# Contents

# Chapter 1

# Recursive Functions

**Recursion:**

In programming terms a recursive function can be defined as a routine that calls itself directly or indirectly.

Using recursive algorithm, certain problems can be solved quite easily. Towers of Hanoi (TOH)is one such programming exercise. Try to write an *iterative* algorithm for TOH. Moreover, every recursive program can be written using iterative methods (Refer Data Structures by Lipschutz).

Mathematically recursion helps to solve few puzzles easily.

For example, a routine interview question,

In a party of N people, each person will shake her/his hand with each other person only once. On total how many hand-shakes would happen?

***Solution:***

It can be solved in different ways, graphs, recursion, etc. Let us see, how recursively it can be solved.

There are N persons. Each person shake-hand with each other only once. Considering N-th person, (s)he has to shake-hand with (N-1) persons. Now the problem reduced to small instance of (N-1) persons. Assuming $T_N$ as total shake-hands, it can be formulated recursively.

$T_N$ = (N-1) + $T_{N-1}$ [$T_1$ = 0, i.e. the last person have already shook-hand with every one]

Solving it recursively yields an arithmetic series, which can be evaluated to N(N-1)/2.

*Exercise: In a party of N couples, only one gender (either male or female) can shake hand with every one. How many shake-hands would happen?*

Usually recursive programs results in poor time complexities. An example is Fibonacci series. The time complexity of calculating n-th Fibonacci number using recursion is approximately $1.6^n$. It means the same computer takes almost 60% more time for next Fibonacci number. Recursive Fibonacci algorithm has overlapped subproblems. There are other techniques like *dynamic programming* to improve such overlapped algorithms.

However, few algorithms, (e.g. merge sort, quick sort, etc…) results in optimal time complexity using recursion.

**Base Case:**

One critical requirement of recursive functions is termination point or base case. Every recursive program must have base case to make sure that the function will terminate. Missing base case results in unexpected behaviour.

**Different Ways of Writing Recursive Functions in C/C++:**

***Function calling itself: (Direct way)***

Most of us aware atleast two different ways of writing recursive programs. Given below is towers of Hanoi code. It is an example of direct calling.

```
// Assuming n-th disk is bottom disk (count down)
void tower(int n, char sourcePole, char destinationPole, char auxiliaryPole)
{
   // Base case (termination condition)
   if(0 == n)
     return;

   // Move first n-1 disks from source pole
   // to auxiliary pole using destination as
   // temporary pole
   tower(n-1, sourcePole, auxiliaryPole,
      destinationPole);

// Move the remaining disk from source
   // pole to destination pole
   printf("Move the disk %d from %c to %c\n", n,
     sourcePole, destinationPole);

   // Move the n-1 disks from auxiliary (now source)
   // pole to destination pole using source pole as
   // temporary (auxiliary) pole
   tower(n-1, auxiliaryPole, destinationPole,
     sourcePole);
}

void main()
{
   tower(3, 'S', 'D', 'A');
}
```

The time complexity of TOH can be calculated by formulating number of moves.

We need to move the first N-1 disks from Source to Auxiliary and from Auxiliary to Destination, i.e. the first N-1 disks requires two moves. One more move of last disk from Source to Destination. Mathematically it can be defined recursively.

$M_N = 2M_{N-1} + 1$.

We can easily solve the above recursive relation ($2^N$-1), which is exponential.

***Recursion using mutual function call: (Indirect way)***

Indirect calling. Though least pratical, a function [funA()] can call another function [funB()] which inturn calls [funA()] former function. In this case both the functions should have the base case.

***Defensive Programming:***

We can combine defensive coding techniques with recursion for graceful functionality of application. Usually recursive programming is not allowed in safety critical applications, such as flight controls, health monitoring, etc. However, one can use a static count technique to avoid uncontrolled calls (NOT in safety critical systems, may be used in soft real time systems).

```
void recursive(int data)
{
   static callDepth;
   if(callDepth > MAX_DEPTH)
      return;

   // Increase call depth
   callDepth++;

   // do other processing
   recursive(data);

   // do other processing
   // Decrease call depth
   callDepth--;
}
```

callDepth depth depends on function stack frame size and maximum stack size.

***Recursion using function pointers: (Indirect way)***

Recursion can also implemented with function pointers. An example is signal handler in POSIX complaint systems. If the handler causes to trigger same event due to which the handler being called, the function will reenter.

*We will cover function pointer approach and iterative solution to TOH puzzle in later article.*

Thanks to Venki for writing the above post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

http://www.geeksforgeeks.org/recursive-functions/

# Chapter 2

# Tail Recursion

**What is tail recursion?**
A recursive function is tail recursive when recursive call is the last thing executed by the function. For example the following C++ function print() is tail recursive.

```
// An example of tail recursive function
void print(int n)
{
    if (n < 0)  return;
    cout << " " << n;

    // The last executed statement is recursive call
    print(n-1);
}
```

**Why do we care?**
The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by compiler. The idea used by compilers to optimize tail-recursive functions 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.

**Can a non-tail recursive function be written as tail-recursive to optimize it?**
Consider the following function to calculate factorial of n. It is a non-tail-recursive function. Although it looks like a tail recursive at first look. If we take a closer look, we can see that the value returned by fact(n-1) is used in fact(n), so the call to fact(n-1) is not the last thing done by fact(n)

```
#include<iostream>
using namespace std;

// A NON-tail-recursive function.  The function is not tail
// recursive because the value returned by fact(n-1) is used in
// fact(n) and call to fact(n-1) is not the last thing done by fact(n)
unsigned int fact(unsigned int n)
{
    if (n == 0) return 1;
    return n*fact(n-1);
}

// Driver program to test above function
int main()
{
    cout << fact(5);
    return 0;
}
```

The above function can be written as a tail recursive function. The idea is to use one more argument and accumulate the factorial value in second argument. When n reaches 0, return the accumulated value.

```
#include<iostream>
using namespace std;

// A tail recursive function to calculate factorial
unsigned factTR(unsigned int n, unsigned int a)
{
    if (n == 0)  return a;

    return factTR(n-1, n*a);
}

// A wrapper over factTR
unsigned int fact(unsigned int n)
{
   return factTR(n, 1);
}

// Driver program to test above function
int main()
{
    cout << fact(5);
    return 0;
}
```

We will soon be discussing more on tail recursion.

**References:**
http://en.wikipedia.org/wiki/Tail_call
http://c2.com/cgi/wiki?TailRecursion

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

http://www.geeksforgeeks.org/tail-recursion/

Category: Misc Tags: Recursion

# Chapter 3

# Check if a number is Palindrome

Given an integer, write a function that returns true if the given number is palindrome, else false. For example, 12321 is palindrome, but 1451 is not palindrome.

Let the given number be *num*. A simple method for this problem is to first reverse digits of *num*, then compare the reverse of *num* with *num*. If both are same, then return true, else false.

