

ACOL106: Data Structures and Algorithms

Ragesh Jaiswal, IITD

Introduction

Digression: Binary Search → Recursive Functions → 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: 0, 1, 1, 2, 3, 5, 8, 13, ...
- 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 ($n = 0$ or $n = 1$) return(n)

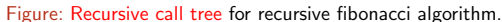
- return(Rfib($n-1$) + Rfib($n-2$))

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

Digression: Binary Search $\rightarrow$ Recursive Functions $\rightarrow$ Fibonacci Sequence

$$\text{Rfib}(n)$$

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



Introduction

Digression: Binary Search → Recursive Functions → Fibonacci Sequence

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)`)

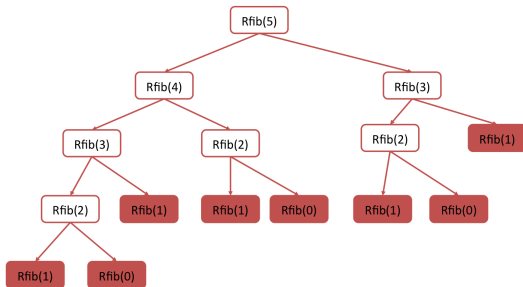


Figure: Recursive call tree for recursive fibonacci algorithm.

Introduction

Digression: Binary Search → Recursive Functions → Fibonacci Sequence

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?

Introduction

Digression: Binary Search

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( $x, A, mid + 1, j$ ))
```

Introduction

Digression: Binary Search

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?

Introduction

Digression: Binary Search

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 .

Introduction

Digression: Binary Search

Problem

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

Algorithm

$\text{BinarySearch}(x, A, i, j)$

- 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($\text{BinarySearch}(x, A, i, mid - 1)$)
- else return($\text{BinarySearch}(x, A, mid + 1, j)$)

- 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?

Introduction

Digression: Binary Search

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 .
- Is $P(1)$ true?
- If $P(1), P(2), \dots, P(k)$ are true, then is $P(k+1)$ also true?

Introduction

Digression: Binary Search

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?

Introduction

Digression: Binary Search

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 $T(n)$.

Introduction

Digression: Binary Search

Problem

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

Algorithm

$\text{BinarySearch}(x, A, i, j)$

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

- 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?

Introduction

Digression: Binary Search $\rightarrow$ Solving Recurrence

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**.

Introduction

Digression: Binary Search $\rightarrow$ Solving Recurrence

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?

Introduction

Digression: Binary Search $\rightarrow$ Solving Recurrence

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 $n_2 = 2^{k+1}$.
- Standard assumption: $T(n_1) \leq T(n) \leq T(n_2)$
- 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, $T(n) = \Theta(\log n)$.

Introduction

Digression: Binary Search $\rightarrow$ Solving Recurrence

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 $n_2 = 2^{k+1}$.
- Standard assumption: $T(n_1) \leq T(n) \leq T(n_2)$
- 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 relations does not change the running time behaviour.*

Introduction

Digression: Binary Search $\rightarrow$ Solving Recurrence

- 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:

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

Introduction

Digression: Binary Search $\rightarrow$ Solving Recurrence

- 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:

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

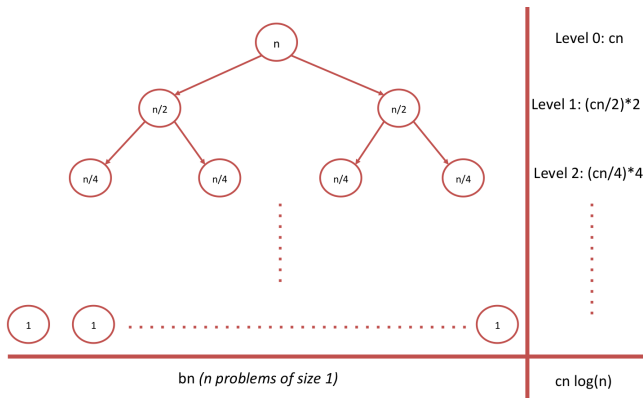
- Another method used to solve recurrence relations is called the **substitution** method.
 - 1 *Guess* the running time bound.
 - 2 Check that the bound holds using Induction.

Introduction

Digression: Binary Search → Solving Recurrence

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

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



End