COL106: Data Structures and Algorithms

Ragesh Jaiswal, IIT Delhi

Introduction

- How do Data Structures play a part in making computational tasks efficient?

Introduction

- How do Data Structures play a part in making computational tasks efficient?

Example problem

Maintain a record of students and their scores on some test so that queries of the following nature may be answered:

- Insert: Insert a new record of a student and his/her score.
- Search: Find the score of a given student.

- Suppose we maintain the information in a 2-dimensional array such that the array is sorted based on the names (dictionary order).
  - How much time does each insert operations take? $O(n)$
  - How much time does each search operation take? $O(\log n)$ using **Binary Search**
  - In this case, if the majority of the operations performed are insert operations, then the previous one is better.

Problem

Given a sorted array A containing n integers and an integer x, check if x is present in A.

Algorithm

$\texttt{BinarySearch}(x, A, i, j)$
- if($j < i$)return("not present")
- $mid \leftarrow \lfloor \frac{i+j}{2} \rfloor$
- if($A[mid] = x$)return("present")
- if($x < A[mid]$)return($\texttt{BinarySearch}(x, A, i, mid - 1)$)
- else return($\texttt{BinarySearch}(x, A, mid + 1, j)$)

- What is the running time of the above algorithm in terms of the Big-O notation?
- Let us denote $T(n)$ as the worst case running time for searching in sorted arrays of size n.
- $T(n) \leq T(\lfloor n/2 \rfloor) + c$ for all $n > 1$ and $T(1) = b$.
- How do we solve such recurrence relation?

Problem

Solving recurrence: $T(n) \leq T(\lfloor n/2 \rfloor) + c$ for all $n > 1$ and $T(1) = b$.

- How do we solve such recurrence relation?
- Assume that n is a power of 2. Then we can write:

$$
\begin{aligned}
T(n) &\leq T(n/2) + c \\
&\leq (T(n/4) + c) + c \\
&= T(n/4) + 2c \\
&\vdots \\
&\leq T(n/2^i) + i \cdot c \\
&\vdots \\
&\leq T(1) + \log n \cdot c \\
&\leq b + c \cdot \log n
\end{aligned}
$$

- So, $T(n) = O(\log n)$
- This is known as unrolling of the recursion.

Problem

Solving recurrence: $T(n) \leq T(\lfloor n/2 \rfloor) + c$ for all $n > 1$ and $T(1) = b$.

- Similarly, we can solve $T(n) \geq T(\lfloor n/2 \rfloor) + d$, $T(1) \geq e$ to show that $T(n) = \Omega(\log n)$.
- What if n is not a power of two?

Problem

Solving recurrence: $T(n) \leq T(\lfloor n/2 \rfloor) + c$ for all $n > 1$ and $T(1) = b$.

- Similarly, we can solve $T(n) \geq T(\lfloor n/2 \rfloor) + d$, $T(1) \geq e$ to show that $T(n) = \Omega(\log n)$.
- What if n is not a power of two?
- Note that $T(n) \leq T(n/2) + c$ does not make sense.
- Let n_1 and n_2 be such that $n_1 \leq n \leq n_2$ and n_1, n_2 are the closest integers to n which are powers of 2.
- Let $n_1 = 2^k$ and $n_2 = 2^{k+1}$.
- We know that $T(n_1) \leq T(n) \leq T(n_2)$
- Furthermore:

$$\begin{aligned} e + d \cdot k &\leq T(n_1) &\leq b + c \cdot k \\ e + d \cdot (k+1) &\leq T(n_2) &\leq b + c \cdot (k+1). \end{aligned}$$

- So, $T(n) = \Theta(\log n)$.

Problem

Solving recurrence: $T(n) \leq T(\lfloor n/2 \rfloor) + c$ for all $n > 1$ and $T(1) = b$.