Following is an interesting method inspired from method#2 of thispost. The idea is to create a copy of *num* and recursively pass the copy by reference, and pass *num* by value. In the recursive calls, divide *num* by 10 while moving down the recursion tree. While moving up the recursion tree, divide the copy by 10. When they meet in a function for which all child calls are over, the last digit of *num* will be ith digit from the beginning and the last digit of copy will be ith digit from the end.

```
// A recursive C++ program to check whether a given number is
// palindrome or not
#include <stdio.h>

// A function that reurns true only if num contains one digit
int oneDigit(int num)
{
    // comparison operation is faster than division operation.
    // So using following instead of "return num / 10 == 0;"
    return (num >= 0 && num < 10);
}

// A recursive function to find out whether num is palindrome
// or not. Initially, dupNum contains address of a copy of num.
bool isPalUtil(int num, int* dupNum)
{
    // Base case (needed for recursion termination): This statement
    // mainly compares the first digit with the last digit
    if (oneDigit(num))
        return (num == (*dupNum) % 10);

    // This is the key line in this method. Note that all recursive
    // calls have a separate copy of num, but they all share same copy
    // of *dupNum. We divide num while moving up the recursion tree
    if (!isPalUtil(num/10, dupNum))
        return false;
    // The following statements are executed when we move up the
    // recursion call tree
    *dupNum /= 10;

    // At this point, if num%10 contains i'th digit from beiginning,
    // then (*dupNum)%10 contains i'th digit from end
    return (num % 10 == (*dupNum) % 10);
}

// The main function that uses recursive function isPalUtil() to
// find out whether num is palindrome or not
int isPal(int num)
{
    // If num is negative, make it positive
    if (num < 0)
       num = -num;

    // Create a separate copy of num, so that modifications made
    // to address dupNum don't change the input number.
    int *dupNum = new int(num); // *dupNum = num

    return isPalUtil(num, dupNum);
}

// Driver program to test above functions
int main()
{
    int n = 12321;
    isPal(n)? printf("Yes\n"): printf("No\n");

    n = 12;
    isPal(n)? printf("Yes\n"): printf("No\n");

    n = 88;
    isPal(n)? printf("Yes\n"): printf("No\n");

    n = 8999;
    isPal(n)? printf("Yes\n"): printf("No\n");
    return 0;
}
```

Output:

```
 Yes
No
Yes
No
```

This article is compiled byAashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

http://www.geeksforgeeks.org/check-if-a-number-is-palindrome/

Category: Misc Tags: Recursion

# Chapter 4

# Reverse a stack using recursion

You are not allowed to use loop constructs like while, for..etc, and you can only use the following ADT functions on Stack S:
isEmpty(S)
push(S)
pop(S)

**Solution:**
The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty. When the stack becomes empty, insert all held items one by one at the bottom of the stack.

For example, let the input stack be

```
    1

First 4 is inserted at the bottom.
    4
So we need a function that inserts at the bottom of a stack using the above given basic stack function. //Belo

void insertAtBottom(struct sNode** top_ref, int item)
{
   int temp;
   if(isEmpty(*top_ref))
   {
       push(top_ref, item);
   }
   else
   {

     /* Hold all items in Function Call Stack until we reach end of
       the stack. When the stack becomes empty, the isEmpty(*top_ref)
       becomes true, the above if part is executed and the item is
       inserted at the bottom */
     temp = pop(top_ref);
     insertAtBottom(top_ref, item);

     /* Once the item is inserted at the bottom, push all the
          items held in Function Call Stack */
     push(top_ref, temp);
   }
}
```

**//Below is the function that reverses the given stack using insertAtBottom()**

```
void reverse(struct sNode** top_ref)
{
  int temp;
  if(!isEmpty(*top_ref))
  {

    /* Hold all items in Function Call Stack until we reach end of
     the stack */
    temp = pop(top_ref);
    reverse(top_ref);

    /* Insert all the items (held in Function Call Stack) one by one
       from the bottom to top. Every item is inserted at the bottom */
    insertAtBottom(top_ref, temp);
  }
}
```

**//Below is a complete running program for testing above functions.**

```
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* structure of a stack node */
struct sNode
{
   char data;
   struct sNode *next;
};

/* Function Prototypes */
void push(struct sNode** top_ref, int new_data);
int pop(struct sNode** top_ref);
bool isEmpty(struct sNode* top);
void print(struct sNode* top);

/* Driveer program to test above functions */
int main()
{
  struct sNode *s = NULL;
  push(&s, 4);
  push(&s, 3);
  push(&s, 2);
  push(&s, 1);
  printf("\n Original Stack ");
  print(s);
  reverse(&s);
  printf("\n Reversed Stack ");
  print(s);
  getchar();
}

/* Function to check if the stack is empty */
bool isEmpty(struct sNode* top)
{
  return (top == NULL)? 1 : 0;
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
  /* allocate node */
  struct sNode* new_node =
            (struct sNode*) malloc(sizeof(struct sNode));

  if(new_node == NULL)
  {
     printf("Stack overflow \n");
     getchar();
     exit(0);
  }

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*top_ref);

  /* move the head to point to the new node */
  (*top_ref)    = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
  char res;
  struct sNode *top;

  /*If stack is empty then error */
  if(*top_ref == NULL)
  {
     printf("Stack overflow \n");
     getchar();
     exit(0);
  }
  else
  {
    top = *top_ref;
    res = top->data;
    *top_ref = top->next;
    free(top);
    return res;
  }
}

/* Functrion to pront a linked list */
void print(struct sNode* top)
{
  printf("\n");
  while(top != NULL)
  {
    printf(" %d ", top->data);
    top =  top->next;
  }
}
```

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem.

## Source

http://www.geeksforgeeks.org/reverse-a-stack-using-recursion/

Category: Misc Tags: Recursion

# Chapter 5

# Sort a stack using recursion

Given a stack, sort it using recursion. Use of any loop constructs like while, for..etc is not allowed. We can only use the following ADT functions on Stack S:

```
is_empty(S)  : Tests whether stack is empty or not.
push(S)      : Adds new element to the stack.
pop(S)       : Removes top element from the stack.
top(S)       : Returns value of the top element. Note that this
               function does not remove element from the stack.
```

Example:

```
Input:  -3
We strongly recommend you to minimize your browser and try this yourself first.

This problem is mainly a variant of Reverse stack using recursion.
The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty. When the s


Algorithm

We can use below algorithm to sort stack elements:

sortStack(stack S)
    if stack is not empty:
        temp = pop(S);
        sortStack(S);
        sortedInsert(S, temp);
```

Below algorithm is to insert element is sorted order:

```
sortedInsert(Stack S, element)
    if stack is empty OR element > top element
```

```
        push(S, elem)
    else
        temp = pop(S)
        sortedInsert(S, element)
        push(S, temp)
```

**Illustration:**

```
Let given stack be
-3
Let us illustrate sorting of stack using above example:
First pop all the elements from the stack and store poped element in variable 'temp'.  After poping all the el

temp = -3   --> stack frame #1
temp = 14   --> stack frame #2
temp = 18   --> stack frame #3
temp = -5   --> stack frame #4
temp = 30       --> stack frame #5
```

Now stack is empty and 'insert_in_sorted_order()' function is called and it inserts 30 (from stack frame #5) at the bottom of the stack. Now stack looks like below:

```
30
Now next element i.e. -5 (from stack frame #4) is picked. Since -5
30  -5
```

