelson-Velskii & Landis, 1962］

- By storing the subtree heights, an internal node knows whether it has become imbalanced.



AVL Tree [Adelson-Velskii & Landis, 1962]

- By storing the subtree heights, an internal node knows whether it has become imbalanced.
- The left subtree height of an internal node can be obtained in O(1) time from its left child. Similarly for the right.



Outline

- Sorting in Linear Time
  - Counting Sort
- Randomized Selection Problem
  - Problem Definition
  - First solution: Selection by sorting
  - A divide-and-conquer algorithm
- Optimal Binary Search Tree Problem
  - Review of Binary Search Tree
  - Problem Definition
  - A Dynamic Programming Algorithm

Optimal BST Problem

Definition

- Given a sequence K = <k_1, k_2,..., k_n> of n distinct keys in sorted order ($k_1<k_2<\ldots<k_n$);

Vocabulary	
Word	Probability
d_0	q_0=0.1
Algorithm	p_1=0.4
d_1	q_1=0.05
Crowd	p_2=0.2
d_2	q_2=0.02
Search	p_3=0.2
d_3	q_3=0.03

Optimal BST Problem

Definition

- Given a sequence K = <$k_1, k_2, \ldots, k_n$> of n distinct keys in sorted order ($k_1 < k_2 < \ldots < k_n$);
  - For each key k_i , the search probability is p_i

Vocabulary

Word	Probability
d_0	q_0=0.1
Algorithm	p_1=0.4
d_1	q_1=0.05
Crowd	p_2=0.2
d_2	q_2=0.02
Search	p_3=0.2
d_3	q_3=0.03

Optimal BST Problem

Definition

- Given a sequence K = <$k_1, k_2, \ldots, k_n$> of n distinct keys in sorted order ($k_1<k_2<\ldots<k_n$);
  - For each key k_i , the search probability is p_i
  - Some values not in K:
    - n+1 “dummy keys” $d_0, d_1, \ldots d_n$;

Vocabulary	
Word	Probability
d_0	q_0=0.1
Algorithm	p_1=0.4
d_1	q_1=0.05
Crowd	p_2=0.2
d_2	q_2=0.02
Search	p_3=0.2
d_3	q_3=0.03

Optimal BST Problem

Definition

- Given a sequence K = <$k_1, k_2,\ldots, k_n$> of n distinct keys in sorted order ($k_1<k_2<\ldots<k_n$);
  - For each key k_i , the search probability is p_i
  - Some values not in K:
    - n+1 "dummy keys" $d_0, d_1, \ldots d_n$;
    - d_0 represents all values < k_1;
    - d_n represents all values > k_n ;
    - d_i represents the values between k_i and k_{i+1}
    - Search probability is q_i

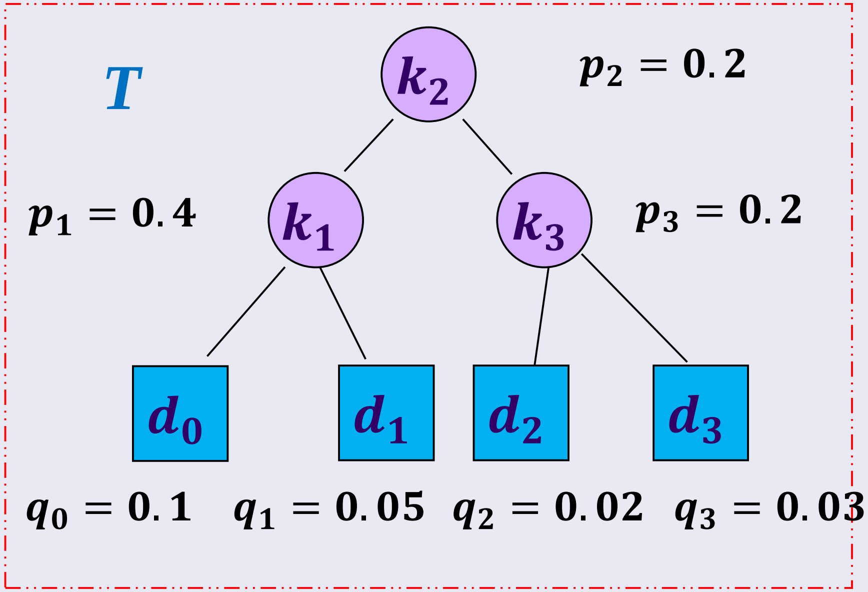
Vocabulary

Word	Probability
d_0	q_0=0.1
Algorithm	p_1=0.4
d_1	q_1=0.05
Crowd	p_2=0.2
d_2	q_2=0.02
Search	p_3=0.2
d_3	q_3=0.03

Optimal BST Problem

Definition

- Construct a BST on these keys and dummy nodes
  - Each key k_i is an internal node;
  - Each “dummy key” d_i is a leaf



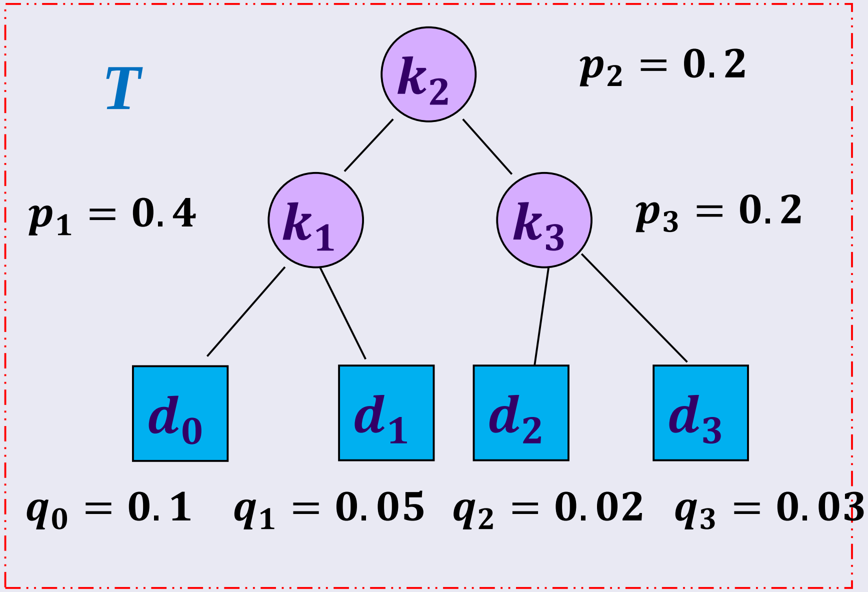
Vocabulary	
Word	Probability
d_0	q_0=0.1
Algorithm	p_1=0.4
d_1	q_1=0.05
Crowd	p_2=0.2
d_2	q_2=0.02
Search	p_3=0.2
d_3	q_3=0.03

Optimal BST Problem

Definition

- Search on this BST, every search is either successful or unsuccessful, we have

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

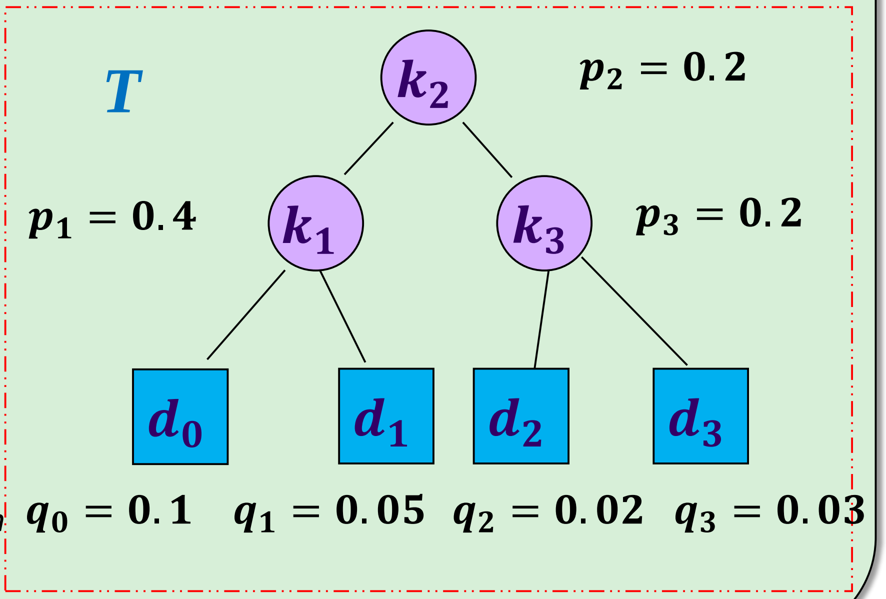


Vocabulary	
Word	Probability
d_0	q_0=0.1
Algorithm	p_1=0.4
d_1	q_1=0.05
Crowd	p_2=0.2
d_2	q_2=0.02
Search	p_3=0.2
d_3	q_3=0.03

Optimal BST Problem

Goal

Construct a BST whose expected search cost is the ***smallest***.



.. Node : distinct keys ($k_1<k_2<\ldots<k_n$)

d_i Leaf: n+1 “dummy keys” d_0, d_1, … d_n

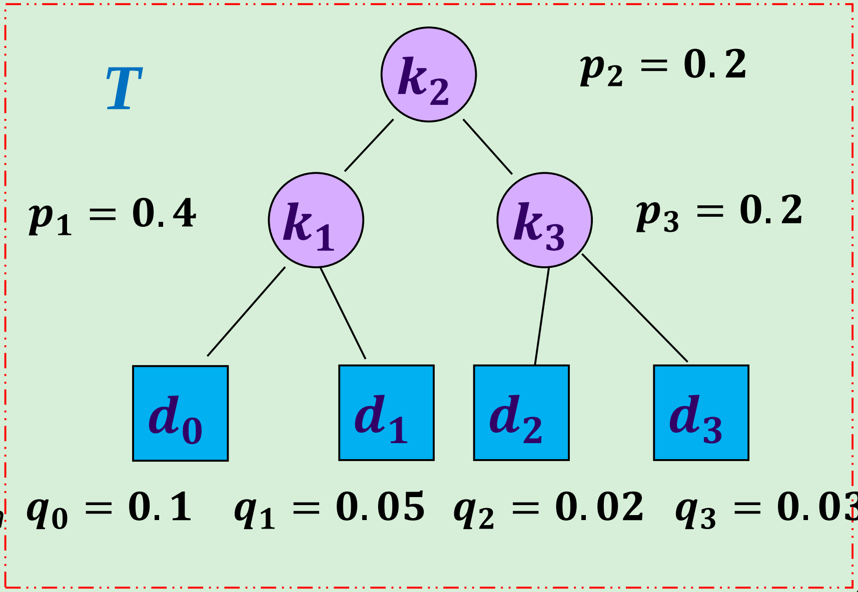
Optimal BST Problem

Goal

*Construct a BST whose expected search cost is the **smallest**.*

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

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



.. Node : distinct keys ($k_1 < k_2 < \ldots < k_n$)

