Contents

[COMPETITIVE PROGRAMMING 3](#_Toc91617554)

[CODING INTERVIEW 3](#_Toc91617555)

[DATA TYPES (Primitive) 3](#_Toc91617556)

[Static Array 5](#_Toc91617557)

[Dynamic Array 5](#_Toc91617558)

[C++ 5](#_Toc91617559)

[JAVA 5](#_Toc91617560)

[C# 5](#_Toc91617561)

[PYTHON 5](#_Toc91617562)

[In C: 5](#_Toc91617563)

[How it is implemented internally? 6](#_Toc91617564)

[Recursion: 6](#_Toc91617565)

[Trace the below code for x = 3 , n = 3 7](#_Toc91617566)

[Trace the below code: 7](#_Toc91617567)

[Trace the below code: 8](#_Toc91617568)

[WAF for calculating the factorial of a number ‘n’ using Tail recursion. 8](#_Toc91617569)

[What is tail recursion? 8](#_Toc91617570)

[Why do we care? 8](#_Toc91617571)

[Tail Call Elimination 9](#_Toc91617572)

[WAP to find the max element in an Array 9](#_Toc91617573)

[Approach 1: Using loop 9](#_Toc91617574)

[Approach 2: Using recursion 10](#_Toc91617575)

[Time Complexity 10](#_Toc91617576)

[Steps in Solving a problem 10](#_Toc91617577)

[Time Complexity Analysis 11](#_Toc91617578)

[Let’s Understand: 11](#_Toc91617579)

[Categories of Time and Space Complexities 12](#_Toc91617580)

[Jumping from one category of algorithm to another category 12](#_Toc91617581)

[Find the sum of all the number n to m. Given n < m 12](#_Toc91617582)

[Standard Algorithms 12](#_Toc91617583)

CRACKING CODING INTERVIEWS

## COMPETITIVE PROGRAMMING

- GOAL (Convert input to output in efficient way)

- MACHINE/COMPUTER/AUTO-EVALUATION/ONLINE-COMPILER

- Write full program

* Reading Input
* Declaring Variables
* Processing Input
* Producing Output

## CODING INTERVIEW

* Function
* Problem Solving Process
  + Comprehend the Problem
  + Approach (es)
  + Analyze
  + Pick the optimal one based on current problem
  + Code the optimal solution
  + Debug and Fix the Code — Avoid this (proactive think)
* Concept of Functions
  + Defining
  + Calling
  + **No Printing of Answers (Always Returning)**
  + Every Problem Discussed will be Function
* Generate all subsets of given set {1,2,3}
  + The set will have 2^3 subsets

1

1 2

1 3

1 2 3

2

2 3

3

{} // Empty subset

- Function Signature

*vector*<*vector*<int>> generateAllSubsets(*vector*<int> set)

int[] generateAllSubsets(int[] set, int size)

void generateAllSubsets(int[] set, int size) // If we only print.

## DATA TYPES (Primitive)

- int, char (INTEGRAL)

- float (FLOATING)

- boolean (BOOL)

USER DEFINED OR CUSTOM DATA TYPES

- struct

- class

- enum

class Interval {

public:

int startTime;

int endTime;

};

1. Defining the data type.

2. Way of declaring a variable of that data type.

Interval a, b;

int a[]3

char c[];

Interval i[];

3. Access members of this data type

Interval a;

int st = a.startTime;

## Static Array

int a[10];

## Dynamic Array

Things to know...

1. Declaring the dynamic array.
2. Adding elements to the dynamic array.
3. Getting the size of dynamic array.
4. Accessing the elements of the dynamic array.
5. Iterating over the dynamic array.

### C++

*vector*<int> a;

a.*push\_back*();

a.*size*();

int ele = a[0];

### JAVA

ArrayList<Integer> al = new ArrayList<>();

al.add(12);

al.size();

al.get(0);

### C#

*List*<int> li = new *List*<int>();

li.Add(12);

li.*Length*;

int ele = li[0];

### PYTHON

lis = []

lis.*append*(12)

l = *len*(lis)

ele = lis[0]

### In C:

void\* *malloc*(int bytes);

void\* *realloc*(void\* p, int newSize);

void memset(void\*p, int val, int sizeInBytes);

void *memmove*(...)