Next 18 (from stack frame #3) is picked. Since 18 30 **18** -5

Next 14 (from stack frame #2) is picked. Since 14 30 **14** -5

Now -3 (from stack frame #1) is picked, as -3 30 -3 -5

**Implementation:**
Below is C implementation of above algorithm.

```
// C program to sort a stack using recursion
#include <stdio.h>
#include <stdlib.h>

// Stack is represented using linked list
struct stack
{
    int data;
    struct stack *next;
};

// Utility function to initialize stack
```

```
void initStack(struct stack **s)
{
    *s = NULL;
}

// Utility function to chcek if stack is empty
int isEmpty(struct stack *s)
{
    if (s == NULL)
        return 1;
    return 0;
}

// Utility function to push an item to stack
void push(struct stack **s, int x)
{
    struct stack *p = (struct stack *)malloc(sizeof(*p));

    if (p == NULL)
    {
        fprintf(stderr, "Memory allocation failed.\n");
        return;
    }

    p->data = x;
    p->next = *s;
    *s = p;
}

// Utility function to remove an item from stack
int pop(struct stack **s)
{
    int x;
    struct stack *temp;

    x = (*s)->data;
    temp = *s;
    (*s) = (*s)->next;
    free(temp);

    return x;
}

// Function to find top item
int top(struct stack *s)
{
    return (s->data);
}

// Recursive function to insert an item x in sorted way
void sortedInsert(struct stack **s, int x)
{
    // Base case: Either stack is empty or newly inserted
    // item is greater than top (more than all existing)
    if (isEmpty(*s) || x > top(*s))
    {
        push(s, x);
        return;
    }

    // If top is greater, remove the top item and recur
    int temp = pop(s);
    sortedInsert(s, x);

    // Put back the top item removed earlier
    push(s, temp);
}

// Function to sort stack
void sortStack(struct stack **s)
{
    // If stack is not empty
    if (!isEmpty(*s))
    {
        // Remove the top item
        int x = pop(s);

        // Sort remaining stack
        sortStack(s);

        // Push the top item back in sorted stack
        sortedInsert(s, x);
    }
}

// Utility function to print contents of stack
void printStack(struct stack *s)
{
    while (s)
    {
        printf("%d ", s->data);
        s = s->next;
    }
    printf("\n");
}

// Driver Program
int main(void)
{
    struct stack *top;

    initStack(&top);
    push(&top, 30);
    push(&top, -5);
    push(&top, 18);
    push(&top, 14);
    push(&top, -3);
    printf("Stack elements before sorting:\n");
    printStack(top);

    sortStack(&top);
    printf("\n\n");

    printf("Stack elements after sorting:\n");
    printStack(top);

    return 0;
}
```

Output:

```
Stack elements before sorting:
-3 14 18 -5 30

Stack elements after sorting:
30 18 14 -3 -5
```

**Exercise:** Modify above code to reverse stack in descending order.

This article is contributed by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

http://www.geeksforgeeks.org/sort-a-stack-using-recursion/

Category: Misc Tags: Recursion, stack

Post navigation

← Hopcroft–Karp Algorithm for Maximum Matching | Set 2 (Implementation) Find the longest path in a matrix with given constraints →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

# Chapter 6

# Print a pattern without using any loop

Given a number n, print following pattern without using any loop.

```
Input: n = 16
Output: 16, 11, 6, 1, -4, 1, 6, 11, 16

Input: n = 10
Output: 10, 5, 0, 5, 10
```

We basically first reduce 5 one by one until we reach a negative or 0. After we reach 0 or negative, we one one add 5 until we reach n.

Source: Microsoft Interview Question.

**We strongly recommend you to minimize your browser and try this yourself first.**

The idea is to use recursion. It is an interesting question to try on your own.

Below is C++ Code. The code uses a flag variable to indicate whether we are moving toward 0 or we are moving toward back to n.

```
// C++ program to print pattern that first reduces 5 one
// by one, then adds 5. Without any loop
#include <iostream>
using namespace std;

// Recursive function to print the pattern.
// n indicates input value
// m indicates current value to be printed
// flag indicates whether we need to add 5 or
// subtract 5.  Initially flag is true.
void printPattern(int n, int m, bool flag)
{
    // Print m.
    cout << m << " ";
    // If we are moving back toward the n and
    // we have reached there, then we are done
    if (flag == false && n ==m)
        return;

    // If we are moving toward 0 or negative.
    if (flag)
    {
      // If m is greater, then 5, recur with true flag
      if (m-5 > 0)
         printPattern(n, m-5, true);
      else // recur with false flag
         printPattern(n, m-5, false);
    }
    else // If flag is false.
         printPattern(n, m+5, false);
}

// Driver Program
int main()
{
     int n = 16;
     printPattern(n, n, true);
    return 0;
}
```

Output:

```
 16, 11, 6, 1, -4, 1, 6, 11, 16
```

**How to print above pattern without any extra variable and loop?**
The above program works fine and prints the desired out, but uses extra variables. We can use two print statements. First one before recursive call that prints all decreasing sequence. Second one after the recursive call to print increasing sequence. Below is C++ implementation of the idea.

```
// C++ program to print pattern that first reduces 5 one
// by one, then adds 5. Without any loop an extra variable.
#include <iostream>
using namespace std;

// Recursive function to print the pattern without any extra
// variable
void printPattern(int n)
{
     // Base case (When n becomes 0 or negative)
     if (n ==0 || n<0)
     {
        cout << n << " ";
        return;
     }
     // First print decreasing order
     cout << n << " ";
     printPattern(n-5);

     // Then print increasing order
     cout << n << " ";
}

// Driver Program
int main()
{
     int n = 16;
     printPattern(n);
    return 0;
}
```

Output:

```
 16, 11, 6, 1, -4, 1, 6, 11, 16
```

Thanks to AKSHAY RATHORE for suggesting above solution.

This article is contributed by **Gautham**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

http://www.geeksforgeeks.org/print-a-pattern-without-using-any-loop/

Category: Misc Tags: Microsoft, Recursion

Post navigation

← Swiggy Interview Experience | Set 2 (On-Campus) 10 mistakes people tend to do in an Interview →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

# Chapter 7

# Minimum steps to reach a destination

Given a number line from -infinity to +infinity. You start at 0 and can go either to the left or to the right. The condition is that in i'th move, you take i steps.
a) Find if you can reach a given number x
b) Find the most optimal way to reach a given number x, if we can indeed reach it. For example, 3 can be reached om 2 steps, (0, 1) (1, 3) and 4 can be reached in 3 steps (0, -1), (-1, 1) (1, 4).

Source: Flipkart Interview Question

**We strongly recommend you to minimize your browser and try this yourself first.**

The important think to note is we can reach any destination as it is always possible to make a move of length 1. At any step i, we can move forward i, then backward i+1.

Below is a recursive solution suggested by Arpit Thapar here.

1) Since distance of +5 and -5 from 0 is same, hence we find answer for absolute value of destination.

2) We use a recursive function which takes as arguments:
i) Source Vertex
ii) Value of last step taken
iii) Destination

3) If at any point source vertex = destination; return number of steps.

4) Otherwise we can go in both of the possible directions. Take the minimum of steps in both cases.

From any vertex we can go to :
(current source + last step +1) and
(current source – last step -1)

5) If at any point, absolute value of our position exceeds the absolute value of our destination then it is intuitive that the shortest path is not possible from here. Hence we make the value of steps INT_MAX, so that when i take the minimum of both possibilities, this one gets eliminated.
If we don't use this last step, the program enters into an INFINITE recursion and gives RUN TIME ERROR.