d_i Leaf: n+1 "dummy keys" $d_0, d_1, \ldots d_n$

Optimal BST Problem

Goal

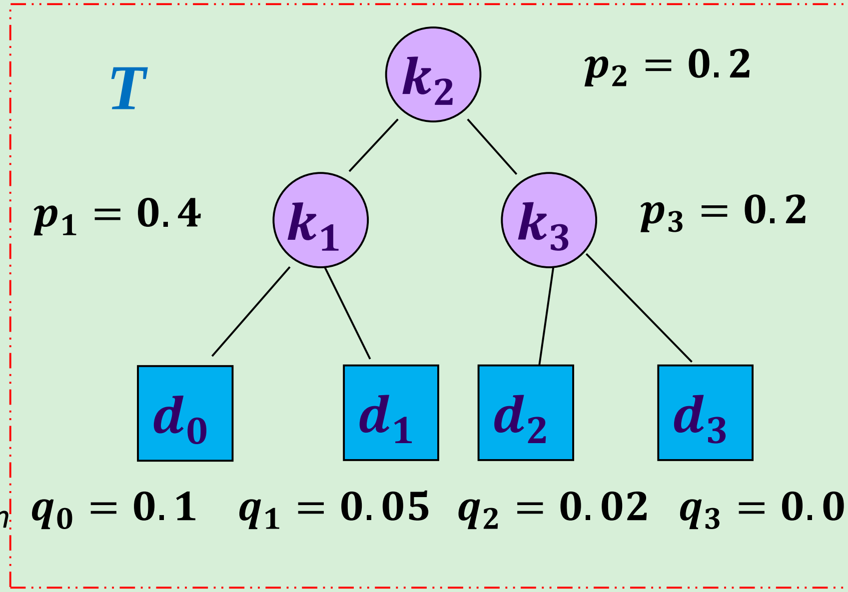
Construct a BST whose expected search cost is the ***smallest***.

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

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

$E.g.$

$$E(T) = 1 * 0.2 + 2 * 0.4 + 2 * 0.2 + 3 * 0.1 + 3 * 0.05 + 3 * 0.02 + 3 * 0.03$$
$$= 2$$



.. Node : distinct keys ($k_1 < k_2 < \ldots < k_n$)

d_i Leaf: n+1 “dummy keys” $d_0, d_1, \ldots d_n$

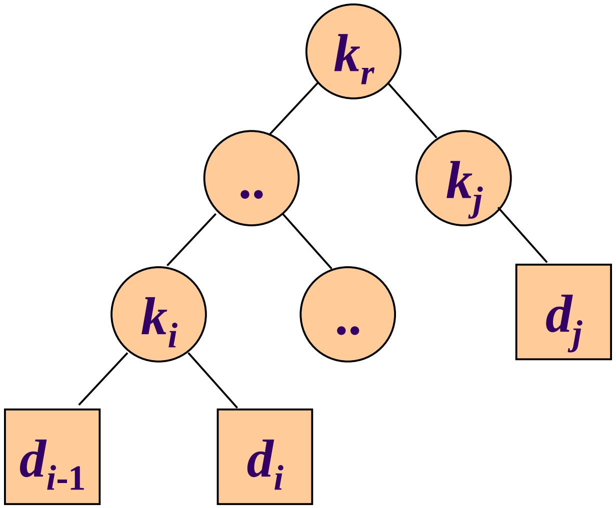
Outline

- Sorting in Linear Time
  - Counting Sort
- Randomized Selection Problem
  - Problem Definition
  - First solution: Selection by sorting
  - A divide-and-conquer algorithm
- Optimal Binary Search Tree Problem
  - Review of Binary Search Tree
  - Problem Definition
  - A Dynamic Programming Algorithm

A DP Algorithm for Optimal BST Problem

Step 1: Space of subproblems

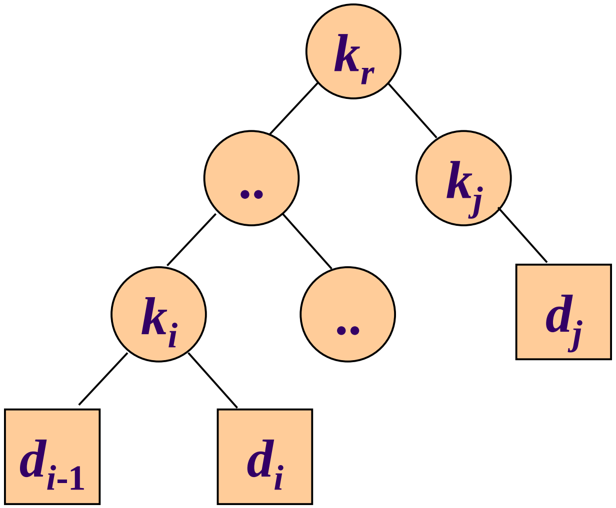
- Subproblem: finding an optimal BST containing the keys k_i , ..., k_j , where $i \geq 1, j \leq n$, and $j \geq i\text{-}1$. (when $j = i\text{-}1$, there are no actual keys, we have just the dummy key d_{i-1} .)



A DP Algorithm for Optimal BST Problem

Step 1: Space of subproblems

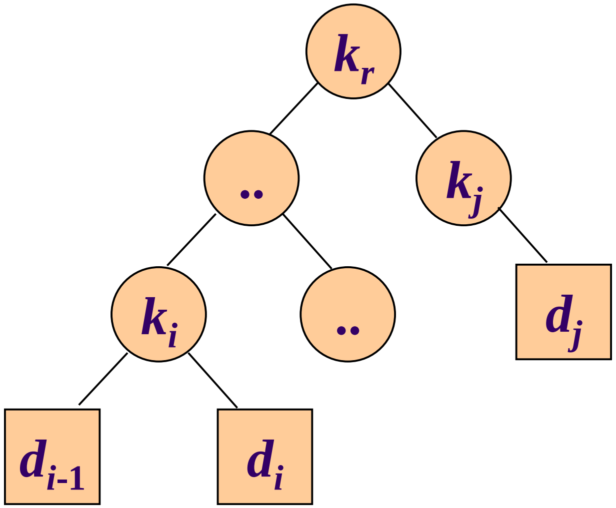
- Subproblem: finding an optimal BST containing the keys k_i , ..., k_j , where $i \geq 1, j \leq n$, and $j \geq i\text{-}1$. (when $j = i\text{-}1$, there are no actual keys, we have just the dummy key d_{i-1} .)
- *e[i,j]:* the expected cost of searching an optimal BST containing the keys k_i, ..., k_j .



A DP Algorithm for Optimal BST Problem

Step 1: Space of subproblems

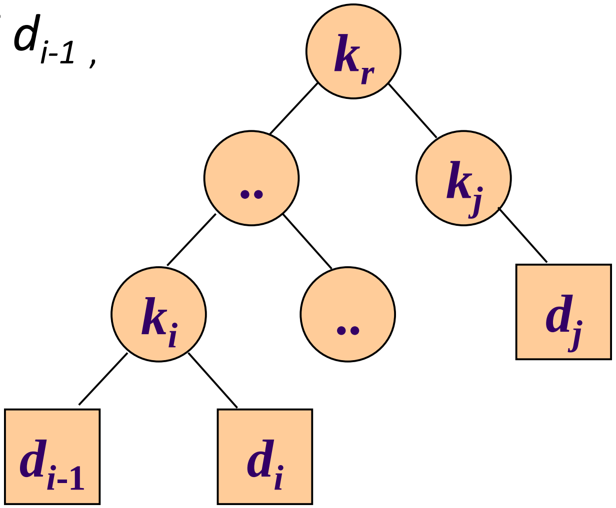
- Subproblem: finding an optimal BST containing the keys k_i , ..., k_j , where $i \geq 1, j \leq n$, and $j \geq i-1$. (when $j = i-1$, there are no actual keys, we have just the dummy key d_{i-1} .)
- *e[i,j]:* the expected cost of searching an optimal BST containing the keys k_i, ..., k_j .
- Ultimately, wish to compute *e[1, n]* .



A DP Algorithm for Optimal BST Problem

Step 1: Space of subproblems

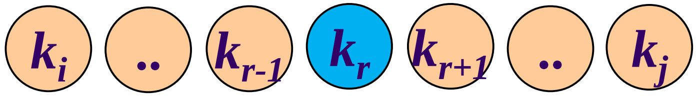
- Subproblem: finding an optimal BST containing the keys k_i , ..., k_j , where $i \geq 1, j \leq n$, and $j \geq i-1$. (when $j = i-1$, there are no actual keys, we have just the dummy key d_{i-1} .)
- $e[i,j]$: the expected cost of searching an optimal BST containing the keys k_i, ..., k_j .
- Ultimately, wish to compute $e[1, n]$.
- Boundary cases:
  - When $j=i-1$, the tree has only one leaf d_{i-1} ,
  - $e[i, i-1] = q_{i-1}$.

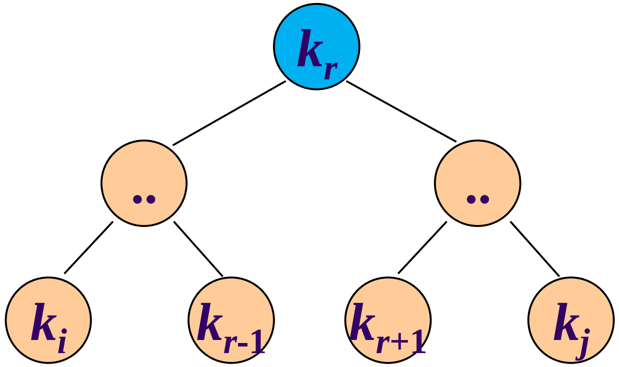


A DP Algorithm for Optimal BST Problem

Step 2: Relating the value of a problem and those of its subproblems

- When $j \geq i$, select a root k_r from among k_i , ..., k_j ,
  - make an optimal BST with keys k_i , ..., k_{r-1} its left subtree
  - and an optimal BST with keys k_{r+1} , ..., k_j its right subtree.





