Algorithms

9. Recursion

Handwritten by pankaj kumar

Recursion

Date
Page No. 0

1. Introduction (1-6)
2. Factorial (6-6)
3. Fibonacci (7-7)
4. Rererse string (7-7)
5. Reverse number (7-7)
6. Power (n^p) (7-7)
7. GCD (7-7)
8. Decimal to Binary (8-8)
9. Sum of Digits (8-8)
10. Check Palindrome (8-8)
11. Check Prime (8-8)
12. Reverse Array (9-9)
13. Print all subsequences of a string (9-9)
14. Subsequences sum (subset sum) with equal sum (10-10)
15. Any one subset sum with equal sum (10-11)
16. Count subset sum with equal sum (11-11)
17. All Permutation (12-13)
18. kth permutation seq. of First N natural no. (14-15)
19. kth symbol in grammar (16-16)
20. N-bit binary no. having more 1's than 0 in all prefix (16-17)
21. Josephus problem (18-18)

Element No duplicate

1. Recursion

Date

- The Process in which a Function calls itself is called recursion and corresponding function is called as recursive function.
- Recursion is a method where the solution to a problem depends on solution to smaller instances of the same problem. By same nature it actually means that the approach that we used to solve the orisinal problem can be used to solve smaller problems as well.

* Principle of Mathematical Induction (PMI):

PMI is a technique for proving a statement, a Formula, or a theorem that is asserted about every natural number. It has Following three steps:

1. Step of trivial case: In this step, we prove the desired statement for $n=1$.
2. Step of assumption: In this step, we will asrume that the desired statements is valid for $n=k$,
3. To prove step: From the result of assumption step, we will prove that $n=k+1$ also holds for the desired equation.

Example:-

prove that, $S(n): 1+2+3+\cdots+n = (n*(n+1))/2$

Step 1: For $n=1$, $S(1)=1$ is true.

Step2: Assume the given statements is true for $n=k$, i.e,

$1+2+3+\cdots+k = (k*(k+1))/2$

Step3: let's prove the statement for $n=k+1$ using step 2.

Adding $(k+1)$ to both LHS and RHS in step 2 result

$1+2+3+\cdots+(k+1) = (k*(k+1))/2 + (k+1)$

Or, $1+2+3+\cdots+(k+1) = (k+1)*((k+2)/2)$

Here, this statement is the same as we obtained above. Hence proved.

Note:- These three steps of PMI are related to the three steps of Recursion. which are as follows.

1. Induction step (IS) and Induction Hypothesis (IH):-

Here, the Induction step is the main problem which we are trying to solve using recursion, whereas Induction Hypothesis is the sub-problem, using which we'll solve the induction step.

- Induction step :- sum of first n natural numbers - F(n)
- Induction Hypothesis :- This gives us the sum of first n-1 natural numbers - F(n-1)

2. Express F(n) in terms of F(n-1) and write code: F(n) = F(n-1)+n

```
int F(int n){
    int ans = F(n-1); //Induction Hypothesis step
    return ans+n; //solving problem from result in previous step
}
```

3. The code is still not complete. The missing part is the base case.

4. After the dry run, we can conclude that for n equal 1 answer is 1, which we already know. so we will use this as a base case;

```
int F(int n){
    if(n==1) { //Base case
        return 1;
    }
    int ans = F(n-1);
    return ans+n;
}
```

Thus, To ~~solve~~ Find our solution we start problem and tell the function to compute rest for us using the particular hypothesis

✳ Working of Recursion :-

- Base Case – A Recursive function must have a terminating condition at which the process will stop calling itself. such case is known as base case.

-In the absence of Base case, it will keep calling itself and get stuck in an infinite loop.

- Recursive call: The recursive function will recursively invoke itself on the smaller version of the problem.
- we need to be careful while writing this step as it is crucial to correctly figure out what your smaller problem is. On the smaller problem, the original problems solutions depends.
- Small calculation: Generally we perform a calculation step in each recursive call. we can achieve this calculation step before or after the recursive call depending upon the nature of prob

Note:- Recursion uses an in-built stack which stores recursive calls. Hence the number of recursive calls must be small as possible.