Below is C++ implementation of above idea. Note that the solution only counts steps.

```
// C++ program to count number of steps to reach a point
#include<bits/stdc++.h>
using namespace std;

// Function to count number of steps required to reach a
// destination.
// source -> source vertex
// step -> value of last step taken
// dest -> destination vertex
int steps(int source, int step, int dest)
{
    // base cases
    if (abs(source) > (dest))  return INT_MAX;
    if (source == dest)     return step;

    // at each point we can go either way

    // if we go on positive side
    int pos = steps(source+step+1, step+1, dest);

    // if we go on negative side
    int neg = steps(source-step-1, step+1, dest);

    // minimum of both cases
    return min(pos, neg);
}

// Driver Program
int main()
{
    int dest = 11;
    cout << "No. of steps required to reach "
         << dest << " is " << steps(0, 0, dest);
    return 0;
}
```

Output:

```
 No. of steps required to reach 11 is 5
```

Thanks to Arpit Thapar for providing above algorithm and implementation.

This article is contributed by Abhay. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

http://www.geeksforgeeks.org/minimum-steps-to-reach-a-destination/

Chapter 8

# Given a string, print all possible palindromic partitions

Given a string, find all possible palindromic partitions of given string.

Example:

```
Input:  nitin
Output: n i t i n
        n iti n
        nitin

Input:  geeks
Output: g e e k s
        g ee k s
```

**We strongly recommend you to minimize your browser and try this yourself first.**

Note that this problem is different fromPalindrome Partitioning Problem, there the task was to find the partitioning with minimum cuts in input string. Here we need to print all possible partitions.

The idea is to go through every substring starting from first character, check if it is palindrome. If yes, then add the substring to solution and recur for remaining part. Below is complete algorithm.

Below is C++ implementation of above idea

```
// C++ program to print all palindromic partitions of a given string
#include<bits/stdc++.h>
using namespace std;

// A utility function to check if str is palindroem
bool isPalindrome(string str, int low, int high)
{
    while (low < high)
    {
        if (str[low] != str[high])
            return false;
        low++;
        high--;
    }
    return true;
}

// Recursive function to find all palindromic partitions of str[start..n-1]
// allPart --> A vector of vector of strings. Every vector inside it stores
//           a partition
// currPart --> A vector of strings to store current partition
void allPalPartUtil(vector<vector<string> >&allPart, vector<string> &currPart,
                   int start, int n, string str)
{
    // If 'start' has reached len
    if (start >= n)
    {
        allPart.push_back(currPart);
        return;
    }

    // Pick all possible ending points for substrings
    for (int i=start; i<n; i++)
    {
        // If substring str[start..i] is palindrome
        if (isPalindrome(str, start, i))
        {
            // Add the substring to result
            currPart.push_back(str.substr(start, i-start+1));

            // Recur for remaining remaining substring
            allPalPartUtil(allPart, currPart, i+1, n, str);

            // Remove substring str[start..i] from current
            // partition
            currPart.pop_back();
        }
    }
}

// Function to print all possible palindromic partitions of
// str. It mainly creates vectors and calls allPalPartUtil()
void allPalPartitions(string str)
{
    int n = str.length();

    // To Store all palindromic partitions
    vector<vector<string> > allPart;

    // To store current palindromic partition
    vector<string> currPart;

    // Call recursive function to generate all partiions
    // and store in allPart
    allPalPartUtil(allPart, currPart, 0, n, str);

    // Print all partitions generated by above call
    for (int i=0; i< allPart.size(); i++ )
    {
        for (int j=0; j<allPart[i].size(); j++)
            cout << allPart[i][j] << " ";
        cout << "\n";
    }
}

// Driver program
int main()
{
    string str = "nitin";
    allPalPartitions(str);
    return 0;
}
```

Output:

```
n i t i n
n iti n
nitin
```

This article is contributed by Ekta Goel. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

http://www.geeksforgeeks.org/given-a-string-print-all-possible-palindromic-partition/

Category: Strings Tags: Recursion

Chapter 9

# Generate all possible sorted arrays from alternate elements of two given sorted arrays

Given two sorted arrays A and B, generate all possible arrays such that first element is taken from A then from B then from A and so on in increasing order till the arrays exhausted. The generated arrays should end with an element from B.

```
For Example
A = {10, 15, 25}
B = {1, 5, 20, 30}

The resulting arrays are:
  10 20
  10 20 25 30
  10 30
  15 20
  15 20 25 30
  15 30
  25 30
```

**Source:** Microsoft Interview Question

**We strongly recommend you to minimize your browser and try this yourself first.**

The idea is to use recursion. In the recursive function, a flag is passed to indicate whether current element in output should be taken from 'A' or 'B'. Below is C++ implementation.

```
#include<bits/stdc++.h>
using namespace std;

void printArr(int arr[], int n);

/* Function to generates and prints all sorted arrays from alternate elements of
   'A[i..m-1]' and 'B[j..n-1]'
   If 'flag' is true, then current element is to be included from A otherwise
   from B.
   'len' is the index in output array C[]. We print output  array  each time
   before including a character from A only if length of output array is
   greater than 0. We try than all possible combinations */
void generateUtil(int A[], int B[], int C[], int i, int j, int m, int n,
                  int len, bool flag)
{
    if (flag) // Include valid element from A
    {
        // Print output if there is at least one 'B' in output array 'C'
        if (len)
            printArr(C, len+1);

        // Recur for all elements of A after current index
        for (int k = i; k < m; k++)
        {
            if (!len)
            {
                /* this block works for the very first call to include
                    the first element in the output array */
                C[len] = A[k];

                // don't increment lem as B is included yet
                generateUtil(A, B, C, k+1, j, m, n, len, !flag);
            }
            else    /* include valid element from A and recur */
            {
                if (A[k] > C[len])
                {
                    C[len+1] = A[k];
                    generateUtil(A, B, C, k+1, j, m, n, len+1, !flag);
                }
            }
        }
    }
    else   /* Include valid element from B and recur */
    {
        for (int l = j; l < n; l++)
        {
            if (B[l] > C[len])
            {
                C[len+1] = B[l];
                generateUtil(A, B, C, i, l+1, m, n, len+1, !flag);
            }
        }
    }
}

/* Wrapper function */
void generate(int A[], int B[], int m, int n)
{
    int C[m+n]; /* output array */
    generateUtil(A, B, C, 0, 0, m, n, 0, true);
}

// A utility function to print an array
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program
int main()
{
    int A[] = {10, 15, 25};
    int B[] = {5, 20, 30};
    int n = sizeof(A)/sizeof(A[0]);
    int m = sizeof(B)/sizeof(B[0]);
    generate(A, B, n, m);
    return 0;
}
```

Output:

```
 10 20
10 20 25 30
10 30
15 20
15 20 25 30
15 30
25 30
```

This article is contributed by Gaurav Ahirwar. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above\

## Source

http://www.geeksforgeeks.org/generate-all-possible-sorted-arrays-from-alternate-elements-of-two-given-arrays/

Category: Arrays Tags: Recursion

Post navigation

← SAP Labs Interview Questions | Set 9 (Fresher) Flipkart Interview Experience | Set 30(For SDE 2) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

# Chapter 10

# Print all non-increasing sequences of sum equal to a given number x

Given a number x, print all possible non-increasing sequences with sum equals to x.

