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.
  - How much time does each insert operations take? $O(1)$
  - How much time does each search operation take? $O(n)$
  - So, if the majority of the operations performed are search operations, then this data structure is perhaps not the right one.

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.

Problem

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

Algorithm

`BinarySearch-v1`(x, A, n)

- If A has no elements, then return("not present")
- Let *mid* denote the *middle* index of the array (i.e., $mid = \lfloor n/2 \rfloor$)
- If $(A[mid] = x)$, then return("present")
- Let A_L denote the *left-half* of the array and A_R denote the *right-half* of the array
- If $(x < A[mid])$
  - Search x in A_L
- else
  - Search x in A_R

Problem

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

Algorithm

BinarySearch-v2(x, A, n)

- If ($n \leq 0$), then return("not present")
- $mid \leftarrow \lfloor n/2 \rfloor$)
- If ($A[mid] = x$), then return("present")
- $A_L \leftarrow A[1...(mid - 1)]$
- $A_R \leftarrow A[(mid + 1)...n]$
- If ($x < A[mid]$)
    - ~~Search x in A_L~~ return(BinarySearch-v2($x, A_L, mid - 1$))
- else
    - ~~Search x in A_R~~ return(BinarySearch-v2($x, A_R, n - mid$))

Problem

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

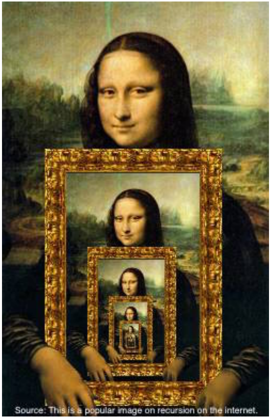
Algorithm

BinarySearch-v2(x, A, n)

- If ($n \leq 0$), then return("not present")
- $mid \leftarrow \lfloor n/2 \rfloor$)
- If ($A[mid] = x$), then return("present")
- $A_L \leftarrow A[1...(mid - 1)]$
- $A_R \leftarrow A[(mid + 1)...n]$
- If ($x < A[mid]$)
  - ~~Search x in A_L~~ return(BinarySearch-v2($x, A_L, mid - 1$))
- else
  - ~~Search x in A_R~~ return(BinarySearch-v2($x, A_R, n - mid$))

- The above function calls marked in red are called *recursive* function calls.
- The function `BinarySearch-v2` is called a *recursive function*.

- Recursion: Self reference.

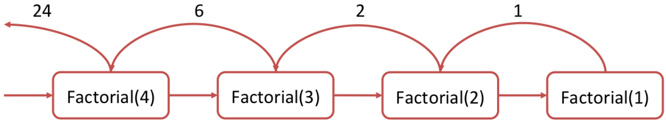


- In our context, we talk about recursive functions.

- Recursive function: A function that makes a call to itself.

Algorithm

`Factorial`(n)
- If $(n = 0$ or $n = 1)$return(1)
- $f \leftarrow$ `Factorial`$(n-1)$
- return$(n \cdot f)$



- Recursive function: A function that makes a call to itself.

Algorithm

`Factorial`(n)
- If $(n = 0$ or $n = 1)$return(1)
- $f \leftarrow$ `Factorial`$(n - 1)$
- return$(n \cdot f)$

- Base case: Returns result for small value of inputs. Defines the recursion termination condition.
- Reduction step: Assuming that the function returns correct value for smaller inputs use function calls on smaller inputs to compute the result on the given input.

- Question: How do we prove correctness of recursive functions?

Algorithm

`Factorial`(n)

- If $(n = 0$ or $n = 1)$return(1)
- $f \leftarrow$ `Factorial`$(n-1)$
- return$(n \cdot f)$

- Question: How do we prove correctness of recursive functions? Induction

Algorithm

Factorial(n)
- If ($n = 0$ or $n = 1$)return(1)
- $f \leftarrow$ Factorial($n - 1$)
- return($n \cdot f$)

- Question: How do we prove correctness of recursive functions? Induction
- Question: Is it always possible to avoid recursive functions?

Algorithm

Factorial(n)
- If ($n = 0$ or $n = 1$)return(1)
- $f \leftarrow$ Factorial($n - 1$)
- return($n \cdot f$)

- Question: How do we prove correctness of recursive functions? Induction
- Question: Is it always possible to avoid recursive functions? Yes.

Algorithm

`Factorial-iterative`(n)
- $f \leftarrow 1$
- for $i = 1$ to n
  - $f \leftarrow f \cdot i$
- return(f)

- In fact, there is some efficiency advantage in not using recursive functions.

- Question: How do we prove correctness of recursive functions? Induction
- Question: Is it always possible to avoid recursive functions? Yes.
- In fact, there is some efficiency advantage in not using recursive functions as function calls involve various time/space overheads.
- Why is recursion used then?