[*] Example:

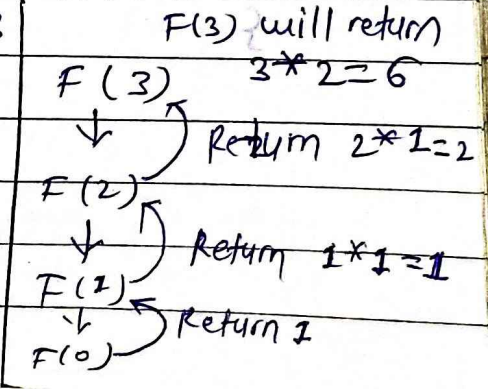
Q. we want to find out the factorial of a natural number.

Approach: Figuring out the three steps of of PMI and then relating the same using recursion.

1. Induction step: calculating the factorial of a number n - F(n)
   Induction Hypothesis: we assume we have already obtained the factorial of n-1, through recursion - F(n-1)
2. Expressing F(n) in terms of F(n-1): F(n) = n * F(n-1).

```
int F(int n){
    int ans = F(n-1); // Assumption step
    return ans*n; // solving problem from Assumption step
}
```

3. this code is still not complete. The missing part is the base case. Now we will dry run to find the case where the recursion step needs to stop. assume for F=3 - as we can see, we already know the answer of n=0, that is 1. so we will keep this as our basecase. Hence the code now becomes.



```
int f(int n){
    if(n==0)
        return 1;  // Base case
    int ans= f(n-1); // Recursive call
    return n* ans   // Small calculation
}
```

* Disadvantages and advantages Over Iterative.

- every recursive program can be written iteratively and vice versa is also true. The recursive program has greater space requirements than iterative program as all function remain in the stack until the base case is reached. It also has greater time requirements bcz of function calls & return overhead.
- Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Honoi, etc. For such problems, It is preferred to write recursive code.

* Direct and InDirect

Direct

```
void fun1()
{
    //some code...
    fun1();
    //some code...
}
```

Indirect

```
void fun1()
{
    //some code...
    fun2();
    //some code...
}
void fun2()
{
    //some code...
    fun1();
    //some code...
}
```

Tail Recursion

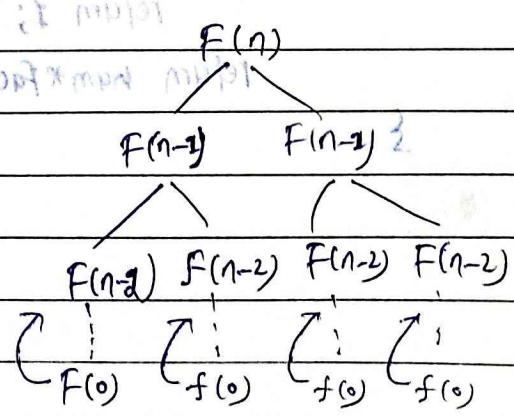
→A recursive function is tail recursive when a recursive call is the last thing executed by the function.

```
→void print(int n)
{
    if(n<0) return;
    cout<<" "<<n;
    print(n-1); //The last executed statement
}
```

- Why do we care?
- The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by the compiler. Compilers usually execute recursive procedure by using a stack. When a procedure is called, its information is pushed onto a stack, and when the function terminates the information is popped out of the stack.
- Thus for the non-tail-recursive functions, the stack depth (maxm amount of stack space at any time during compilation) is more. The idea used by compilers to optimize tail-recursive function is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use.

Tree Recursion

```
void Fun(int n)
{
    if(n>0)
    {
        cout<<n;
        Fun(n-1);
        Fun(n-1);
    }
}
```



[*] Nested Recursion:

```
int fun(int n)
{
    if(n>100)
        return n-10;
    return fun(fun(n+11));
```

[*] Example Algorithm of Recursion

- Fibonacci Series, Factorial Finding
- Merge Sort, Quick Sort
- Binary Search
- Tree Traversal and many Tree problems: In order, Preorder, Postorder
- Graph Traversal: DFS [Depth First Search] & BFS [Breadth First search]
- Dynamic Programming Examples
- Divide and Conquer Algorithms
- Tower of Hanoi puzzle.
- Backtracking Algorithms