Examples:

```
Input: x = 3
Output: 1 1 1
        2 1
        3

Input: x = 4
Output: 1 1 1 1
        2 1 1
        2 2
        3 1
        4
```

**We strongly recommend you to minimize your browser and try this yourself first.**

The idea is to use a recursive function, an array arr[] to store all sequences one by one and an index variable curr_idx to store current next index in arr[]. Below is algorithm.

1) If current sum is equal to x, then print current sequence.
2) Place all possible numbers from 1 to x-curr_sum numbers at curr_idx in array. Here curr_sum is sum of current elements in arr[]. After placing a number, recur for curr_sum + number and curr_idx+1,

Below is C++ implementation of above steps.

```
// C++ program to generate all non-increasing sequences
// of sum equals to x
#include<bits/stdc++.h>
using namespace std;

// Utility function to print array arr[0..n-1]
void printArr(int arr[], int n)
{
    for (int i=0; i<n; i++)
      cout << arr[i] << " ";
    cout << endl;
}

//  Recursive Function to generate all non-increasing sequences
// with sum x
// arr[]    --> Elements of current sequence
// curr_sum --> Current Sum
// curr_idx --> Current index in arr[]
void generateUtil(int x, int arr[], int curr_sum, int curr_idx)
{
   // If current sum is equal to x, then we found a sequence
   if (curr_sum == x)
   {
      printArr(arr, curr_idx);
      return;
   }

   // Try placing all numbers from 1 to x-curr_sum at current index
   int num = 1;

   // The placed number must also be smaller than previously placed
   // numbers, i.e., arr[curr_idx-1] if there exists a pevious number
   while (num<=x-curr_sum && (curr_idx==0 || num<=arr[curr_idx-1]))
   {
       // Place number at curr_idx
       arr[curr_idx] = num;

       // Recur
       generateUtil(x, arr, curr_sum+num, curr_idx+1);

       // Try next number
       num++;
   }
}

// A wrapper over generateUtil()
void generate(int x)
{
    int arr[x]; // Array to store sequences on by one
    generateUtil(x, arr, 0, 0);
}

// Driver program
int main()
{
    int x = 5;
    generate(x);
    return 0;
}
```

Output:

```
 1 1 1 1 1
2 1 1 1
2 2 1
3 1 1
3 2
4 1
5
```

This article is contributed by **Ashish Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

http://www.geeksforgeeks.org/print-all-non-increasing-sequences-of-sum-equal-to-a-given-number/

Category: Misc Tags: combionatrics, Recursion

# Chapter 11

# Print all increasing sequences of length k from first n natural numbers

Given two positive integers n and k, print all increasing sequences of length k such that the elements in every sequence are from first n natural numbers.

Examples:

```
Input: k = 2, n = 3
Output: 1 2
        1 3
        2 3

Input: k = 5, n = 5
Output: 1 2 3 4 5

Input: k = 3, n = 5
Output: 1 2 3
        1 2 4
        1 2 5
        1 3 4
        1 3 5
        1 4 5
        2 3 4
        2 3 5
        2 4 5
        3 4 5
```

**We strongly recommend to minimize the browser and try this yourself first.**

It's a good recursion question. The idea is to create an array of length k. The array stores current sequence. For every position in array, we check the previous element and one by one put all elements greater than the previous element. If there is no previous element (first position), we put all numbers from 1 to n.

Following is C++ implementation of above idea.

```
// C++ program to  print all increasing sequences of
// length 'k' such that the elements in every sequence
// are from first 'n' natural numbers.
#include<iostream>
using namespace std;

// A utility function to print contents of arr[0..k-1]
void printArr(int arr[], int k)
{
    for (int i=0; i<k; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// A recursive function to print all increasing sequences
// of first n natural numbers.  Every sequence should be
// length k. The array arr[] is used to store current
// sequence.
void printSeqUtil(int n, int k, int &len, int arr[])
{
    // If length of current increasing sequence becomes k,
    // print it
    if (len == k)
    {
        printArr(arr, k);
        return;
    }

    // Decide the starting number to put at current position:
    // If length is 0, then there are no previous elements
    // in arr[].  So start putting new numbers with 1.
    // If length is not 0, then start from value of
    // previous element plus 1.
    int i = (len == 0)? 1 : arr[len-1] + 1;

    // Increase length
    len++;

    // Put all numbers (which are greater than the previous
    // element) at new position.
    while (i<=n)
    {
        arr[len-1] = i;
        printSeqUtil(n, k, len, arr);
        i++;
    }

    // This is important. The variable 'len' is shared among
    // all function calls in recursion tree. Its value must be
    // brought back before next iteration of while loop
    len--;
}

// This function prints all increasing sequences of
// first n natural numbers. The length of every sequence
// must be k.  This function mainly uses printSeqUtil()
void printSeq(int n, int k)
{
    int arr[k];  // An array to store individual sequences
    int len = 0; // Initial length of current sequence
    printSeqUtil(n, k, len, arr);
}

// Driver program to test above functions
int main()
{
    int k = 3, n = 7;
    printSeq(n, k);
    return 0;
}
```

Output:

```
 1 2 3
1 2 4
1 2 5
1 2 6
1 2 7
1 3 4
1 3 5
1 3 6
1 3 7
1 4 5
1 4 6
1 4 7
1 5 6
1 5 7
1 6 7
2 3 4
2 3 5
2 3 6
2 3 7
2 4 5
2 4 6
2 4 7
2 5 6
2 5 7
2 6 7
3 4 5
3 4 6
3 4 7
3 5 6
3 5 7
3 6 7
4 5 6
4 5 7
4 6 7
5 6 7
```

This article is contributed by **Arjun**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

http://www.geeksforgeeks.org/print-increasing-sequences-length-k-first-n-natural-numbers/

Category: Misc Tags: Recursion

Chapter 12

# Print all possible strings of length k that can be formed from a set of n characters

Given a set of characters and a positive integer k, print all possible strings of length k that can be formed from the given set.

Examples:

```
Input:
set[] = {'a', 'b'}, k = 3

Output:
aaa
aab
aba
abb
baa
bab
bba
bbb


Input:
set[] = {'a', 'b', 'c', 'd'}, k = 1
Output:
a
b
c
d
```

For a given set of size n, there will be n^k possible strings of length k. The idea is to start from an empty output string (we call it *prefix* in following code). One by one add all characters to *prefix*. For every character added, print all possible strings with current prefix by recursively calling for k equals to k-1.
Following is Java implementation for same.

```
// Java program to print all possible strings of length k
class PrintAllKLengthStrings {

    // Driver method to test below methods
    public static void main(String[] args) {
        System.out.println("First Test");
        char set1[] = {'a', 'b'};
        int k = 3;
        printAllKLength(set1, k);

        System.out.println("\nSecond Test");
        char set2[] = {'a', 'b', 'c', 'd'};
        k = 1;
        printAllKLength(set2, k);
    }

    // The method that prints all possible strings of length k.  It is
    //  mainly a wrapper over recursive function printAllKLengthRec()
    static void printAllKLength(char set[], int k) {
        int n = set.length;
        printAllKLengthRec(set, "", n, k);
    }

    // The main recursive method to print all possible strings of length k
    static void printAllKLengthRec(char set[], String prefix, int n, int k) {

        // Base case: k is 0, print prefix
        if (k == 0) {
            System.out.println(prefix);
            return;
        }

        // One by one add all characters from set and recursively
        // call for k equals to k-1
        for (int i = 0; i < n; ++i) {

            // Next character of input added
            String newPrefix = prefix + set[i];

            // k is decreased, because we have added a new character
            printAllKLengthRec(set, newPrefix, n, k - 1);
        }
    }
}
```

Output:

```
 First Test
aaa
aab
aba
abb
```

```
baa
bab
bba
bbb

Second Test
a
b
c
d
```

The above solution is mainly generalization of this post.