## How it is implemented internally?

class DynamicArray {

int\* arr;

int capacity = 2;

int size = 0;

void init() {

arr = new int[capacity];

}

public:

void init(int initialCap) {

capacity = initialCap;

init();

}

void push\_back(int x) {

if (size < capacity)

arr[size++] = x;

else{

capacity = capacity \* 2;

int\* temp = new int[capacity];

*memcpy*(temp, arr, size);

delete[] arr;

arr = temp;

}

}

int get(int index) {

return arr[index];

}

};

## Recursion:

- Function calling itself directly or indirectly

- One call.

- Multiple calls.

Example:

int fib(int n) {

if (n <= 1)

eturn n;

else

return fib(n - 1) + fib(n - 2);

}

### Trace the below code for x = 3 , n = 3

int getAnswer(int x, int n)

{

if (n == 0)

return 1;

int half = getAnswer(x, n / 2);

int halfSqr = half \* half;

if ((n & 1) == 1)

return halfSqr \* x;

return halfSqr;

}

### Trace the below code:

#include <iostream>

#include <string>

using namespace *std*;

int a[] = { 1,2,3 };

int an = 3;

void allSubSets(int idx, *string* currentSubset) {

if (idx == an) {

*cout* << "{" << currentSubset << " }\n";

return;

}

allSubSets(idx + 1, currentSubset + " " + *to\_string*(a[idx]));

allSubSets(idx + 1, currentSubset);

}

int main() {

allSubSets(0, "");

return 0;

}

### Trace the below code:

#include <iostream>

#include <string>

using namespace *std*;

*string* s = "abc";

void swap(int i, int j) {

char temp = s[i];

s[i] = s[j];

s[j] = temp;

}

void allPermutations(int idx) {

if (idx == s.*length*()) {

*cout* << s << "\n";

return;

}

for (int nextIdx = idx; nextIdx < s.*length*(); nextIdx++) {

swap(idx, nextIdx);

allPermutations(idx + 1);

swap(idx, nextIdx);

}

}

int main() {

allPermutations(0);

return 0;

}

## WAF for calculating the factorial of a number ‘n’ using Tail recursion.

### What is tail recursion?

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

### 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 procedures by using a stack. This stack consists of all the pertinent information, including the parameter values, for each recursive call.
* 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 (maximum amount of stack space used at any time during compilation) is more.
* 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.

|  |  |
| --- | --- |
| #include <iostream>  #include <string>  using namespace *std*;  int factorial(int n) {  if (n == 0)  return 1;  return n \* factorial(n - 1);  }  int main(void) {  *cout* << factorial(5);  return 0;  } | * This function to calculate the factorial of ‘n’ 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). |

### Tail Call Elimination

#include <iostream>

using namespace *std*;

unsigned factorialHelper(unsigned int n, unsigned int a) {

if (n == 1)

return a;

return factorialHelper(n - 1, n\*a);

}

unsigned int factorial(unsigned int n) {

return factorialHelper(n, 1);

}

int main() {

*cout* << factorial(5);

return 0;

}

## WAP to find the max element in an Array

### Approach 1: Using loop

#include <iostream>

using namespace *std*;

int myArr[] = { -4, 1, 5, 9, -3 };

int findMaxElementInArray(int\* arr, int l) {

int maxEle = arr[0];

for (auto i = 1; i < l; i++) {

if (maxEle < arr[i])

maxEle = arr[i];

}

return maxEle;

}

int main() {

*cout* << findMaxElementInArray(myArr, sizeof(myArr)/sizeof(myArr[0]));

return 0;

}

### Approach 2: Using recursion

#include <iostream>

#include <algorithm>

using namespace *std*;

int myArr[] = { -4, 1, 5, 9, -3 };

int findMaxElementInArrayHelper(int\* arr, int left, int right) {

if (left == right)

return arr[right];

if (left > right)

return *INT\_MIN*;

int mid = (right + left) / 2;

int leftAns = findMaxElementInArrayHelper(arr, left, mid);

int rightAns = findMaxElementInArrayHelper(arr, mid + 1, right);

return *max*(leftAns, rightAns);

}

int findMaxElementInArray(int\* arr, int len) {

return findMaxElementInArrayHelper(arr, 0, len-1);

}

int main() {

*cout* << findMaxElementInArray(myArr, sizeof(myArr)/sizeof(myArr[0]));

return 0;

}

## Time Complexity

* Learn to compare algorithm’s efficiency.
* Understand various categories of algorithms.
* Know best complexities for standard algorithms.