[*] Solved Problems

Basic

(1) Factorial

```
int fact(int num){
    if(num==0)
        return 1;
    return num*fact(num-1)
}
```

num = 5

5 * F(4) = 120
↓
4 * F(3) = 24
↓
3 * F(2) = 6
↓
2 * F(1) = 2
↓
1 * F(0) = 1
↓
F(0) → 1

② Fibonacci

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

③ Reverse string

```
string reverse(string s, int end){
    if(end == -1)
        return "";
    return s[end] + reverse(s, end-1);
}
```

④ Reverse Number

```
int reverse(int n, int ans=0){
    if(n==0)
        return ans;
    ans = ans*10 + n%10
    return reverse(n/10, ans);
}
```

⑤ Power (n^p)

```
int power(int n, int p){
    if(p==0)
        return 1;
    return n* power(n, p-1);
}
```

⑥ GCD

```
int GCD(int a, int b){
    if(a==0)
        return b;
    return GCD(b%a, a);
}
```

7. Decimal to Binary

```
int DecimalToBinary(int n){
    if(n==1)
        return 1;
    return DecimalToBinary(n/2)*10 + n%2;
}
```

8. Sum of Digits

```
int DigitSum(int n){
    if(n==0)
        return 0;
    return n%10 + DigitSum(n/10);
}
```

9. Check Palindrome

```
bool checkP(string s, int i, int j){ // i = start index
    if(i==j)                          // j = last index
        return true;
    if(s[i] != s[j])
        return false;
    return checkP(s, i+1, j-1);
}
```

10. Check Prime

```
bool CheckPrime(int n, int i=2){
    if(i*i >= n)
        return true;
    if(i%n == 0)
        return false;
    return CheckPrime(n, i+1);
}
```

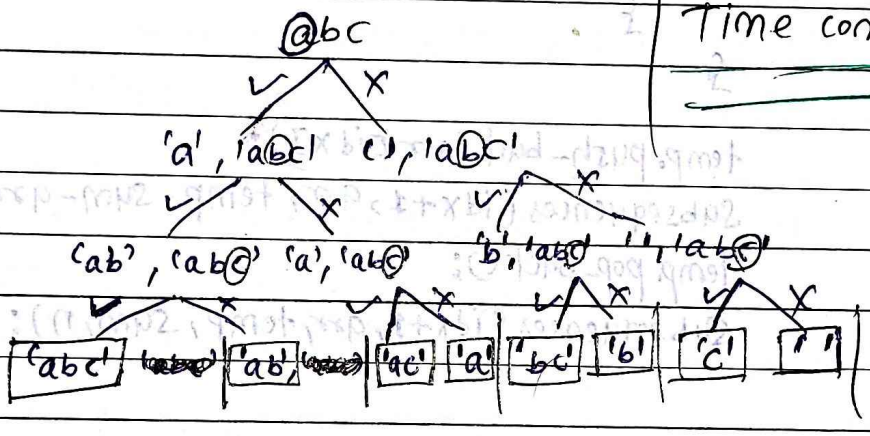
Medium & Hard →

(11) Reverse Array

```
void f(int i, int arr[], int n){ //i=0 & n= size of array
        if(i>=n/2) return;
        swap(arr[i], arr[n-i-1]);
        f(i+1, arr, n);
}
```

(12) Print all subsequences of a String (Subset)

- subsequences is generated by deleting some character of a given string without chaging its order.
- Input = abc.
- output = a, b, c, ab, bc, ac, abc



Time complexity (2^n)

Program:

```
void printSubsequences(int idx, string input, string output){
        if(idx == input.size()){
                cout<<output<<endl;
                return;
        }
        printSubsequences(idx+1, input, output+input[idx]);
        printSubsequences(idx+1, input, output);
}

void main(){
        printSubsequences(0, "abc", "");
}
```

[Note:- Subset need not to follow sequence, relative order, but subsequences must]

Date
Page No. 10

(13) Print all subsequences whose sum is sum(k) (Subset sum)