This article is contriibuted by **Abhinav Ramana**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

http://www.geeksforgeeks.org/print-all-combinations-of-given-length/

Category: Misc Tags: combionatrics, Java, Recursion

Post navigation

← QuickSort on Singly Linked List Convert a given Binary Tree to Doubly Linked List | Set 2 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 13

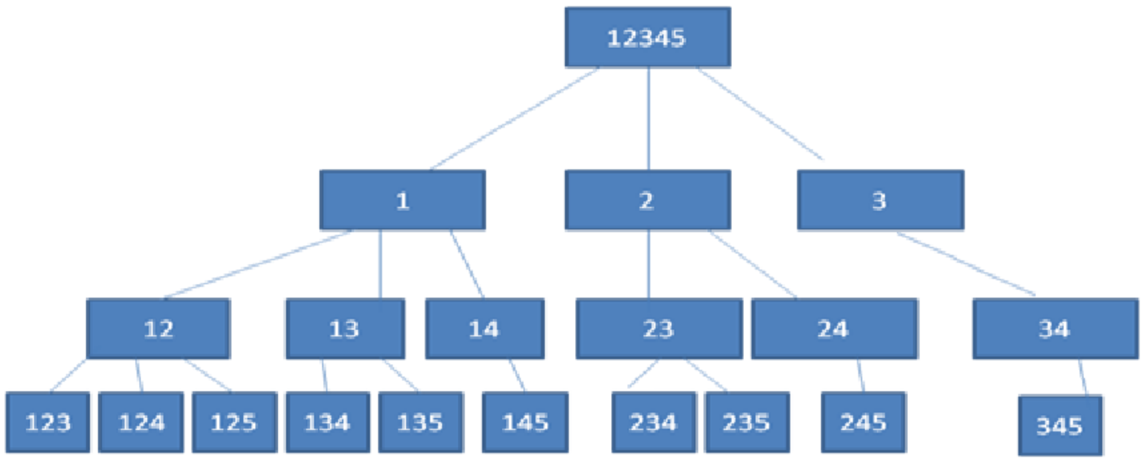
# Print all possible combinations of r elements in a given array of size n

Given an array of size n, generate and print all possible combinations of r elements in array. For example, if input array is {1, 2, 3, 4} and r is 2, then output should be {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4} and {3, 4}.

Following are two methods to do this.

**Method 1 (Fix Elements and Recur)**
We create a temporary array 'data[]' which stores all outputs one by one. The idea is to start from first index (index = 0) in data[], one by one fix elements at this index and recur for remaining indexes. Let the input array be {1, 2, 3, 4, 5} and r be 3. We first fix 1 at index 0 in data[], then recur for remaining indexes, then we fix 2 at index 0 and recur. Finally, we fix 3 and recur for remaining indexes. When number of elements in data[] becomes equal to r (size of a combination), we print data[].

Following diagram shows recursion tree for same input.



Following is C++ implementation of above approach.

```
// Program to print all combination of size r in an array of size n
#include <stdio.h>
void combinationUtil(int arr[], int data[], int start, int end, int index, int r);
```

```
// The main function that prints all combinations of size r
// in arr[] of size n. This function mainly uses combinationUtil()
void printCombination(int arr[], int n, int r)
{
    // A temporary array to store all combination one by one
    int data[r];

    // Print all combination using temprary array 'data[]'
    combinationUtil(arr, data, 0, n-1, 0, r);
}

/* arr[]  ---> Input Array
   data[] ---> Temporary array to store current combination
   start & end ---> Staring and Ending indexes in arr[]
   index  ---> Current index in data[]
   r ---> Size of a combination to be printed */
void combinationUtil(int arr[], int data[], int start, int end, int index, int r)
{
    // Current combination is ready to be printed, print it
    if (index == r)
    {
        for (int j=0; j<r; j++)
            printf("%d ", data[j]);
        printf("\n");
        return;
    }

    // replace index with all possible elements. The condition
    // "end-i+1 >= r-index" makes sure that including one element
    // at index will make a combination with remaining elements
    // at remaining positions
    for (int i=start; i<=end && end-i+1 >= r-index; i++)
    {
        data[index] = arr[i];
        combinationUtil(arr, data, i+1, end, index+1, r);
    }
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int r = 3;
    int n = sizeof(arr)/sizeof(arr[0]);
    printCombination(arr, n, r);
}
```

Output:

```
 1 2 3
1 2 4
1 2 5
1 3 4
```

```
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

*How to handle duplicates?*
Note that the above method doesn't handle duplicates. For example, if input array is {1, 2, 1} and r is 2, then the program prints {1, 2} and {2, 1} as two different combinations. We can avoid duplicates by adding following two additional things to above code.
1) Add code to sort the array before calling combinationUtil() in printCombination()
2) Add following lines at the end of for loop in combinationUtil()

```
        // Since the elements are sorted, all occurrences of an element
        // must be together
        while (arr[i] == arr[i+1])
             i++;
```

See **this**for an implementation that handles duplicates.

**Method 2 (Include and Exclude every element)**
Like the above method, We create a temporary array data[]. The idea here is similar to Subset Sum Problem. We one by one consider every element of input array, and recur for two cases:

1) The element is included in current combination (We put the element in data[] and increment next available index in data[])
2) The element is excluded in current combination (We do not put the element and do not change index)

When number of elements in data[] become equal to r (size of a combination), we print it.

This method is mainly based on Pascal's Identity, i.e. $\mathbf{n_{c_r}} = \mathbf{n\text{-}1_{c_r}} + \mathbf{n\text{-}1_{c_{r\text{-}1}}}$

Following is C++ implementation of method 2.

```
// Program to print all combination of size r in an array of size n
#include<stdio.h>
void combinationUtil(int arr[],int n,int r,int index,int data[],int i);

// The main function that prints all combinations of size r
// in arr[] of size n. This function mainly uses combinationUtil()
void printCombination(int arr[], int n, int r)
{
    // A temporary array to store all combination one by one
    int data[r];

    // Print all combination using temprary array 'data[]'
    combinationUtil(arr, n, r, 0, data, 0);
}

/* arr[]  ---> Input Array
   n      ---> Size of input array
   r      ---> Size of a combination to be printed
   index  ---> Current index in data[]
   data[] ---> Temporary array to store current combination
   i      ---> index of current element in arr[]     */
void combinationUtil(int arr[], int n, int r, int index, int data[], int i)
{
    // Current cobination is ready, print it
    if (index == r)
    {
        for (int j=0; j<r; j++)
            printf("%d ",data[j]);
        printf("\n");
        return;
    }

    // When no more elements are there to put in data[]
    if (i >= n)
        return;

    // current is included, put next at next location
    data[index] = arr[i];
    combinationUtil(arr, n, r, index+1, data, i+1);

    // current is excluded, replace it with next (Note that
    // i+1 is passed, but index is not changed)
    combinationUtil(arr, n, r, index, data, i+1);
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int r = 3;
    int n = sizeof(arr)/sizeof(arr[0]);
    printCombination(arr, n, r);
    return 0;
}
```

Output:

```
 1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

*How to handle duplicates in method 2?*
Like method 1, we can following two things to handle duplicates.
1) Add code to sort the array before calling combinationUtil() in printCombination()
2) Add following lines between two recursive calls of combinationUtil() in combinationUtil()

```
        // Since the elements are sorted, all occurrences of an element
        // must be together
        while (arr[i] == arr[i+1])
             i++;
```

See **this**for an implementation that handles duplicates.