### Steps in Solving a problem

* Find More than One Approach
  + Algorithms.
* Compare Algorithms (Lang/Machine — Neutral)
  + Time taken to execute.
  + Memory used (variables)
* Time Taken by algorithm to run.
  + Time Complexity.
* Memory Used by algorithm to run.
  + Space Complexity.
* Expressed as a MATHEMATICAL FORMULA
  + In terms of INPUT SIZE(s)

## Time Complexity Analysis

* Number of times instructions run in a **Function**.
  + -Practical
  + -Asymptotic (Big-Oh)
* Assignment/Operator Expressions are counted 1.
* Number of times instructions in a **Loop Run**.

### Let’s Understand:

|  |  |
| --- | --- |
| void printDetails(int n) {  for (int i = 1; i < n; i \*= 2)  *printf*("Hello: %d\n", i);  } | Time Complexity: log2n |

* Sequential loop always **adds** up.

|  |  |
| --- | --- |
| void printDetails(int n, int m)  {  int k = 1;  for (int j = 1; j < m; j++)  *printf*("Hello: %d\n", k++);  for (int i = n; i >= 0; i--)  *printf*("Hello: %d\n", k++);  } | Practical Time Complexity:  Outer Loop = m - 1  Inner Loop = n + 1  Big-Oh: O(m+n) |

* Independent nested loops always multiply.

|  |  |
| --- | --- |
| void printDetails(int n, int m)  {  int k = 1;  for (int j = 1; j <= m; j++)  for (int i = n; i >= 0; i - —)  *printf*("Hello: %d\n", k++);  } | Practical Time Complexity:  Outer Loop = m  Inner Loop = n + 1  Big-Oh: O(m+n) |

* Dependent nested loops are treated differently.

|  |  |
| --- | --- |
| void printDetails(int n, int m)  {  int k = 1;  for (int i = 0; i < n; i++)  for (int j = i + 1; j < n; j++)  *printf*("Hello: %d\n", k++);  } | Practical Time Complexity:  For i = 0, Inner Loop = n - 1  For i = 1, Inner Loop = n - 2  For i = 2, Inner Loop = n - 3  …  For i = n, Inner Loop = 0  So, Practical time complexity is = (n\*n+1)/2  Big-Oh: O(n2) |

## Categories of Time and Space Complexities

* Arrange in increasing order of complexity. Consider large values of N (>=100):
  + O(N2)
  + O(logN)
  + O(N3)
  + O(N!)
  + O(NlogN)
  + O(2N)
  + O(1)
  + O(N)
* Answer:
  + O(1) < O(logN) < O(NlogN) < O(N) < O(N2) < O(N3) < O(2N) < O(N!)
  + O(2N) – Subset generation.
  + O(N!) – Permutation generation.

## Jumping from one category of algorithm to another category

### Find the sum of all the number n to m. Given n < m

Approach 1:

|  |  |
| --- | --- |
| int getSum(int n, int m)  {  int sum = 0;  for (auto i = n; i <= m; i++)  sum += i;  return sum;  } | Practical Time Complexity: m - n + 1  **Big-Oh: O(m-n)** |

Approach 2:

|  |  |
| --- | --- |
| int getSum(int n, int m)  {  return (m\*(m + 1) / 2) - (n\*(n - 1) / 2);  } | Practical Time Complexity: 1  **Big-Oh: O(1)** |

## Standard Algorithms

* Sort an array of N (106) integers.
  + Quick, Merge, Heap **O(NlogN)**
  + Insertion, Selection, Bubble O(N2)
  + What if range of integers is only 0 to 100?
    - Count Sort O(N)
* Find index of a number K in sorted array of size N.
  + Binary Search O(logN)
* Find row with max sum in a matrix of size RxC
  + Nested Loops O(R \* C)
* Print all Triplets of an array of size N.
  + Nested Loops O(N3)
* Print all Subsets of an array of size N.
  + O(2N)
* Print all Permutations of a string of size N.
  + O(N!)