Input: arr[] = {1,2,3}, k=3

Output: 1 2, 3

Program:-

```
void Subsequences(int idx, int arr[], vector<int> temp, int sum,
                                                    int n){
    if(idx == n){
        if(sum==0){
            for(auto i:temp) cout<<i<<" ";
            cout<<endl;
        }
    }

    temp.push_back(arr[idx]);
    Subsequences(idx+1, arr, temp, sum-arr[idx], n);
    temp.pop_back();
    Subsequences(idx+1, arr, temp, sum, n);
}

int main(){
    int arr[]={1,2,3}
    int n=3, sum=3;
    vector<int> temp;
    Subsequences(0, arr, temp, sum, n);
    return 0;
}
```

Note:- if we want to print any one subsequence with sum k.

program:-

```
void Subsequences(int idx, int arr[], vector<int> temp, int sum,
                                                    int n){
```

```
if(idx==n){
    if(sum==0){
        for(auto i:temp) cout<<i<<" ";
        cout<<endl;
        return true;
    }
    return false;
}
temp.push_back(arr[idx]);
if(subsequences(idx+1,arr,temp,sum-arr[idx],n) return true;
temp.pop_back();
if(subsequences(idx+1,arr,temp,sum,n) return true;
return false;
}
```

Note:- If we want to count number of subsequences whose sum is k.

```
int subsequences(int idx, int arr[], vector<int> temp, int sum,
                                                    int n){
    if(idx==n){
        if(sum==0) return 1;
        return 0;
    }
    temp.push_back(arr[idx]);
    int l= subsequences(idx+1,arr,temp,sum-arr[idx],n);
    tem.pop_back();
    int r= subsequences(idx+1,arr,temp,sum,n);
    return l+r;
}
```

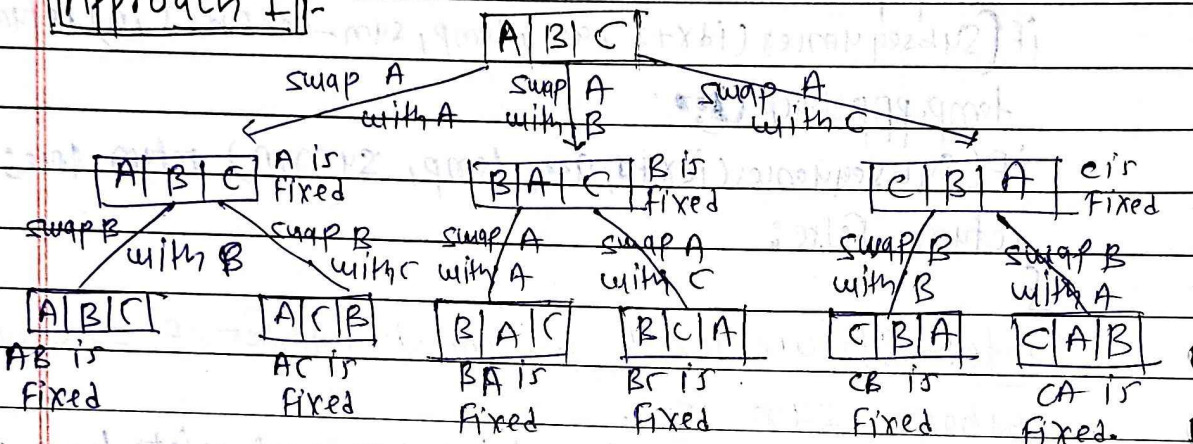
(14) Print all permutation of a string:

- permutation is also called an "arrangement number" or "order", is a rearrangement of the element of an ordered list S into a one-to-one correspondence with S itself.
- A string of length n has n! permutation

Input :- ABC

Output :- ABC, ACB, BAC, BCA, CBA, CAB

Approach 1 :-



program :-

```
void permute(string a, int idx, int n){
    if (idx == n){
        cout << a << endl;
        return;
    }
    for(int i=idx; i<n; i++){
        swap(a[idx], a[i]);
        permute(a, idx+1, n);
        swap(a[idx], a[i]); //Backtrack
    }
}
int main(){
    permute (0, "ABC", 3);
}
```