This article is contributed by **Bateesh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

http://www.geeksforgeeks.org/print-all-possible-combinations-of-r-elements-in-a-given-array-of-size-n/

Category: Arrays Tags: MathematicalAlgo, Recursion

Post navigation

← Fab.com Pune Interview Experience Goldman Sachs Interview Experience →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 14

# Practice questions for Linked List and Recursion

Assume the structure of a Linked List node is as follows.

```
struct node
{
  int data;
  struct node *next;
};
```

Explain the functionality of following C functions.

**1. What does the following function do for a given Linked List?**

```
void fun1(struct node* head)
{
  if(head == NULL)
    return;

  fun1(head->next);
  printf("%d  ", head->data);
}
```

fun1() prints the given Linked List in reverse manner. For Linked List 1->2->3->4->5, fun1() prints 5->4->3->2->1.

**2. What does the following function do for a given Linked List ?**

```
void fun2(struct node* head)
{
  if(head== NULL)
    return;
  printf("%d  ", head->data);

  if(head->next != NULL )
    fun2(head->next->next);
  printf("%d  ", head->data);
}
```

fun2() prints alternate nodes of the given Linked List, first from head to end, and then from end to head. If Linked List has even number of nodes, then fun2() skips the last node. For Linked List 1->2->3->4->5, fun2() prints 1 3 5 5 3 1. For Linked List 1->2->3->4->5->6, fun2() prints 1 3 5 5 3 1.

Below is a complete running program to test above functions.

```
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct node
{
  int data;
  struct node *next;
};


/* Prints a linked list in reverse manner */
void fun1(struct node* head)
{
  if(head == NULL)
    return;

  fun1(head->next);
  printf("%d  ", head->data);
}

/* prints alternate nodes of a Linked List, first
  from head to end, and then from end to head. */
void fun2(struct node* start)
{
  if(start == NULL)
    return;
  printf("%d  ", start->data);

  if(start->next != NULL )
    fun2(start->next->next);
  printf("%d  ", start->data);
}

/* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
```

```
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Drier program to test above functions */
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  /* Using push() to construct below list
    1->2->3->4->5  */
  push(&head, 5);
  push(&head, 4);
  push(&head, 3);
  push(&head, 2);
  push(&head, 1);

  printf("\n Output of fun1() for list 1->2->3->4->5 \n");
  fun1(head);

  printf("\n Output of fun2() for list 1->2->3->4->5 \n");
  fun2(head);

  getchar();
  return 0;
}
```

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

## Source

http://www.geeksforgeeks.org/practice-questions-for-linked-list-and-recursion/

Category: Linked Lists Tags: Recursion

Chapter 15

# Practice Questions for Recursion | Set 1

Explain the functionality of following functions.

**Question 1**

```
int fun1(int x, int y)
{
  if(x == 0)
    return y;
  else
    return fun1(x - 1,  x + y);
}
```

Answer: The function fun() calculates and returns ((1 + 2 … + x-1 + x) +y) which is x(x+1)/2 + y. For example if x is 5 and y is 2, then fun should return 15 + 2 = 17.

**Question 2**

```
void fun2(int arr[], int start_index, int end_index)
{
  if(start_index >= end_index)
     return;
  int min_index;
  int temp;

  /* Assume that minIndex() returns index of minimum value in
    array arr[start_index...end_index] */
  min_index = minIndex(arr, start_index, end_index);

  temp = arr[start_index];
  arr[start_index] = arr[min_index];
  arr[min_index] = temp;

  fun2(arr, start_index + 1, end_index);
}
```

Answer: The function fun2() is a recursive implementation of Selection Sort.

Please write comments if you find any of the answers/codes incorrect, or you want to share more information about the topics discussed above.

## Source

http://www.geeksforgeeks.org/practice-questions-for-recursion/

Category: Misc Tags: Recursion

# Chapter 16

# Practice Questions for Recursion | Set 2

Explain the functionality of following functions.

**Question 1**

```
/* Assume that n is greater than or equal to 1 */
int fun1(int n)
{
  if(n == 1)
     return 0;
  else
     return 1 + fun1(n/2);
}
```

Answer: The function calculates and returns $\lfloor \log_2(n) \rfloor$. For example, if n is between 8 and 15 then fun1() returns 3. If n is between 16 to 31 then fun1() returns 4.

**Question 2**

```
/* Assume that n is greater than or equal to 0 */
void fun2(int n)
{
  if(n == 0)
    return;

  fun2(n/2);
  printf("%d", n%2);
}
```

Answer: The function fun2() prints binary equivalent of n. For example, if n is 21 then fun2() prints 10101.

Note that above functions are just for practicing recursion, they are not the ideal implementation of the functionality they provide.

Please write comments if you find any of the answers/codes incorrect.

## Source

http://www.geeksforgeeks.org/practice-questions-for-recursion-set-2/

Category: Misc Tags: Recursion

# Chapter 17

# Practice Questions for Recursion | Set 3

Explain the functionality of below recursive functions.

**Question 1**

```
void fun1(int n)
{
   int i = 0;
   if (n > 1)
     fun1(n-1);
   for (i = 0; i < n; i++)
     printf(" * ");
}
```

Answer: Total numbers of stars printed is equal to 1 + 2 + …. (n-2) + (n-1) + n, which is n(n+1)/2.

**Question 2**

```
#define LIMIT 1000
void fun2(int n)
{
  if (n <= 0)
     return;
  if (n > LIMIT)
    return;
  printf("%d ", n);
  fun2(2*n);
  printf("%d ", n);
}
```

Answer: For a positive n, fun2(n) prints the values of n, 2n, 4n, 8n … while the value is smaller than LIMIT. After printing values in increasing order, it prints same numbers again in reverse order. For example

fun2(100) prints 100, 200, 400, 800, 800, 400, 200, 100.
If n is negative, then it becomes a non-terminating recursion and causes overflow.

Please write comments if you find any of the answers/codes incorrect, or you want to share more information about the topics discussed above.

## Source

http://www.geeksforgeeks.org/practice-questions-for-recursion-set-3/

Category: Misc Tags: Recursion

Chapter 18

# Practice Questions for Recursion | Set 4

**Question 1**
Predict the output of following program.

```
#include<stdio.h>
void fun(int x)
{
  if(x > 0)
  {
     fun(--x);
     printf("%d\t", x);
     fun(--x);
  }
}

int main()
{
  int a = 4;
  fun(a);
  getchar();
  return 0;
}
```

Output: 0 1 2 0 3 0 1

```
                   fun(4);
                   /
                fun(3), print(3), fun(2)(prints 0 1)
               /
           fun(2), print(2), fun(1)(prints 0)
           /
       fun(1), print(1), fun(0)(does nothing)
       /
```

```
    fun(0), print(0), fun(-1) (does nothing)
```

**Question 2**
Predict the output of following program. What does the following fun() do in general?

```
int fun(int a[],int n)
{
  int x;
  if(n == 1)
    return a[0];
  else
    x = fun(a, n-1);
  if(x > a[n-1])
    return x;
  else
    return a[n-1];
}

int main()
{
  int arr[] = {12, 10, 30, 50, 100};
  printf(" %d ", fun(arr, 5));
  getchar();
  return 0;
}
```

Output: 100
fun() returns the maximum value in the input array a[] of size n.

