Binary Search

Overview

- **Recursive solutions**
- **Implementation**

Problem Solving

- **Problem: Compute the factorial of a given integer n>=0**
  - **If n=0, factorial =1**
  - **If n > 0 ....?**

Problem Solving

- **Problem: Compute the factorial of a given integer n>=0**
  - `If n=0, factorial =1`
  - `If n > 0`
    1. `factorial = n * (n-1)* (n-2)*…*1`
    2. `factorial = n * factorial (n-1)`

Problem Solving – Different Approaches

- **Iterative Solution**
  - `factorial = n * (n-1) * (n-2) * … * 1`
  - **Use a looping construct**

- **Recursive Solution – direct, based on the mathematical formula**
  - `factorial = n * factorial (n-1)`
  - **Recursive function call**

Factorial - Recursive function

```
int factorial (int n){
// returns the factorial of n, given n>=0
   if (n<2)
      return 1;
   else
      return n * factorial (n-1);
}
```

```
factorial(3)   3*factorial(2)   3*2*factorial(1)     3*2*1
```

Factorial - Recursive calls and returns

main	x=factorial(3)	x gets 6
factorial(3)	3*factorial(2)	returns 6
factorial(2)	2*factorial(1)	returns 2
factorial(1)	//recursion stops	returns 1

Recursive Solution

- Divide into smaller problems
- Solve the sub problems recursively (direct solution when the sub problem size is small enough)
- Obtain the solution to the original problem from the solutions to the smaller sub problems

```
fib(n)= fib(n-1) + fib(n-2)
```

- Problems with natural recursive solution
- Simple, elegant and concise code

Recursion – Base Case, Recursive case

- **Recursive Case**
  - Subproblems are large enough to solve recursively
- **Base Case**
  - Direct solution (without recursion) when the subproblem is small enough
- More than one base case / recursive case possible as in `fibonacci()`
  - `fib(n)= fib(n-1) + fib(n-2)`

```
int fibbonacci(int n)
{
   if(n == 0){
     return 0;
   } else if(n == 1) {
     return 1;
   } else {
    return (fibbonacci(n-1) + fibbonacci(n-2));
   }
}
```

Searching Problem : Formal Specification

- **Input : A sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a key k**
- **Output : An index i such that k=A[i] or the special value -1 if k does not appear in A**

Searching Problem : Sorted sequence

- **Input : A sequence of n numbers** $A=<a_1, a_2, \ldots, a_n>$ **such that** $a_1 <= a_2 <= \ldots <= a_n$ **and a key k**

- Linear Search ?
  - Number of comparisons?
  - Can we reduce the number of comparisons, given the **input is sorted?**

Example

A: 3 5 7 9 11 12 35 40 48 52 65 k : 48

- **You know how to reduce the number of comparisons, right ?**

Binary Search

- Input : A sequence of n numbers A=<a_1, a_2,…, a_n> such that a_1 <= a_2 <= … <= a_n and a key k
- Binary Search
  - Compare k with the middle element of the sequence, say A[mid]
  - If k = A[mid] return mid
  - If k < A [mid] search in the first half, (A[1..mid-1])
  - If k > A [mid] search in the second half, (A[mid+1..n])
- Number of comparisons?

Example

A: 3 5 7 9 11 12 35 40 48 52 65 k : 48

A[mid] is 12 , k > 12 search A [7 .. 11]

A: 3 5 7 9 11 12 35 40 48 52 65

A[mid]=48 matches k

Search finished with just 2 comparisons

k : 65 ? k:3 ? k:6?

Binary Search

- Binary Search
  - If k < A [mid] search in the first half (A[1..mid-1])
  - If k > A [mid] search in the second half (A[mid+1..n])
- Size of the sequence to be searched is reduced to half
- n → n/2 → n/4 → … →1

Binary Search – Algorithm

```
BinarySearch (A, m, n, k)
   if (m>n) ………………….????   //Base Case
   mid=(m+n)/2
   if A[mid]=k  return mid;      //Base Case
   else if k < A[mid]
      BinarySearch(A, m, mid-1, k)
   else if k > A[mid]
      BinarySearch(A, mid+1, n, k)
```

Binary Search – Algorithm

```
BinarySearch (A, m, n, k)
   if (m>n) return -1;    //Base Case
   mid=(m+n)/2
   if A[mid]=k  return mid;     //Base Case
   else if k < A[mid]
      BinarySearch(A, m, mid-1, k)
   else if k > A[mid]
      BinarySearch(A, mid+1, n, k)
```

Calls / Returns

Call Sequence :

```
main()→  f()→   g()→ h()
```

Return Sequence:

```
main()←  f()←   g()← h()
```

Return Address

main()	f()	g()	h()
......	......	......	......
f()	g()	h()	
S1	S2	S3	return
...	...	...	
	return	return	

Return Address : Address of the instruction in the caller function to which control should return

Passing parameters and return values

```
main()                    f(a, b)
x = f(y, z)                ......
......                     return(c)
```

- For each active function store return address, parameters, local variable etc.
- Each activation require separate storage area

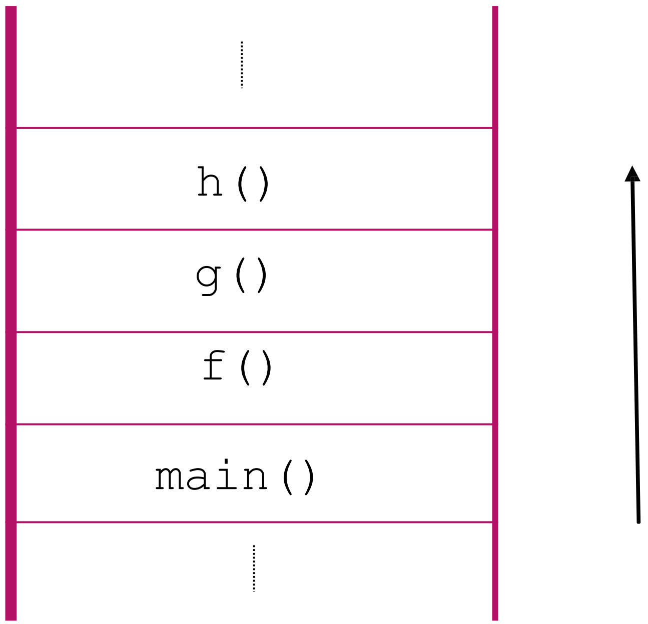
Activation Records

- Activation Record / Stack Frame
- Data Area in memory for storing the data relevant to one activation of the function (parameters, locals, return address…)
  - Set up on entry to the function
  - Area can be reallocated after return
- Stacked one on top of the other (call stack/control stack)

`main()` → `f()` → `g()` → `h()`

How many activation records when `h()` is active?

Stack of Activation records



Activation Records – Recursive Functions

- **Multiple activations of the same function**
  - **Multiple activation records**
- **Time for setting up activation records**
- **Space in stack**

Conclusion

- **Recursive solution- simple, elegant and concise code**
- **Functional Programming Languages – Scheme, ML**
- **More space requirement – keep track of status**
- **Common mistake – Missing base case – never stops**

Reference

- **Text Books and other materials on Programming Language Theory/Compilers/Algorithms**

Reference

- **Text Books and other materials on Programming Language Theory/Compilers/Algorithms**