CS1100 – Introduction to Programming

Instructor:

Shweta Agrawal (shweta.a@cse.iitm.ac.in)

Lecture 23

Recursion Example: The Binomial Coefficient

$\binom{n}{k}$ denotes :

Recursion Example: The Binomial Coefficient

$\binom{n}{k}$ denotes :

- Number of ways of choosing k items from a collection of n items.

Recursion Example: The Binomial Coefficient

$\binom{n}{k}$ denotes :

- Number of ways of choosing k items from a collection of n items.
- Number of subsets of size k of a set of size n.

Recursion Example: The Binomial Coefficient

$\binom{n}{k}$ denotes :

- Number of ways of choosing k items from a collection of n items.
- Number of subsets of size k of a set of size n.
- Coefficient of x^k in the expansion of $(1+x)^n$.

Recursion Example: The Binomial Coefficient

$\binom{n}{k}$ denotes :

- Number of ways of choosing k items from a collection of n items.
- Number of subsets of size k of a set of size n.
- Coefficient of x^k in the expansion of $(1 + x)^n$.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Task : Write a function which: given n and k computes $\binom{n}{k}$.

Pascal's Triangle

$$\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}$$

Recursion Example: The Binomial Coefficient

$\binom{n}{k}$ denotes :

- Number of ways of choosing k items from a collection of n items.
- Number of subsets of size k of a set of size n.
- Coefficient of x^k in the expansion of $(1+x)^n$.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Task : Write a function which: given n and k computes $\binom{n}{k}$.

Pascal's Triangle

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}
$$

Pascal's Identity :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Recursion Example: The Binomial Coefficient

$\binom{n}{k}$ denotes :

- Number of ways of choosing k items from a collection of n items.
- Number of subsets of size k of a set of size n.
- Coefficient of x^k in the expansion of $(1+x)^n$.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Task : Write a function which: given n and k computes $\binom{n}{k}$.

Pascal's Triangle

$$\begin{array}{ccccccccccc} &&&&& 1 &&&&& \\ &&&& 1 && 1 &&&& \\ &&& 1 && 2 && 1 &&& \\ && 1 && 3 && 3 && 1 && \\ & 1 && 4 && 6 && 4 && 1 & \\ 1 && 5 && 10 && 10 && 5 && 1 \end{array}$$

Pascal's Identity :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Base : $\binom{n}{0} = \binom{n}{n} = 1$.

Recursion Example: The Binomial Coefficient

$$\texttt{bin(n,k)} = \begin{cases} 1 \text{ if } k = 0 \\ 1 \text{ if } n = k \\ \texttt{bin(n-1,k-1)} + \texttt{bin(n-1,k)} & \text{otherwise} \end{cases}$$

Recursion Example: The Binomial Coefficient

$$\texttt{bin(n,k)} = \begin{cases} 1 \text{ if } k = 0 \\ 1 \text{ if } n = k \\ \texttt{bin(n-1,k-1)} + \texttt{bin(n-1,k)} & \text{otherwise} \end{cases}$$

```
int binom(int n, int k)
{
 if(k == 0 || n == k)
    return 1;
 int s = binom(n-1,k-1);
 int t = binom(n-1,k);
 return (s+t);
}
```

Recursion Example: The Binomial Coefficient

$$\texttt{bin(n,k)} = \begin{cases} 1 \text{ if } k = 0 \\ 1 \text{ if } n = k \\ \texttt{bin(n-1,k-1)} + \texttt{bin(n-1,k)} & \text{otherwise} \end{cases}$$

```
int binom(int n, int k)
{
 if(k == 0 || n == k)
    return 1;
 int s = binom(n-1,k-1);
 int t = binom(n-1,k);
 return (s+t);
}
```

```
#include<stdio.h>
int main()
{
 int n,k;
 printf("Entern n, k : ");
 scanf("%d %d",&n,&k);
 printf("%d\n",binom(n,k));
 return 0;
}
```

Exercise : Print the Pascal's Triangle

$\binom{0}{0}$

$\binom{1}{0} \quad \binom{1}{1}$

$\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2}$

$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$

$\binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4}$

$\binom{5}{0} \quad \binom{5}{1} \quad \binom{5}{2} \quad \binom{5}{3} \quad \binom{5}{4} \quad \binom{5}{5}$