A DP Algorithm for Optimal BST Problem

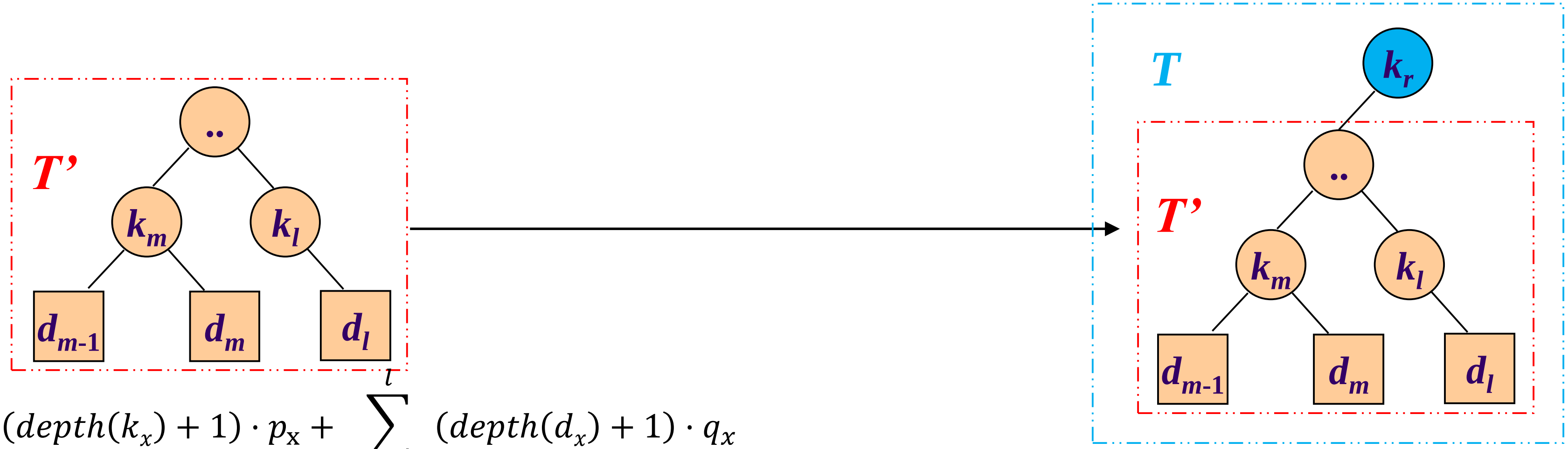
Step 2: Relating the value of a problem and those of its subproblems

- What happens to the expected search cost of a subtree when it becomes a subtree of a node?

A DP Algorithm for Optimal BST Problem

Step 2: Relating the value of a problem and those of its subproblems

- What happens to the expected search cost of a subtree when it becomes a subtree of a node?

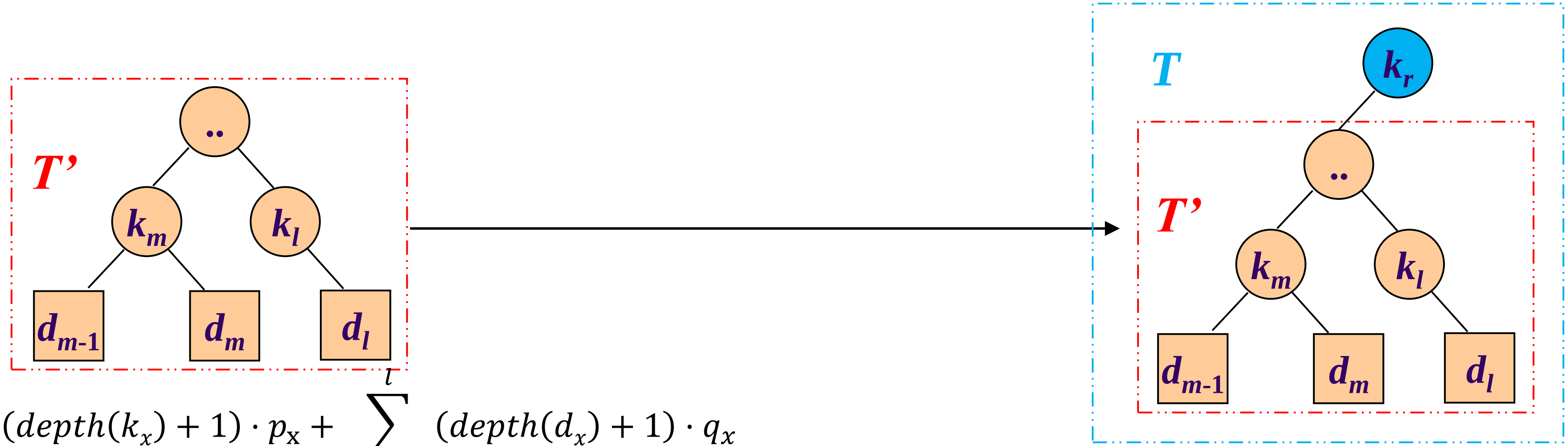


$$E_{T'} = \sum_{x=m}^{l} (depth(k_x) + 1) \cdot p_x + \sum_{x=m-1}^{l} (depth(d_x) + 1) \cdot q_x$$

A DP Algorithm for Optimal BST Problem

Step 2: Relating the value of a problem and those of its subproblems

- What happens to the expected search cost of a subtree when it becomes a subtree of a node?



$$E_{T'} = \sum_{x=m}^{l} (depth(k_x)+1) \cdot p_x + \sum_{x=m-1}^{l} (depth(d_x)+1) \cdot q_x$$

$$E_T = \sum_{x=m}^{l} (depth(k_x)+1+1) \cdot p_x + \sum_{x=m-1}^{l} (depth(d_x)+1+1) \cdot q_x$$

A DP Algorithm for Optimal BST Problem

Step 2: Relating the value of a problem and those of its subproblems

- What happens to the expected search cost of a subtree when it becomes a subtree of a node?

$E_{T'} = e[m, l]$

$$E_T = \sum_{x=m}^{l} (depth(k_x) + 1 + 1) \cdot p_x + \sum_{x=m-1}^{l} (depth(d_x) + 1 + 1) \cdot q_x$$

A DP Algorithm for Optimal BST Problem

Step 2: Relating the value of a problem and those of its subproblems

- What happens to the expected search cost of a subtree when it becomes a subtree of a node?

$E_{T'} = e[m, l]$

$$\begin{aligned} E_T &= \sum_{x=m}^{l} (depth(k_x) + 1 + 1) \cdot p_x + \sum_{x=m-1}^{l} (depth(d_x) + 1 + 1) \cdot q_x \\ &= \sum_{x=m}^{l} (depth(k_x) + 1) \cdot p_x + \sum_{x=m-1}^{l} (depth(d_x) + 1) \cdot q_x + \sum_{x=m}^{l} p_x + \sum_{x=m-1}^{l} q_x \\ &= e[m, l] + w[m, l] \end{aligned}$$

A DP Algorithm for Optimal BST Problem

Step 2: Relating the value of a problem and those of its subproblems

- What happens to the expected search cost of a subtree when it becomes a subtree of a node?

$$E_{T'} = e[m, l]$$

$$w[m, l] = \sum_{x=m}^{l} p_x + \sum_{x=m-1}^{l} q_x$$

$$E_T = \sum_{x=m}^{l} (depth(k_x) + 1 + 1) \cdot p_x + \sum_{x=m-1}^{l} (depth(d_x) + 1 + 1) \cdot q_x$$

$$= \sum_{x=m}^{l} (depth(k_x) + 1) \cdot p_x + \sum_{x=m-1}^{l} (depth(d_x) + 1) \cdot q_x + \sum_{x=m}^{l} p_x + \sum_{x=m-1}^{l} q_x$$

$$= e[m, l] + w[m, l]$$

A DP Algorithm for Optimal BST Problem

Step 2: Relating the value of a problem and those of its subproblems

- What happens to the expected search cost of a subtree when it becomes a subtree of a node?

$$E_{T'} = e[m,l]$$

$$w[m,l] = \sum_{x=m}^{l} p_x + \sum_{x=m-1}^{l} q_x$$
$$= w[m,l-1] + p_l + q_l$$

$$E_T = \sum_{x=m}^{l} (depth(k_x) + 1 + 1) \cdot p_x + \sum_{x=m-1}^{l} (depth(d_x) + 1 + 1) \cdot q_x$$
$$= \sum_{x=m}^{l} (depth(k_x) + 1) \cdot p_x + \sum_{x=m-1}^{l} (depth(d_x) + 1) \cdot q_x + \sum_{x=m}^{l} p_x + \sum_{x=m-1}^{l} q_x$$
$$= e[m,l] + w[m,l]$$

A DP Algorithm for Optimal BST Problem

Step 2: Relating the value of a problem and those of its subproblems

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

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

A DP Algorithm for Optimal BST Problem

Step 2: Relating the value of a problem and those of its subproblems

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

$$e[i, j] = p_r + (e[i, r\text{-}1] + w[i, r\text{-}1]) + (e[r+1, j] + w[r+1, j])$$

- Noting that $w[i, j] = w[i, r\text{-}1] + p_r + w[r+1, j]$

$$(w[i, r-1] = \sum_{x=i}^{r-1} p_x + \sum_{x=i-1}^{r-1} q_x \; , w[r+1, j] = \sum_{x=r+1}^{j} p_x + \sum_{x=r}^{j} q_x \;)$$

A DP Algorithm for Optimal BST Problem

Step 2: Relating the value of a problem and those of its subproblems

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

$$e[i\,,j] = p_r + (e[i\,, r\text{-}1] + w[i, r\text{-}1]) + (e[r+1, j] + w[r+1\,, j])$$

- Noting that $w[i, j] = w[i, r\text{-}1] + p_r + w[r+1, j]$

$$(w[i, r-1] = \sum_{x=i}^{r-1} p_x + \sum_{x=i-1}^{r-1} q_x \;, w[r+1, j] = \sum_{x=r+1}^{j} p_x + \sum_{x=r}^{j} q_x \;)$$

- We rewrite e[i, j] as

$$e[i, j\,] = e[i, r\text{-}1] + e[r+1, j] + w[i, j]$$

A DP Algorithm for Optimal BST Problem

Step 2: Relating the value of a problem and those of its subproblems

- Choose k_r as the root that gives the lowest expected search cost, giving us our final recursive formulation:

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

- *e[i, j]* give the expected search costs in optimal BST.