Time complexity :- $O(n*n!)$
Auxiliary space :- $O(n-idx)$

Approach-2

```
void permute (string s, string ans, int freq[]){
    if (ans.length() == s.length()){
        cout<<answer<<endl;
        return;
    }
    for(int i=0; i<s.length(); i++){
        if(!freq[i]){
            freq[i]=1;
            permute(s, ans+s[i], freq);
            freq[i]=0;
        }
    }
}
int main(){
    string s="ABC"
    int n=s.length();
    int freq[n]={0};
    permute(s, "", freq);
    return 0;
}
```

Time complexity :- $O(n*n!)$
Auxiliary space :- $O(n)$

(15) k-th Permutation Sequence of First N natural number.

Input N=3, k=4 / 1 2 3 4 : 123, 132, 213, (231), 312, 321
Output : 231

Naive Approach :-

the simple approach is to find all permutation sequences and output the kth out of them.

Efficient Approach :-

Given N=4 , k=9

There are 6 no. starting with 1: 1234, 1243, 1324, 1342, 1423, 1432
There are 6 no. starting with 2: 2134, 2143, (2314), 2341, 2413, 2431
similarly there are 6 no. starting with 3 and 6 no. starting with 4.

- This is bcz when we choose one place out of 4 places, there are 3 places remaining to be filled and those 3 places can be filled in 6 ways i.e (N-1)! ways.
- So we have to keep identifying which digit to choose.

Step 1
- Initially we have digit from {1,2,3,4}.
  since k=9, means it belongs to the second set of 6 numbers and hence, we begin with 2, i.e at index [(k/(n-1)!)]

Step 2
- Now the first place of answer will be 2 and we have now, k = 9%6 = [(k%(n-1)!)], and remaining digit will be {1,3,4}.

Now
There are 2 no. starting with 1: 134, 143
There are 2 no. starting with 3: 314, 341
There are 2 no. starting with 4: 413, 431
Now we will Repeat step 1 and step 2 until we get the exact kth permutation i.e k=1

Output will be :- 2314

Program :-

```
void kthPermute (int k, int n, int fact[], vector<int> num, string& ans){
    if(n==1){
        ans += to_string(num.back());
        return;
    }
    int idx = k/fact[n-1];
    if(k%fact[n-1]==0) idx--;

    ans += to_string(num[idx]);
    num.erase(num.begin()+idx);
    k-=fact[n-1]*idx;
    kthPermute (k, n-1, fact, num, ans);
}

int main(){
    int n=4, k=9;
    string ans="";
    int fact[]={1,1,2,6,24,120,720,720*7, 720*7*8, 720*7*8*9};
    vector<int> num;
    for(int i=0; i<n; i++){
        num.push_back(i+1);
    }
    kthPermute (k,n, fact, num, ans);
    cout<<ans;
    return 0;
}
```

Time complexity :- $O(n^2)$

(16) kth Symbol in Grammar

→ Grammar generation rule:- 1st row → 0 and then next row will be generated by using previous row bit to convert 0 by 01 and 1 by 10.

Grammar :-

```
0
0 1
0 1 1 0
0 1 1 0 1 0 0 1
```

same as previous row | inverse of previous row

Approach,

Input: n=4, k=5
Output: 1

Program :-

```
int kthSymbol(int n, int k){
    if(n==1) return 0;
    if(k > pow(2,n-2))
        return kthSymbol(n-1, k-int(pow(2,n-2)))==0 ? 1 : 0;
    else
        return kthSymbol(n-1,k);
}

int main(){
    cout << kthSymbol(4,5);
    return 0;
}
```

(17) Print N-bit binary numbers having more 1s than 0s in all prefixes.