Exercise : Print the Pascal's Triangle

$$\binom{0}{0}$$

$$\binom{1}{0} \quad \binom{1}{1}$$

$$\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2}$$

$$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$$

$$\binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4}$$

$$\binom{5}{0} \quad \binom{5}{1} \quad \binom{5}{2} \quad \binom{5}{3} \quad \binom{5}{4} \quad \binom{5}{5}$$

```c
#include <stdio.h>

int main()
{
 int i,j,n;
 printf("Enter n :");
 scanf("%d",&n);
 for (i=0;i<n;i++)
 {
   for (j=0;j<=i;j++) {
    printf("%6d",binom(i,j));
   }
   printf("\n");
 }
}
```

Recursion Example: Virahanka Numbers/Fibonacci Numbers

Suppose you have an unlimited supply of bricks of heights 1 and 2. You want to construct a tower of height n. In how many ways can you do this? Let this number be V_n.

Recursion Example: Virahanka Numbers/Fibonacci Numbers

Suppose you have an unlimited supply of bricks of heights 1 and 2. You want to construct a tower of height n. In how many ways can you do this? Let this number be V_n.
Right answer comes from the right questions.

Recursion Example: Virahanka Numbers/Fibonacci Numbers

Suppose you have an unlimited supply of bricks of heights 1 and 2. You want to construct a tower of height n. In how many ways can you do this? Let this number be V_n.
Right answer comes from the right questions.

Question: In any given arrangement, what is the bottom-most brick, is it of height 1 or 2?

Recursion Example: Virahanka Numbers/Fibonacci Numbers

Suppose you have an unlimited supply of bricks of heights 1 and 2. You want to construct a tower of height n. In how many ways can you do this? Let this number be V_n.
Right answer comes from the right questions.

Question: In any given arrangement, what is the bottom-most brick, is it of height 1 or 2?
Two answers possible (It could be 1 or it could be 2, but not both).

Recursion Example: Virahanka Numbers/Fibonacci Numbers

Suppose you have an unlimited supply of bricks of heights 1 and 2. You want to construct a tower of height n. In how many ways can you do this? Let this number be V_n.
Right answer comes from the right questions.

Question: In any given arrangement, what is the bottom-most brick, is it of height 1 or 2?
Two answers possible (It could be 1 or it could be 2, but not both).

Number of ways of building a tower of height n	$=$	Number of ways of building a tower of height n with bottom-most brick of height 1	$+$	Number of ways of building a tower of height n with bottom-most brick of height 2.

Recursion Example: Virahanka Numbers/Fibonacci Numbers

Suppose you have an unlimited supply of bricks of heights 1 and 2. You want to construct a tower of height n. In how many ways can you do this? Let this number be V_n.
Right answer comes from the right questions.

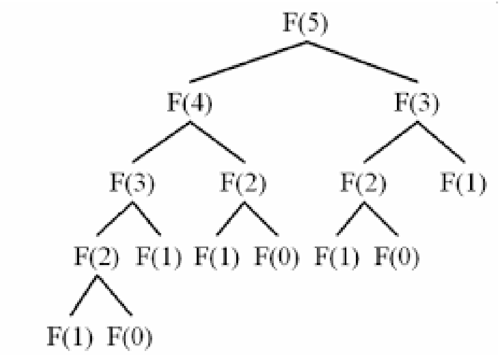
Question: In any given arrangement, what is the bottom-most brick, is it of height 1 or 2?
Two answers possible (It could be 1 or it could be 2, but not both).

Number of ways of building a tower of height n	=	Number of ways of building a tower of height n with bottom-most brick of height 1	+	Number of ways of building a tower of height n with bottom-most brick of height 2.

That is, $V_n = V_{n-1} + V_{n-2}$ Nice !. But how do we compute V_n

Recursion Example: Virahanka Numbers

```
int Virahanka(int n)
{
  if(n == 1) return 1; // V_1
  if(n == 2) return 2; // V_2
  // returning V_{n-1} + V_{n-2}
  return Virahanka(n-1) + Virahanka(n-2);
}
```



Sorting an Array

Sorting an Array

- **Iterative Thinking**: We talked about selection sort.