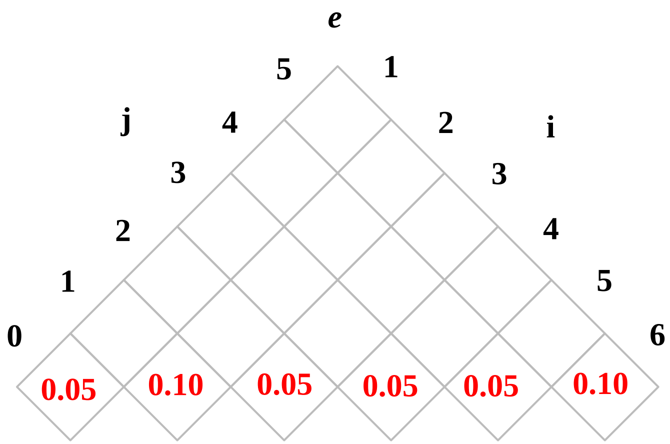
An Example of Optimal Binary Search Tree

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

An Example of Optimal Binary Search Tree

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

Step 0: Initialization

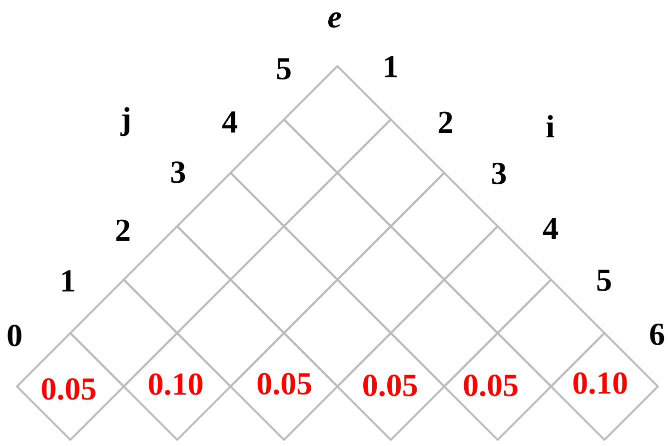


$e[i, i\text{-}1] = q_{i-1}$

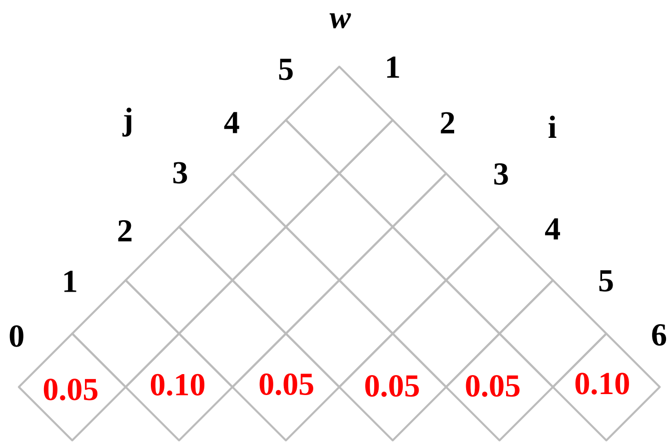
An Example of Optimal Binary Search Tree

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

Step 0: Initialization



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



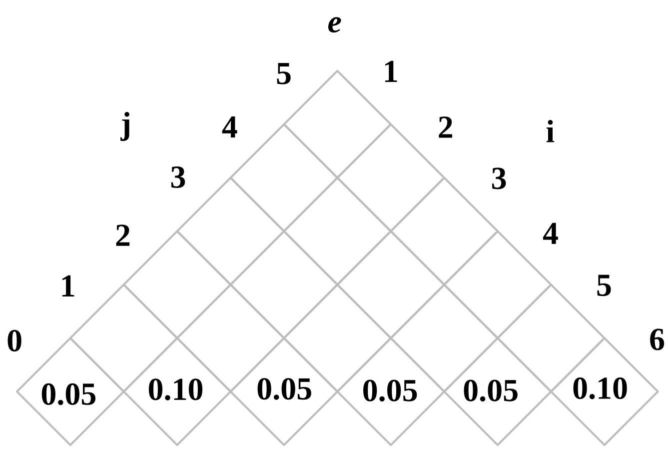
$w[i, i\text{-}1] = q_{i\text{-}1}$

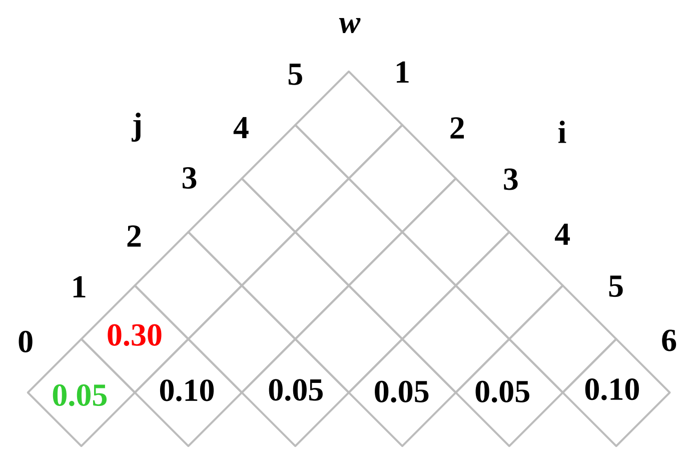
An Example of Optimal Binary Search Tree

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

Step 1: Computing *e[i, i], w[i, i]*

$$
\begin{aligned}
w[1,1] &= w[i, j-1] + p_j + q_j \\
&= w[1,0] + p_1 + q_1 \\
&= 0.05 + 0.15 + 0.10 = 0.30
\end{aligned}
$$





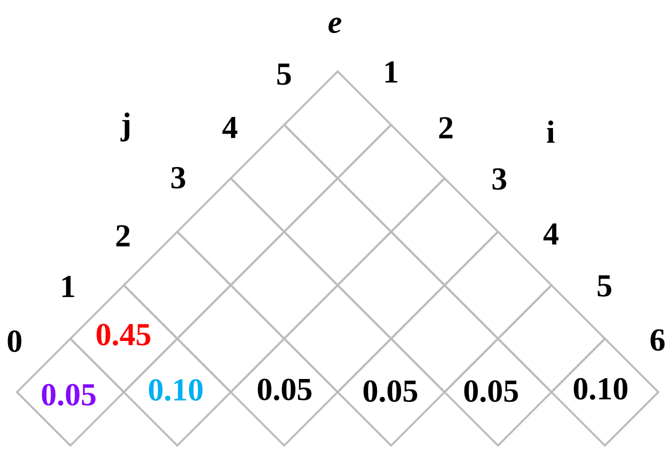
An Example of Optimal Binary Search Tree

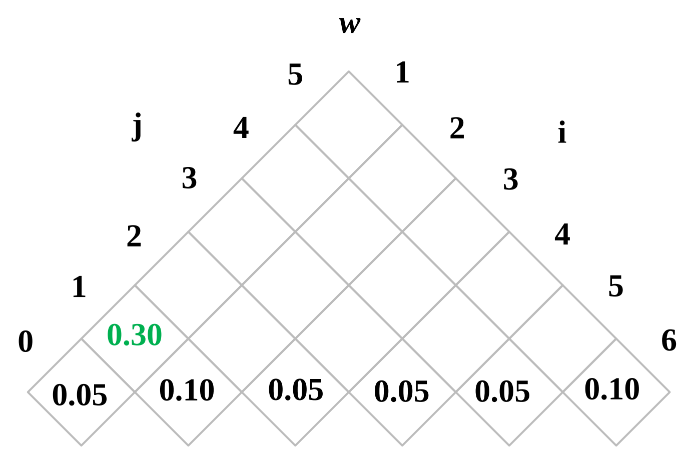
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Step 1: Computing *e[i, i], w[i, i]*

$$\begin{aligned} e[1,1] &= \min_{1\le r\le 1}(e[i,r-1]+e[r+1,j]+w[i,j]) \\ &= e[1,0]+e[2,1]+w[1,1] \\ &= 0.05+0.10+0.30=0.45 \end{aligned}$$

$$\begin{aligned} w[1,1] &= w[i,j-1]+p_j+q_j \\ &= w[1,0]+p_1+q_1 \\ &= 0.05+0.15+0.10=0.30 \end{aligned}$$





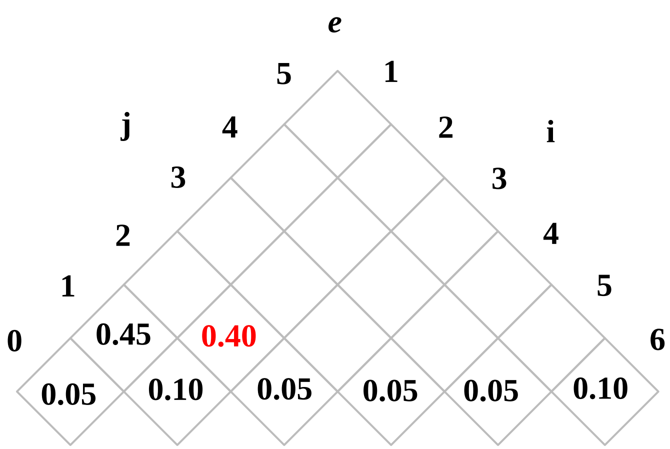
An Example of Optimal Binary Search Tree

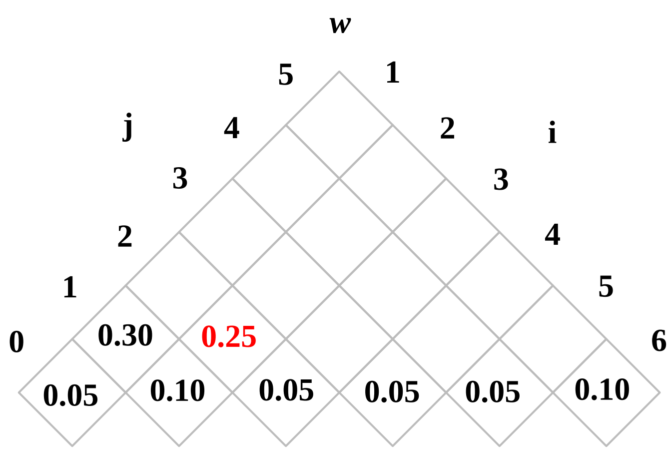
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Step 1: Computing *e[i, i], w[i, i]*

$$\begin{aligned} e[2,2] &= \min_{2\le r\le 2}(e[i,r-1]+e[r+1,j]+w[i,j]) \\ &= e[2,1]+e[3,2]+w[2,2] \\ &= 0.10+0.05+0.25 = 0.40 \end{aligned}$$

$$\begin{aligned} w[2,2] &= w[i,j-1]+p_j+q_j \\ &= w[2,1]+p_2+q_2 \\ &= 0.10+0.10+0.05 = 0.25 \end{aligned}$$