Input : n=2
Output : 11 10

0 1 (x)

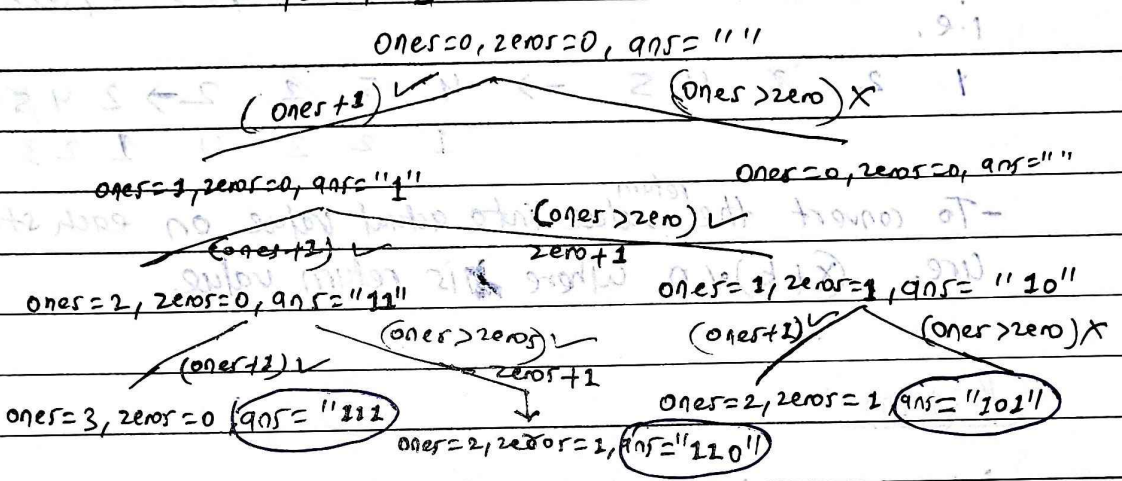
For prefix 0 there is more 0 than 1

Input : n=4
Output : 1111, 1110, 1101, 1100, 1011, 1010

1 0 0 1 (x)

For prefix 100 there is more 0s than 1's,

- A simple but not efficient solution will be to generate all N-bit binary numbers and print those numbers that satify the condition. the time complexity of this solution is exponential
- An efficient solution is to generate only those N-bit numbers that satisfy the given condition. We use recursion. At each point in the recursion, we append 0 and 1 to the partially formed number and recur with one less digit.



Program :-

```
void NBitBinary(int n, int ones=0, int zeros=0, string ans="") {
    if (ans.size() == n) {
        cout << ans << endl;
        return;
    }
    NBitBinary(n, ones+1, zeros, ans+"1");
    if (ones > zeros)
        NBitBinary(n, ones, zeros+1, ans+"0");
}

int main() {
    NBitBinary(4);
    return 0;
}
```

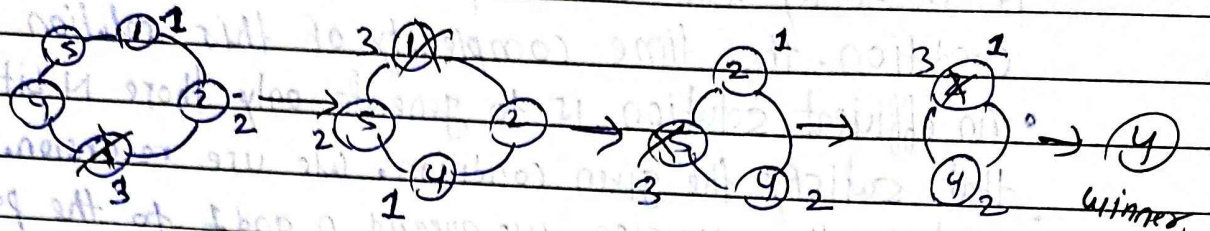
Time complexity (2^n)

Date

Page No. 18.

(18) Josephus Problem (Winner of the death in a circle)

Input n=5, k=3



We use recursion to solve this problem

- in second round we assume (4)→1, (5)→2, (1)→3, (2)→4

i.e.

1 2 ~~3~~ 4 5 → 4 5 ~~1~~ 2 → 2 4 ~~5~~ → ~~4~~ → 4

1 2 3 4 1 2 3 1 2 1

- To convert the return value into actual value on each step we use, (x+k)%n where x is return value

Program :-

```
int answer(int n, int k){
    if (n==1) return 0;
    int x = answer(n-1, k);
    return (x+k)%n;
}

int main(){
    cout << answer(5,2)+1;
    return 0;
}
```