Sorting an Array

- **Iterative Thinking**: We talked about selection sort.
- **Recursive Thinking**: Take out the last element, sort the first $n-1$ elements recursively, invoking the same function recursively. And then insert this last element in the right position.

Sorting an Array

- **Iterative Thinking**: We talked about selection sort.
- **Recursive Thinking**: Take out the last element, sort the first $n - 1$ elements recursively, invoking the same function recursively. And then insert this last element in the right position. Hey, this is insertion sort !

Sorting an Array

- **Iterative Thinking**: We talked about selection sort.
- **Recursive Thinking**: Take out the last element, sort the first $n-1$ elements recursively, invoking the same function recursively. And then insert this last element in the right position. Hey, this is insertion sort !
- **(Even better) Recursive Thinking**: Divide the array into two halves, recursively sort them, merge the resulting arrays into one array keeping the result to be sorted.
  This is new ! - called **merge sort**.

Searching in a Sorted Array

Searching in a Sorted Array

Given an array A that is sorted, search for a given element `key` in the array.

Searching in a Sorted Array

Given an array A that is sorted, search for a given element `key` in the array.

- **Iterative Thinking**: Run over each element in the array by using a `for` loop, check if the key is equal to any of them.

Searching in a Sorted Array

Given an array A that is sorted, search for a given element `key` in the array.

- **Iterative Thinking**: Run over each element in the array by using a `for` loop, check if the key is equal to any of them.
- **Recursive Thinking**: Take out the last element, check if it is equal to the key, if so stop, else search for the key in the remaining $n-1$ elements recursively.

Searching in a Sorted Array

Given an array A that is sorted, search for a given element `key` in the array.

- **Iterative Thinking**: Run over each element in the array by using a `for` loop, check if the key is equal to any of them.
- **Recursive Thinking**: Take out the last element, check if it is equal to the key, if so stop, else search for the key in the remaining $n-1$ elements recursively.
- **(Even better) Recursive Thinking**: Divide the array into two halves, check if the key element is less than the middle element - then search in the left half of the array, else search in the right half of the array. This is called **binary search**.

Searching in a Sorted Array

Given an array A that is sorted, search for a given element `key` in the array.

- **Iterative Thinking**: Run over each element in the array by using a `for` loop, check if the key is equal to any of them.
- **Recursive Thinking**: Take out the last element, check if it is equal to the key, if so stop, else search for the key in the remaining $n-1$ elements recursively.
- **(Even better) Recursive Thinking**: Divide the array into two halves, check if the key element is less than the middle element - then search in the left half of the array, else search in the right half of the array. This is called **binary search**.

Coding Binary Search

```
#include <stdio.h>

int binarySearch(int array[], int x, int low, int high) {
  if (high >= low) {
    int mid = low + (high - low) / 2;

    // If found at mid, then return it
    if (array[mid] == x)
      return mid;

    // Search the left half
    if (array[mid] > x)
      return binarySearch(array, x, low, mid - 1);

    // Search the right half
    return binarySearch(array, x, mid + 1, high);
  }
  return -1;
}

int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 4;
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
}
```

Analyzing Binary Search

Given an array A that is sorted, search for a given element `key` in the array.

Analyzing Binary Search

Given an array A that is sorted, search for a given element `key` in the array.

- Divide the array into two halves, check if the key element is less than the middle element - then search in the left half of the array, else search in the right half of the array.

Analyzing Binary Search

Given an array A that is sorted, search for a given element `key` in the array.

- Divide the array into two halves, check if the key element is less than the middle element - then search in the left half of the array, else search in the right half of the array.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.

Analyzing Binary Search

Given an array A that is sorted, search for a given element `key` in the array.

- Divide the array into two halves, check if the key element is less than the middle element - then search in the left half of the array, else search in the right half of the array.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = T(n/2) + C$.

Analyzing Binary Search

Given an array A that is sorted, search for a given element `key` in the array.

- Divide the array into two halves, check if the key element is less than the middle element - then search in the left half of the array, else search in the right half of the array.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = T(n/2) + C$.
- This gives us $T(n) \approx \log_2(n)$.

The relations with $T(n)$ in LHS are called *recurrence relations*