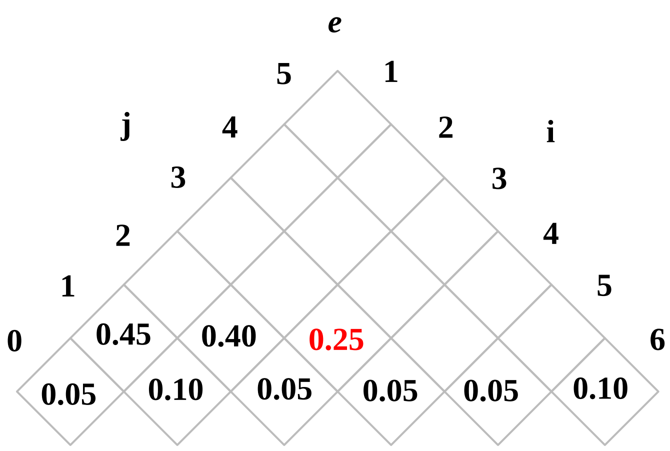
An Example of Optimal Binary Search Tree

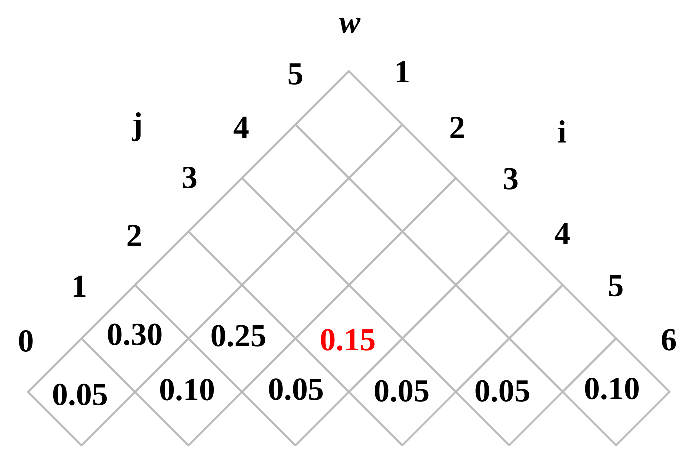
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Step 1: Computing *e[i, i], w[i, i]*

$$\begin{aligned} e[3,3] &= \min_{3\le r\le 3}(e[i,r-1]+e[r+1,j]+w[i,j]) \\ &= e[3,2]+e[4,3]+w[3,3] \\ &= 0.05+0.05+0.15 = 0.25 \end{aligned}$$

$$\begin{aligned} w[3,3] &= w[i,j-1]+p_j+q_j \\ &= w[3,2]+p_3+q_3 \\ &= 0.05+0.05+0.05 = 0.15 \end{aligned}$$





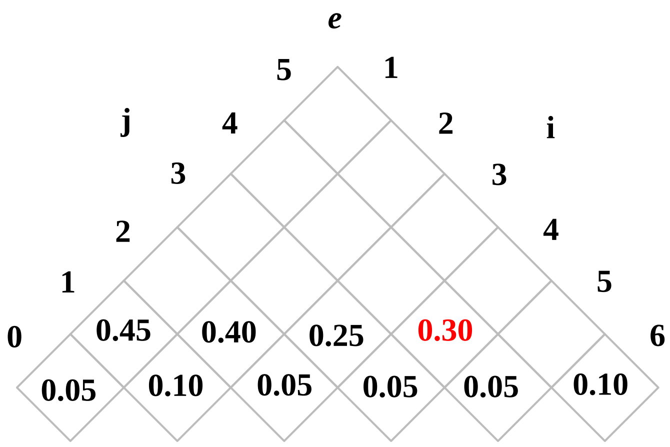
An Example of Optimal Binary Search Tree

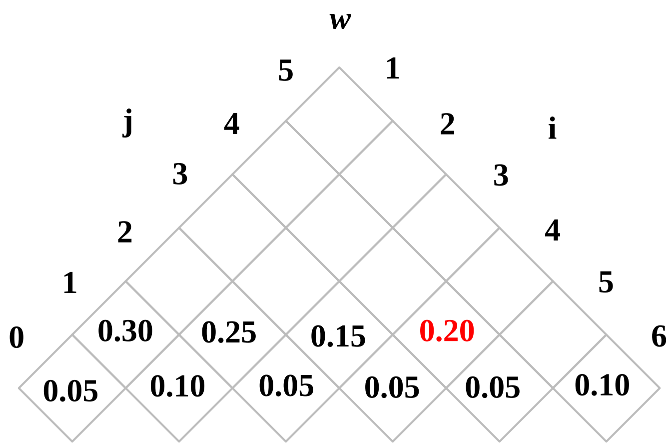
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Step 1: Computing *e[i, i], w[i, i]*

$$\begin{aligned} e[4,4] &= \min_{4 \le r \le 4} (e[i, r-1] + e[r+1, j] + w[i,j]) \\ &= e[4,3] + e[5,4] + w[4,4] \\ &= 0.05 + 0.05 + 0.20 = 0.30 \end{aligned}$$

$$\begin{aligned} w[4,4] &= w[i, j-1] + p_j + q_j \\ &= w[4,3] + p_4 + q_4 \\ &= 0.05 + 0.10 + 0.05 = 0.20 \end{aligned}$$





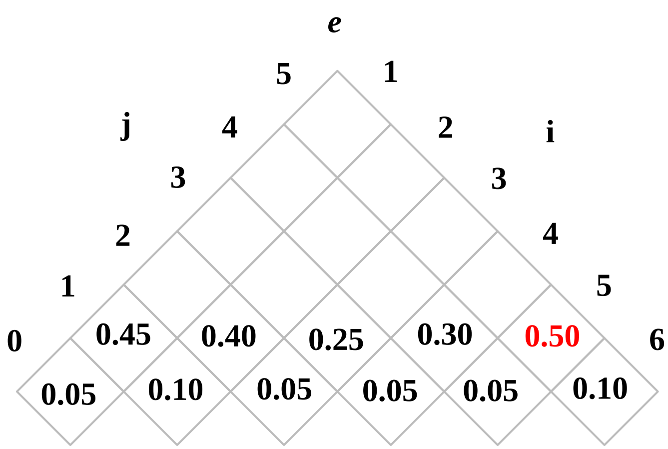
An Example of Optimal Binary Search Tree

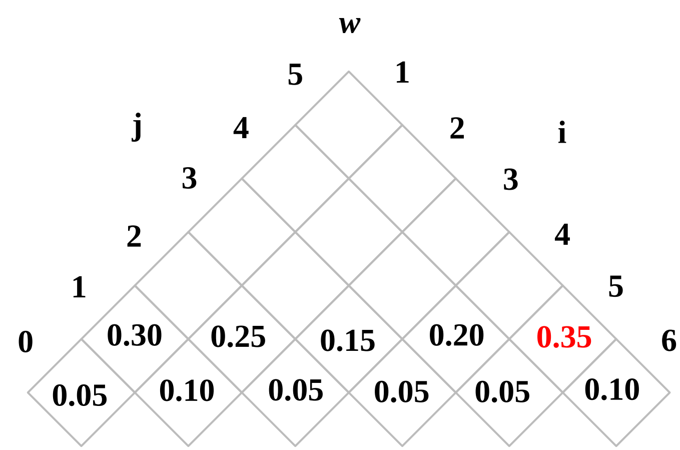
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Step 1: Computing *e[i, i], w[i, i]*

$$
\begin{aligned}
e[5,5] &= \min_{5\le r\le 5}(e[i,r-1]+e[r+1,j]+w[i,j]) \\
&= e[5,4]+e[6,5]+w[5,5] \\
&= 0.05+0.10+0.35 = 0.50
\end{aligned}
$$

$$
\begin{aligned}
w[5,5] &= w[i,j-1]+p_j+q_j \\
&= w[5,4]+p_5+q_5 \\
&= 0.05+0.20+0.10 = 0.35
\end{aligned}
$$





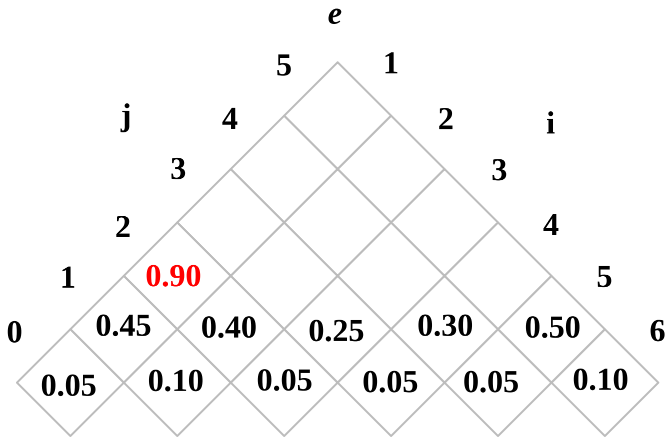
An Example of Optimal Binary Search Tree