- Question: How do we prove correctness of recursive functions? Induction
- Question: Is it always possible to avoid recursive functions? Yes.
- In fact, there is some efficiency advantage in not using recursive functions as function calls involve various time/space overheads.
- Why is recursion used then?
  - In many cases, using recursion makes the program much simpler and easy to understand and analyse.
  - Many problems in Computer Science have inherent recursive structures (e.g., Fibonacci sequence)

- The Fibonacci sequence is defined in the following recursive manner:
  - Base case: $F(0) = 0$, $F(1) = 1$
  - For all $n > 1$, $F(n) = F(n-1) + F(n-2)$
- So, the sequence is:
  - $F(0) = 0$
  - $F(1) = 1$
  - $F(2) = F(1) + F(0) = 1 + 0 = 1$
  - $F(3) = F(2) + F(1) = 1 + 1 = 2$
  - $F(4) = F(3) + F(2) = 2 + 1 = 3$
  - $\vdots$

- The Fibonacci sequence is defined in the following recursive manner:
  - Base case: $F(0) = 0$, $F(1) = 1$
  - For all $n > 1$, $F(n) = F(n-1) + F(n-2)$
- So, the sequence is: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$
- The problem itself is defined in a recursive manner. So, it is natural to write a recursive method to solve this.

Algorithm

`Recursive-Fib`(n)
  If ($n = 0$ or $n = 1$)return(n)
  - return(`Recursive-Fib`($n-1$) + `Recursive-Fib`($n-2$))

- The Fibonacci sequence is defined in the following recursive manner:
  - Base case: $F(0) = 0$, $F(1) = 1$
  - For all $n > 1$, $F(n) = F(n-1) + F(n-2)$
- So, the sequence is: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$
- The problem itself is defined in a recursive manner. So, it is natural to write a recursive method to solve this.

Algorithm

`Rfib`(n)
  If $(n = 0 \text{ or } n = 1)$return$(n)$
  - return(`Rfib`$(n-1)$ + `Rfib`$(n-2)$)

- How do we analyse the running time of the above algorithm?

Algorithm

$\texttt{Rfib}(n)$
- If $(n = 0 \text{ or } n = 1)$return$(n)$
- return$(\texttt{Rfib}(n-1) + \texttt{Rfib}(n-2))$

- How do we analyse the running time of the above algorithm?

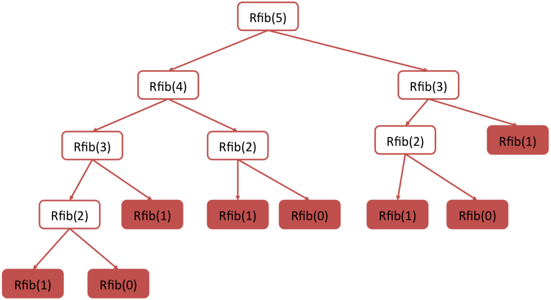


Figure : Recursive call tree for recursive fibonacci algorithm.

Algorithm

`Rfib`(n)
   If $(n = 0$ or $n = 1)$return(n)
   - return(`Rfib`$(n-1)$ + `Rfib`$(n-2)$)

- How do we analyse the running time of the above algorithm?
- Note that the same recursive call is made multiple times (e.g., `Rfib`(2))

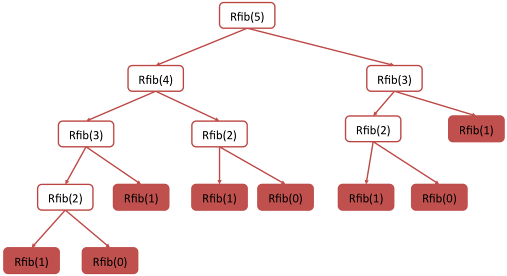


Figure : Recursive call tree for recursive fibonacci algorithm.

Algorithm

`Rfib`(n)
 If ($n = 0$ or $n = 1$)return(n)
 - return(`Rfib`$(n-1)$ + `Rfib`$(n-2)$)

- How do we analyse the running time of the above algorithm?
- Note that the same recursive call is made multiple times (e.g., `Rfib(2)`)
- In general, there are a lot of redundant calls.
- The running time of the above recursive algorithm can in fact be shown to be $\Omega(2^{n/2})$.
- Question: Can we find the n^{th} fibonacci number much faster than this?

Problem

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

Algorithm

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(BinarySearch$(x, A, i, mid - 1)$)
- else return(BinarySearch$(x, A, mid + 1, j)$)

Problem

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

Algorithm

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(BinarySearch$(x, A, i, mid - 1)$)
- else return(BinarySearch$(x, A, mid + 1, j)$)

- How do we prove the correctness of above algorithm?

Problem

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

Algorithm

`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(`BinarySearch`$(x, A, i, mid - 1)$)
- else return(`BinarySearch`$(x, A, mid + 1, j)$)

- How do we prove the correctness of above algorithm?
  Induction
- $P(i)$: The algorithm correctly searches any given element in any sorted array of size i.

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($\texttt{BinarySearch}(x, A, mid + 1, j)$)

- How do we prove the correctness of above algorithm? Induction
- $P(i)$: The algorithm correctly searches any given element in any sorted array of size i.
- Is $P(1)$ true?

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($\texttt{BinarySearch}(x, A, mid + 1, j)$)

- How do we prove the correctness of above algorithm? Induction
- $P(i)$: The algorithm correctly searches any given element in any sorted array of size i.
- Is $P(1)$ true?
- If $P(1), P(2), ..., P(k)$ are true, then is $P(k+1)$ also true?

Problem

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

Algorithm

`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(`BinarySearch`$(x, A, i, mid - 1)$)
- else return(`BinarySearch`$(x, A, mid + 1, j)$)

- What is the running time of the above algorithm in terms of the Big-O notation?

Problem

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

Algorithm

`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(`BinarySearch`$(x, A, i, mid - 1)$)
- else return(`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.
- Try writing a recurrence-relation for $T(n)$.

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?

End