- Similarly, we can solve $T(n) \geq T(\lfloor n/2 \rfloor) + d$, $T(1) \geq e$ to show that $T(n) = \Omega(\log n)$.
- What if n is not a power of two?
- Note that $T(n) \leq T(n/2) + c$ does not make sense.
- Let n_1 and n_2 be such that $n_1 \leq n \leq n_2$ and n_1, n_2 are the closest integers to n which are powers of 2.
- Let $n_1 = 2^k$ and $n_2 = 2^{k+1}$.
- We know that $T(n_1) \leq T(n) \leq T(n_2)$
- Furthermore:

$$\begin{aligned} e + d \cdot k &\leq T(n_1) &\leq b + c \cdot k \\ e + d \cdot (k+1) &\leq T(n_2) &\leq b + c \cdot (k+1). \end{aligned}$$

- So, $T(n) = \Theta(\log n)$.
- Informal comment: *Dropping floors and ceilings in these recurrence relation does not change the running time behaviour.*

- Recurrence relations of running time may also be written using big-(O, Ω, Θ) notation.
  - For example, for binary search the recurrence relation for running time may be written as:

$$T(n) = T(\lfloor n/2 \rfloor) + O(1) \text{ for all } n > 1; \quad T(1) = O(1)$$

  - Again, we can use the idea of unrolling to solve such recurrence relations.
- Exercise: Solve:

$$T(n) = T(n-1) + O(1) \text{ for all } n > 1; \quad T(1) = O(1)$$

- Recurrence relations of running time may also be written using big-(O, Ω, Θ) notation.
  - For example, for binary search the recurrence relation for running time may be written as:

$$T(n) = T(\lfloor n/2 \rfloor) + O(1) \text{ for all } n > 1; \quad T(1) = O(1)$$

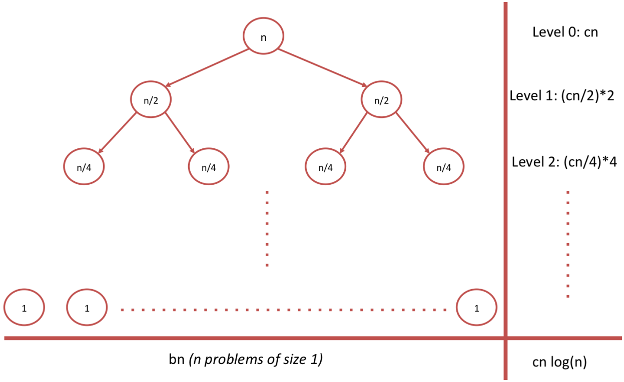
  - Again, we can use the idea of unrolling to solve such recurrence relations.
- Exercise: Solve:

$$T(n) = T(n-1) + O(1) \text{ for all } n > 1; \quad T(1) = O(1)$$

- Another method used to solve recurrence relations is called the substitution method.
  1. *Guess* the running time bound.
  2. Check that the bound holds using Induction.

- Another way of viewing unrolling of the recursion is Recurrence Trees.
  - For example, consider the following recurrence relation:

$$T(n) \leq 2 \cdot T(n/2) + c \cdot n \text{ for all } n > 1; \quad T(1) \leq b$$



- Solve: $T(n) \leq 2 \cdot T(n/2) + cn^2$; $T(1) \leq c$

- Solve: $T(n) \leq 2 \cdot T(n/2) + cn^2$; $T(1) \leq c$
- Solve: $T(n) \leq T(n/3) + c$; $T(1) \leq b$

- In Binary Search, we divided the array into two equal parts and then *zoomed* into one of the halves.
- Consider Ternary Search where we divide the array into three equal parts and then *zoom* into one of the three parts.

- In Binary Search, we divided the array into two equal parts and then *zoomed* into one of the halves.
- Consider Ternary Search where we divide the array into three equal parts and then *zoom* into one of the three parts.
- What is the running time of Ternary Search? Is it better than Binary Search?

Problem

Multiplying two n-bit numbers: Given two n-bit numbers, A and B, Design an algorithm to output $A \cdot B$.

Problem

Multiplying two n-bit numbers: Given two n-bit numbers, A and B, Design an algorithm to output $A \cdot B$.

- Solution 1: Use long multiplication.
- What is the running time of the algorithm that uses long multiplication?

Problem

Multiplying two n-bit numbers: Given two n-bit numbers, A and B, Design an algorithm to output $A \cdot B$.

- Solution 1: Use long multiplication.
- What is the running time of the algorithm that uses long multiplication? $O(n^2)$
- Is there a faster algorithm?

End