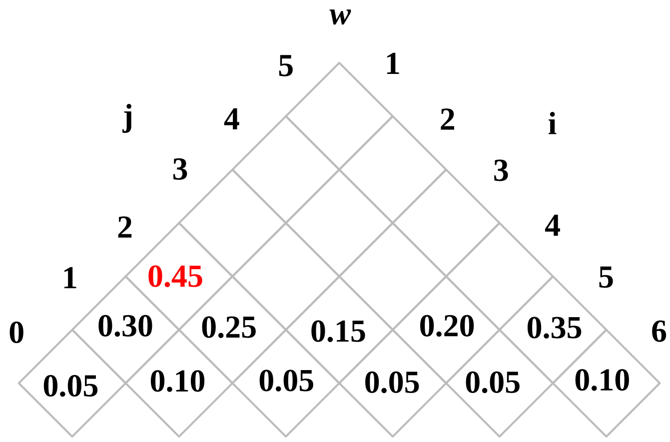
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Step 2: Computing *e[i, i+1], w[i, i+1]*

$$e[1,2] = \min_{1\le r\le 2}(e[i,r-1]+e[r+1,j]+w[i,j])$$
$$= min\begin{cases} e[1,0]+e[2,2]+w[1,2] \\ e[1,1]+e[3,2]+w[1,2] \end{cases}$$
$$= min\begin{cases} 0.05+0.40+0.45=0.90 \\ 0.45+0.05+0.45=0.95 \end{cases}$$

$$w[1,2] = w[i,j-1]+p_j+q_j$$
$$= w[1,1]+p_2+q_2$$
$$= 0.30+0.10+0.05 = 0.45$$





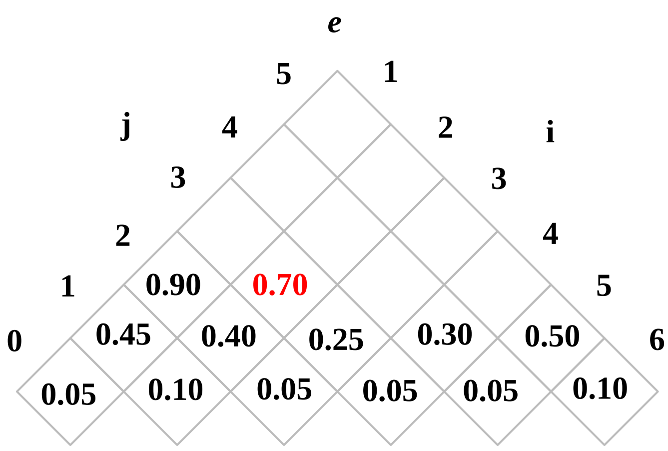
An Example of Optimal Binary Search Tree

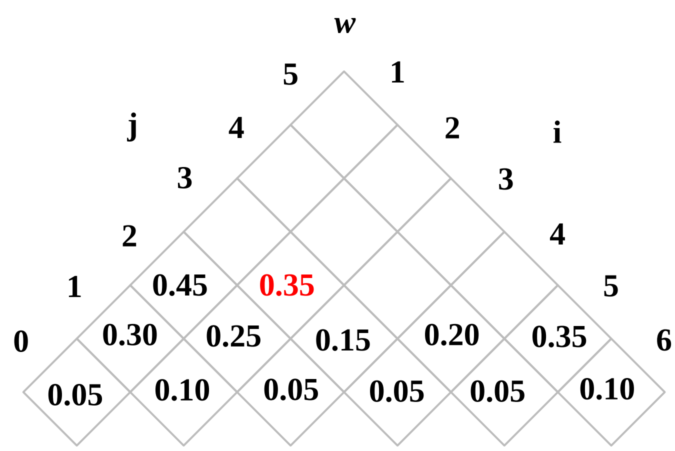
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Step 2: Computing *e[i, i+1], w[i, i+1]*

$$e[2,3] = \min_{2\le r\le 3}(e[i,r-1] + e[r+1,j] + w[i,j])$$
$$= min\begin{cases} e[2,1] + e[3,3] + w[2,3] \\ e[2,2] + e[4,3] + w[2,3] \end{cases}$$
$$= min\begin{cases} 0.10 + 0.25 + 0.35 = 0.70 \\ 0.40 + 0.05 + 0.35 = 0.80 \end{cases}$$

$$w[2,3] = w[i,j-1] + p_j + q_j$$
$$= w[2,2] + p_3 + q_3$$
$$= 0.25 + 0.05 + 0.05 = 0.35$$





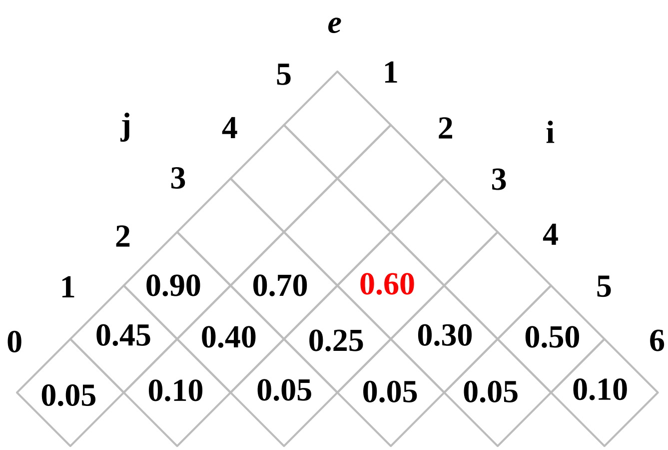
An Example of Optimal Binary Search Tree

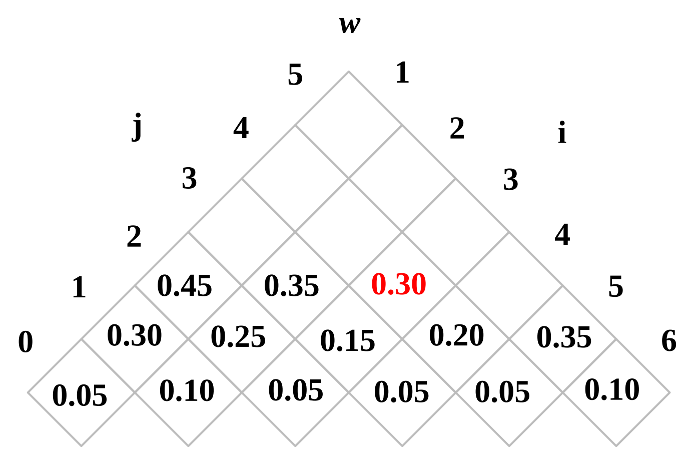
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Step 2: Computing *e[i, i+1], w[i, i+1]*

$$e[3,4] = \min_{3 \le r \le 4} (e[i, r-1] + e[r+1, j] + w[i,j])$$
$$= min \begin{cases} e[3,2] + e[4,4] + w[3,4] \\ e[3,3] + e[5,4] + w[3,4] \end{cases}$$
$$= min \begin{cases} 0.05 + 0.30 + 0.30 = 0.65 \\ 0.25 + 0.05 + 0.30 = 0.60 \end{cases}$$

$$w[3,4] = w[i, j-1] + p_j + q_j$$
$$= w[3,3] + p_4 + q_4$$
$$= 0.15 + 0.10 + 0.05 = 0.30$$





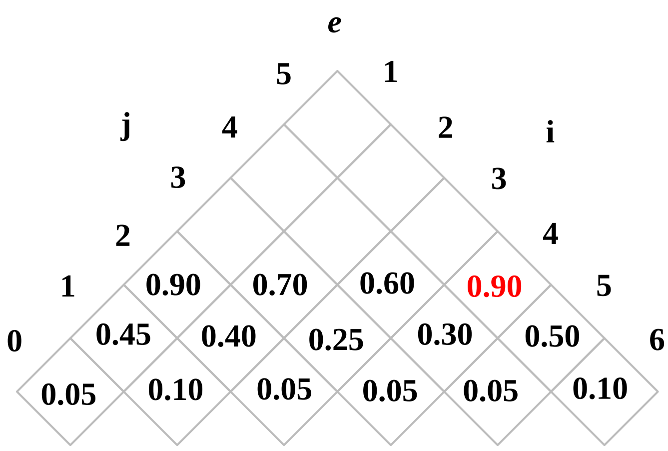
An Example of Optimal Binary Search Tree

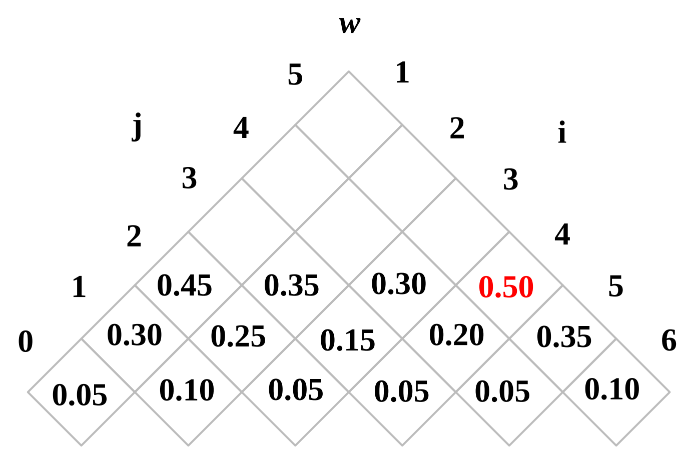
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Step 2: Computing *e[i, i+1], w[i, i+1]*

$$e[4,5] = \min_{4\le r\le 5}(e[i,r-1] + e[r+1,j] + w[i,j])$$
$$= min\begin{cases} e[4,3] + e[5,5] + w[4,5] \\ e[4,4] + e[6,5] + w[4,5] \end{cases}$$
$$= min\begin{cases} 0.30 + 0.50 + 0.50 = 1.30 \\ 0.30 + 0.10 + 0.50 = 0.90 \end{cases}$$

$$w[4,5] = w[i,j-1] + p_j + q_j$$
$$= w[4,4] + p_5 + q_5$$
$$= 0.20 + 0.20 + 0.10 = 0.50$$





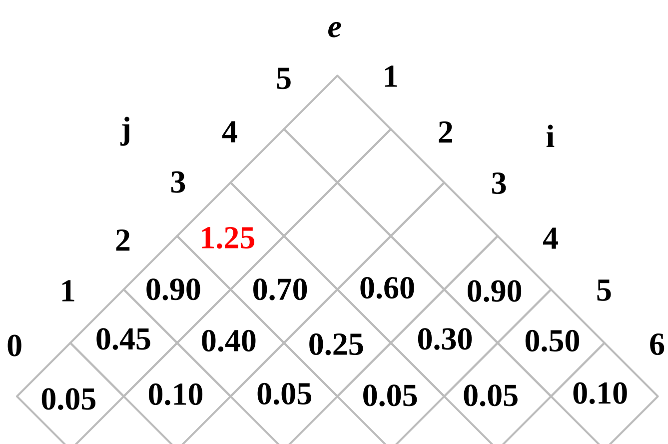
An Example of Optimal Binary Search Tree