**Question 3**
Predict the output of following program. What does the following fun() do in general?

```
int fun(int i)
{
  if ( i%2 ) return (i++);
  else return fun(fun( i - 1 ));
}

int main()
{
  printf(" %d ", fun(200));
  getchar();
  return 0;
}
```

Output: 199
If n is odd then returns n, else returns (n-1). Eg., for n = 12, you get 11 and for n = 11 you get 11. The statement *"return i++;"* returns value of i only as it is a post increment.

Please write comments if you find any of the answers/codes incorrect, or you want to share more information/questions about the topics discussed above.

## Source

http://www.geeksforgeeks.org/practice-questions-for-recursion-set-4/

Category: Misc Tags: Recursion

# Chapter 19

# Practice Questions for Recursion | Set 5

**Question 1**

Predict the output of following program. What does the following fun() do in general?

```
#include<stdio.h>

int fun(int a, int b)
{
   if (b == 0)
       return 0;
   if (b % 2 == 0)
       return fun(a+a, b/2);

   return fun(a+a, b/2) + a;
}

int main()
{
  printf("%d", fun(4, 3));
  getchar();
  return 0;
}
```

Output: 12

It calulates a*b (a multipied b).

**Question 2**
In question 1, if we replace + with * and replace return 0 with return 1, then what does the changed function do? Following is the changed function.

```
 #include<stdio.h>

int fun(int a, int b)
```

```
{
   if (b == 0)
       return 1;
   if (b % 2 == 0)
       return fun(a*a, b/2);

   return fun(a*a, b/2)*a;
}

int main()
{
  printf("%d", fun(4, 3));
  getchar();
  return 0;
}
```

Output: 64
It calulates a^b (a raised to power b).

**Question 3**
Predict the output of following program. What does the following fun() do in general?

```
 #include<stdio.h>

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

int main()
{
  printf(" %d ", fun(99));
  getchar();
  return 0;
}
```

Output: 91

```
fun(99) = fun(fun(110)) since 99 ? 100
           = fun(100)    since 110 > 100
           = fun(fun(111)) since 100 ? 100
           = fun(101)    since 111 > 100
           = 91        since 101 > 100
```

Returned value of fun() is 91 for all integer rguments n 101. This function is known as McCarthy 91 function.

Please write comments if you find any of the answers/codes incorrect, or you want to share more information/questions about the topics discussed above.

## Source

http://www.geeksforgeeks.org/practice-questions-for-recursion-set-5/

Category: Misc Tags: Recursion

Chapter 20

# Practice Questions for Recursion | Set 6

**Question 1**
Consider the following recursive C function. Let *len* be the length of the string s and *num* be the number of characters printed on the screen, give the relation between *num* and *len* where *len* is always greater than 0.

```
void abc(char *s)
{
    if(s[0] == '\0')
        return;

    abc(s + 1);
    abc(s + 1);
    printf("%c", s[0]);
}
```

Following is the relation between *num* and *len*.

```
  num = 2^len-1
```

```
s[0] is 1 time printed
s[1] is 2 times printed
s[2] is 4 times printed
s[i] is printed 2^i times
s[strlen(s)-1] is printed 2^(strlen(s)-1) times
total = 1+2+....+2^(strlen(s)-1)
      = (2^strlen(s)) - 1
```

For example, the following program prints 7 characters.

```
#include<stdio.h>
```

```
void abc(char *s)
{
    if(s[0] == '\0')
        return;

    abc(s + 1);
    abc(s + 1);
    printf("%c", s[0]);
}

int main()
{
    abc("xyz");
    return 0;
}
```

Thanks to bharat nag for suggesting this solution.

**Question 2**

```
#include<stdio.h>
int fun(int count)
{
    printf("%d\n", count);
    if(count < 3)
    {
      fun(fun(fun(++count)));
    }
    return count;
}

int main()
{
    fun(1);
    return 0;
}
```

Output:

```
 1
 2
 3
 3
 3
 3
 3
```

The main() function calls fun(1). fun(1) prints “1” and calls fun(fun(fun(2))). fun(2) prints “2” and calls fun(fun(fun(3))). So the function call sequence becomes fun(fun(fun(fun(fun(3))))). fun(3) prints “3” and returns 3 (note that count is not incremented and no more functions are called as the if condition is not true for count 3). So the function call sequence reduces to fun(fun(fun(fun(3)))). fun(3) again prints “3” and returns 3. So the function call again reduces to fun(fun(fun(3))) which again prints “3” and reduces to fun(fun(3)). This continues and we get “3” printed 5 times on the screen.

Please write comments if you find any of the answers/codes incorrect, or you want to share more information/questions about the topics discussed above.

## Source

http://www.geeksforgeeks.org/practice-questions-for-recursion-set-6/

Category: Misc Tags: Recursion

Chapter 21

# Practice Questions for Recursion | Set 7

**Question 1** Predict the output of the following program. What does the following fun() do in general?

```
#include <stdio.h>

int fun ( int n, int *fp )
{
    int t, f;

    if ( n <= 1 )
    {
        *fp = 1;
        return 1;
    }
    t = fun ( n-1, fp );
    f = t + *fp;
    *fp = t;
    return f;
}

int main()
{
    int x = 15;
    printf("%d\n",fun(5, &x));

    return 0;
}
```

Output:

```
 8
```

The program calculates nth Fibonacci Number. The statement t = fun ( n-1, fp ) gives the (n-1)th Fibonacci number and *fp is used to store the (n-2)th Fibonacci Number. Initial value of *fp (which is 15 in the above prgram) doesn't matter. Following recursion tree shows all steps from 1 to 10, for exceution of fun(5, &x).

```
                              (1) fun(5, fp)
                              /           \
                         (2) fun(4, fp)  (10) t = 5, f = 8, *fp = 5
                         /          \
                   (3) fun(3, fp)  (9) t = 3, f = 5, *fp = 3
                  /            \
           (4) fun(2, fp)      (8) t = 2, f = 3, *fp = 2
          /          \
   (5) fun(1, fp)   (7) t = 1, f = 2, *fp = 1
   /
(6) *fp = 1
```

**Question 2:** Predict the output of the following program.

```
#include <stdio.h>

void fun(int n)
{
    if(n > 0)
    {
        fun(n-1);
        printf("%d ", n);
        fun(n-1);
    }
}

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

Output

```
 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

```
                     fun(4)
                   /
                fun(3), print(4), fun(3) [fun(3) prints 1 2 1 3 1 2 1]
               /
           fun(2), print(3), fun(2) [fun(2) prints 1 2 1]
           /
       fun(1), print(2), fun(1) [fun(1) prints 1]
       /
    fun(0), print(1), fun(0) [fun(0) does nothing]
```

Please write comments if you find any of the answers/codes incorrect, or you want to share more information/questions about the topics discussed above.

## Source

http://www.geeksforgeeks.org/practice-questions-for-recursion-set-7/

Category: Misc Tags: Recursion