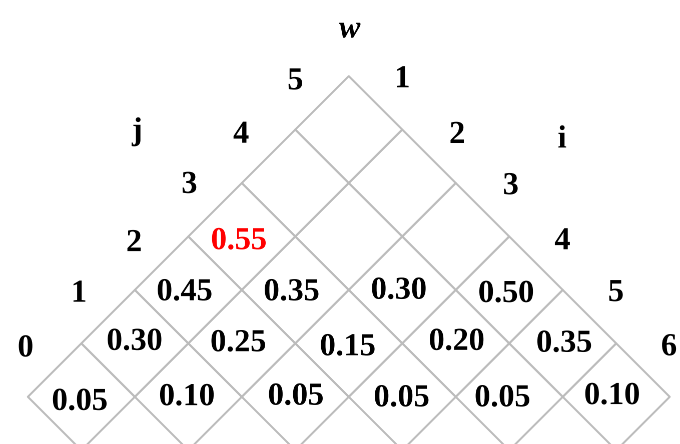
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Step 3: Computing *e[i, i+2], w[i, i+2]*

$$e[1,3] = \min_{1\le r\le 3}(e[i,r-1]+e[r+1,j]+w[i,j])$$
$$= min\begin{cases} e[1,0]+e[2,3]+w[1,3] \\ e[1,1]+e[3,3]+w[1,3] \\ e[1,2]+e[4,3]+w[1,3] \end{cases}$$
$$= min\begin{cases} 0.05+0.70+0.55=1.30 \\ 0.45+0.25+0.55=1.25 \\ 0.90+0.05+0.55=1.55 \end{cases}$$

$$w[1,3] = w[i,j-1]+p_j+q_j$$
$$= w[1,2]+p_3+q_3$$
$$= 0.45+0.05+0.05 = 0.55$$





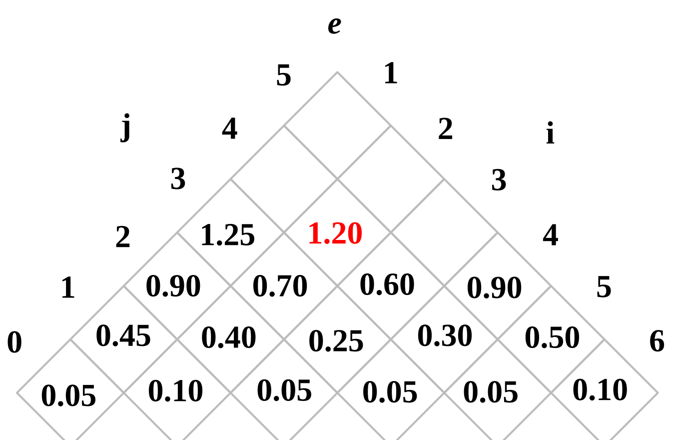
An Example of Optimal Binary Search Tree

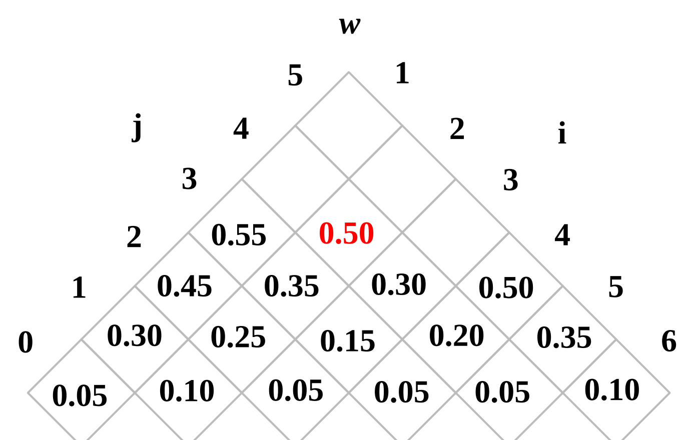
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Step 3: Computing *e[i, i+2], w[i, i+2]*

$$e[2,4] = \min_{2\le r\le 4}(e[i,r-1]+e[r+1,j]+w[i,j])$$
$$= min\begin{cases} e[2,1]+e[3,4]+w[2,4] \\ e[2,2]+e[4,4]+w[2,4] \\ e[2,3]+e[5,4]+w[2,4] \end{cases}$$
$$= min\begin{cases} 0.10+0.60+0.50 = 1.20 \\ 0.40+0.30+0.50 = 1.20 \\ 0.70+0.05+0.50 = 1.25 \end{cases}$$

$$w[2,4] = w[i,j-1]+p_j+q_j$$
$$= w[2,3]+p_4+q_4$$
$$= 0.35+0.10+0.05 = 0.50$$





An Example of Optimal Binary Search Tree

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

Step 3: Computing *e[i, i+2], w[i, i+2]*

$$e[3,5] = \min_{3\le r\le 5}(e[i,r-1]+e[r+1,j]+w[i,j])$$

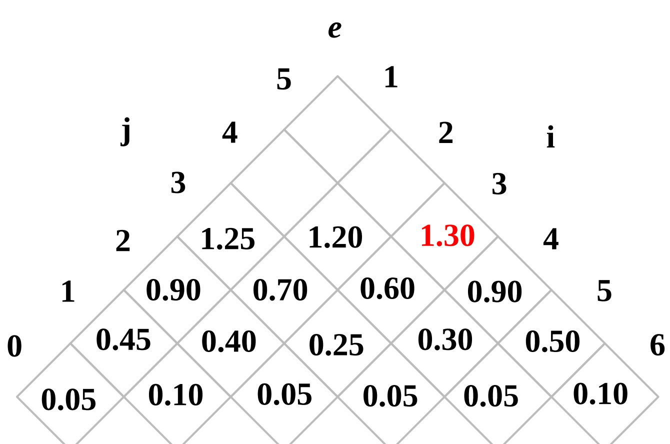
$$= min\begin{cases} e[3,2]+e[4,5]+w[3,5] \\ e[3,3]+e[5,5]+w[3,5] \\ e[3,4]+e[6,5]+w[3,5] \end{cases}$$

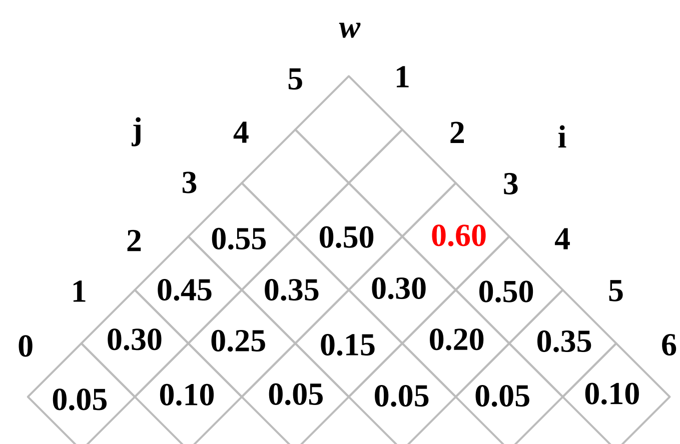
$$= min\begin{cases} 0.05+0.90+0.60=1.55 \\ 0.25+0.50+0.60=1.35 \\ 0.60+0.10+0.60=1.30 \end{cases}$$

$$w[3,5] = w[i,j-1]+p_j+q_j$$

$$= w[3,4]+p_5+q_5$$

$$= 0.30+0.20+0.10 = 0.60$$





An Example of Optimal Binary Search Tree

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

Step 4: Computing *e[i, i+3], w[i, i+3]*

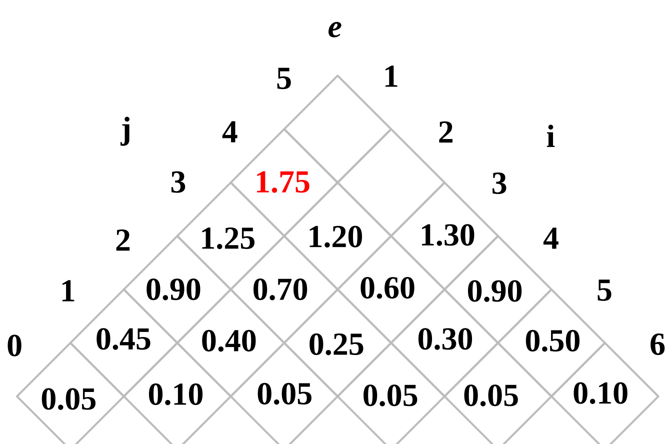
$$e[1,4] = \min_{1\le r\le 4}(e[i,r-1] + e[r+1,j] + w[i,j])$$

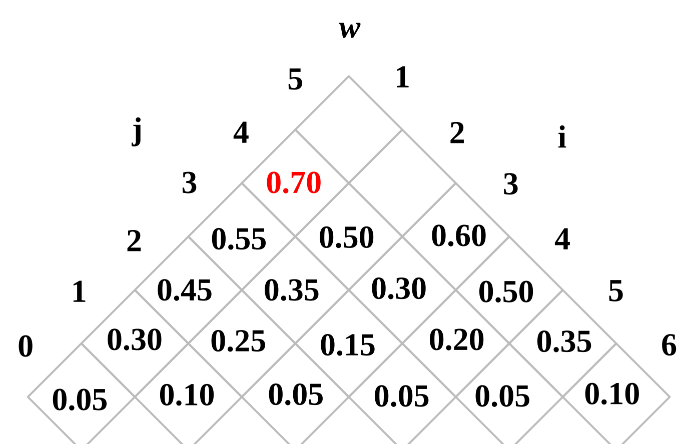
$$= min\begin{cases} e[1,0] + e[2,4] + w[1,4], e[1,1] + e[3,4] + w[1,4] \\ e[1,2] + e[4,4] + w[1,4], e[1,4] + e[5,4] + w[1,4] \end{cases}$$

$$= min\begin{cases} 0.05 + 1.20 + 0.70, 0.45 + 0.60 + 0.70 \\ 0.90 + 0.30 + 0.70, 1.75 + 0.05 + 0.70 \end{cases}$$

$$= 1.75$$

$$w[1,4] = w[i,j-1] + p_j + q_j$$
$$= w[1,3] + p_4 + q_4$$
$$= 0.55 + 0.10 + 0.05 = 0.70$$





An Example of Optimal Binary Search Tree

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

Step 4: Computing *e[i, i+3], w[i, i+3]*

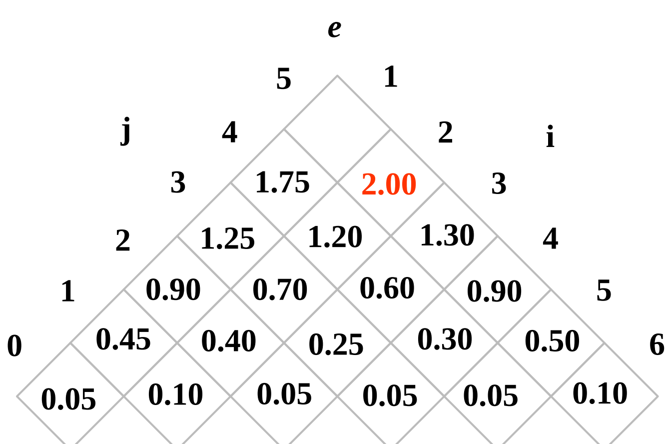
$$e[2,5] = \min_{2\le r\le 5}(e[i,r-1]+e[r+1,j]+w[i,j])$$

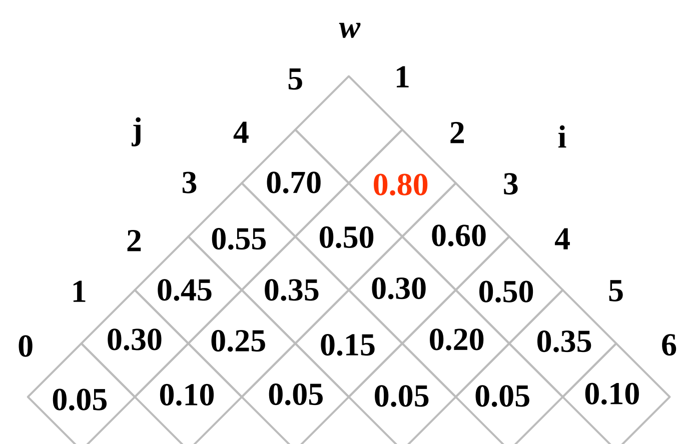
$$= min\begin{cases} e[2,1]+e[3,5]+w[2,5], e[2,2]+e[4,5]+w[2,5] \\ e[2,3]+e[5,5]+w[1,5], e[2,5]+e[6,5]+w[2,5] \end{cases}$$

$$= min\begin{cases} 0.10+1.30+0.80, 0.40+0.90+0.80 \\ 0.70+0.50+0.80, 2.00+0.10+0.80 \end{cases}$$

$$= 2.00$$

$$\begin{aligned} w[2,5] &= w[i,j-1]+p_j+q_j \\ &= w[2,4]+p_5+q_5 \\ &= 0.50+0.20+0.10 = 0.80 \end{aligned}$$





An Example of Optimal Binary Search Tree

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

Step 5: Computing *e[i, i+4], w[i, i+4]*

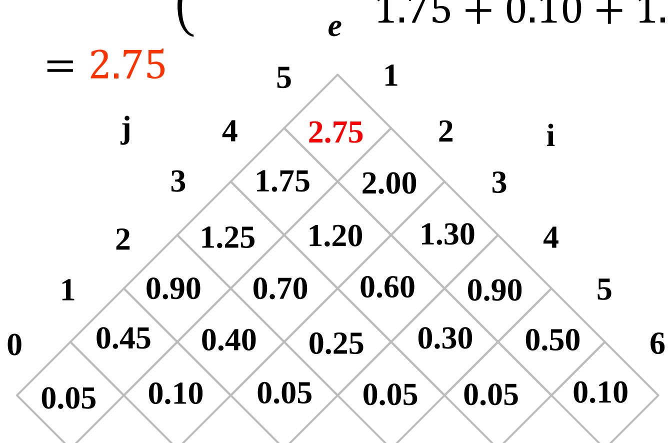
$$e[1,5] = \min_{1\le r\le 5}(e[i,r-1]+e[r+1,j]+w[i,j])$$

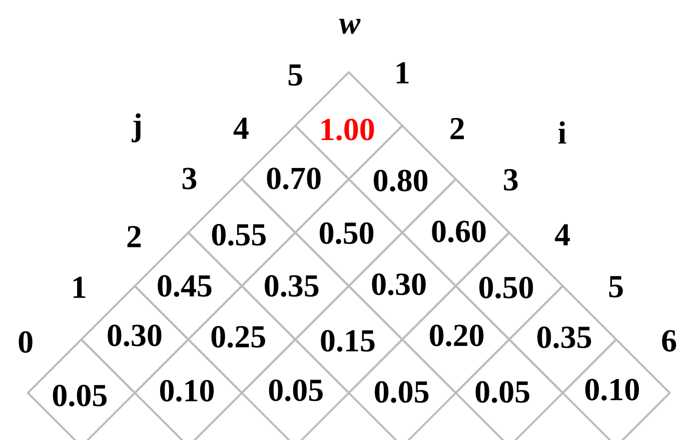
$$= min\begin{cases} e[1,0]+e[2,5]+w[1,5], e[1,1]+e[3,5]+w[1,5] \\ e[1,2]+e[4,5]+w[1,5], e[1,3]+e[5,5]+w[1,5] \\ e[1,4]+e[6,5]+w[1,5] \end{cases}$$

$$= min\begin{cases} 0.05+2.00+1.00, 0.45+1.30+1.00 \\ 0.90+0.90+1.00, 1.25+0.50+1.00 \\ 1.75+0.10+1.00 \end{cases}$$

$$= 2.75$$

$$\begin{aligned} w[1,5] &= w[i,j-1]+p_j+q_j \\ &= w[1,4]+p_5+q_5 \\ &= 0.70+0.20+0.10 = 1.00 \end{aligned}$$





A DP Algorithm for Optimal BST Problem

Algorithm 1: 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;

A DP Algorithm for Optimal BST Problem

Algorithm 1: 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* $n+1$ **do**
  $e[i, i-1] = q_{i-1}$;
  $w[i, i-1] = q_{i-1}$;
end

A DP Algorithm for Optimal BST Problem

Algorithm 1: 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* $n+1$ **do**
    $e[i, i-1] = q_{i-1}$;
    $w[i, i-1] = q_{i-1}$;
end
for l=*1 to* n **do**

A DP Algorithm for Optimal BST Problem

Algorithm 1: 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* $n+1$ **do**
    $e[i, i-1] = q_{i-1}$;
    $w[i, i-1] = q_{i-1}$;
end
for l=*1* *to* n **do**
    for i=*1* *to* $n-l+1$ **do**
        $j = i + l - 1$;
        $e[i, j] = \infty$;
        $w[i, j] = w[i, j-1] + p_j + q_j$;

A DP Algorithm for Optimal BST Problem

Algorithm 1: 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* $n+1$ **do**
    $e[i, i-1] = q_{i-1}$;
    $w[i, i-1] = q_{i-1}$;
end
for l=*1* *to* n **do**
    for i=*1* *to* $n-l+1$ **do**
        $j = i + l - 1$;
        $e[i, j] = \infty$;
        $w[i, j] = w[i, j-1] + p_j + q_j$;
        for r=i *to* j **do**
            $t = e[i, r-1] + e[r+1, j] + w[i, j]$;

A DP Algorithm for Optimal BST Problem

Algorithm 1: 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 do
    e[i, i-1] = q_{i-1};
    w[i, i-1] = q_{i-1};
end
for l=1 to n do
    for i=1 to n-l+1 do
        j = i+l-1;
        e[i, j] = ∞;
        w[i, j] = w[i, j-1] + p_j + q_j;
        for r=i to j do
            t = e[i, r-1] + e[r+1, j] + w[i, j];
            if t < e[i, j] then
                e[i, j] = t;
                root[i, j] = r;
            end
```

A DP Algorithm for Optimal BST Problem

Algorithm 1: 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 do
    e[i, i-1] = q_{i-1};
    w[i, i-1] = q_{i-1};
end
for l=1 to n do
    for i=1 to n-l+1 do
        j = i+l-1;
        e[i, j] = ∞;
        w[i, j] = w[i, j-1] + p_j + q_j;
        for r=i to j do
            t = e[i, r-1] + e[r+1, j] + w[i, j];
            if t < e[i, j] then
                e[i, j] = t;
                root[i, j] = r;
            end
        end
    end
end
```

A DP Algorithm for Optimal BST Problem

Algorithm 1: 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* $n+1$ **do**
    $e[i, i-1] = q_{i-1}$;
    $w[i, i-1] = q_{i-1}$;
end
for l=*1 to* n **do**
    for i=*1 to* $n-l+1$ **do**
        $j = i + l - 1$;
        $e[i, j] = \infty$;
        $w[i, j] = w[i, j-1] + p_j + q_j$;
        for r=i *to* j **do**
            $t = e[i, r-1] + e[r+1, j] + w[i, j]$;
            if $t < e[i, j]$ **then**
                $e[i, j] = t$;
                $root[i, j] = r$;
            end
        end
    end
end
return e and $root$;

A DP Algorithm for Optimal BST Problem

Algorithm 1: 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* $n+1$ **do**
  $e[i, i-1] = q_{i-1}$;
  $w[i, i-1] = q_{i-1}$;
end
for l=*1 to* n **do**
  for i=*1 to* $n-l+1$ **do**
    $j = i + l - 1$;
    $e[i, j] = \infty$;
    $w[i, j] = w[i, j-1] + p_j + q_j$;
    for r=i *to* j **do**
      $t = e[i, r-1] + e[r+1, j] + w[i, j]$;
      if $t < e[i, j]$ **then**
        $e[i, j] = t$;
        $root[i, j] = r$;
      end
    end
  end
end
return e and $root$;

Complexity: The loops are nested three levels deep. Each loop index takes on $\leq n$ values. Hence the time complexity is $O(n^3)$. Space complexity is $\Theta(n^2)$.


thank you
danke
謝謝
ngiyabonga
teşekkür ederim
gracias
merci
obrigado
bedankt
dziękuję
go raibh maith agat
mochchakkeram
sukriya
kop khun krap
terima kasih
grazie
arigatô
dakujem
takk
tapadh leat
спасибо
Баярлалаа
рахмат
faafetai lava
vinaka
спасибі
blagodaram
mersi
kia ora
barka
welalin
tack
spas
dank je
misaotra
matondo
paldies
grazzi
mahalo
хвала
asante
manana
obrigada
tenki
murakoze
chokrane
mamnun
дякую
djiere dieuf
tau
kiitos
dankie
dhanyavad
nanni
nandri
mauruuru
köszönöm
hvala
gràçie
bayarlalaa
enkosi
sobodi
dĕkuji
sagolun
chnorakaloutioun
gratias ago
gràcies
sulpáy
taiku
akun
dankon
ačiū
mèsi
didi madloba
najis tuke
kam sah hamnida
rahmat
ありがとう
tanemirt
rahmet
diolch
dhanyavadagalu
shukriya
mercé
мерси
trugarez
তোমাকে ধন্যবাদ
감사합니다
xièxie
ευχαριστώ