# 1. Write a C++ program to compare two given strings and return the number of the positions where they contain the same length 2 substring

Sample Input:

"abcdefgh", "abijsklm"

"abcde", "osuefrcd"

"pqrstuvwx", "pqkdiewx"

Sample Output:

1

1

2

#include <iostream>

using namespace std;

int test(string str1, string str2)

{

int ctr = 0;

for (int i = 0; i < str1.length()-1; i++)

{

string firstString = str1.substr(i, 2);

for (int j = 0; j < str2.length()-1; j++)

{

string secondString = str2.substr(j, 2);

if (firstString==secondString)

ctr++;

}

}

return ctr;

}

int main()

{

cout << test("abcdefgh", "abijsklm") << endl;

cout << test("abcde", "osuefrcd") << endl;

cout << test("pqrstuvwx", "pqkdiewx") << endl;

return 0;

}

# 2. Create a new string from a given string where a specified character have been removed except starting and ending position of the given string.

Sample Input:

"xxHxix", "x"

"abxdddca", "a"

"xabjbhtrb", "b"

Sample Output:

xHix

abxdddca

xajhtrb

#include <iostream>

using namespace std;

string test(string str1, string c)

{

for (int i = str1.length() - 2; i > 0; i--)

{

if (str1[i] == c[0])

{

str1 = str1.erase(i, 1);

}

}

return str1;

}

int main()

{

cout << test("xxHxix", "x") << endl;

cout << test("abxdddca", "a") << endl;

cout << test("xabjbhtrb", "b") << endl;

return 0;

}

# 3. Write a C++ program to create a new string of the characters at indexes 0,1, 4,5, 8,9 ... from a given string

Sample Input:

"Python"

"JavaScript"

"HTML"

Sample Output:

Pyon

JaScpt

HT

#include <iostream>

using namespace std;

string test(string str1)

{

string result = "";

for (int i = 0; i < str1.length(); i += 4)

{

int c = i + 2;

int n = 0;

n += c > str1.length() ? 1 : 2;

result += str1.substr(i, n);

}

return result;

}

int main()

{

cout << test("Python") << endl;

cout << test("JavaScript") << endl;

cout << test("HTML") << endl;

return 0;

}

# 4. Write a C++ programming to get the maximum product from a given integer after breaking the integer into the sum of at least two positive integers

Sample Input: 12

Sample Output: 81

Explanation: 12 = 3 + 3 + 3 + 3, 3 x 3 x 3 x 3 = 81.

Sample Input: 7

Sample Output: 12

Explanation: 7 = 3 + 2 + 2, 3 x 2 x 2 = 12.

#include <iostream>

#include <cmath>

using namespace std;

int integer\_Break(int n) {

if (n == 2) {

return 1;

} else if (n == 3) {

return 2;

} else if (n % 3 == 0) {

return (int) pow(3, n / 3);

} else if (n % 3 == 1) {

return 2 \* 2 \* (int) pow(3, (n - 4) / 3);

} else {

return 2 \* (int) pow(3, n / 3);

}

}

int main()

{

int n = 7;

cout << "\nMaximum product of " << n << " after breaking the integer is " << integer\_Break(n) << endl;

n = 9;

cout << "\nMaximum product of " << n << " after breaking the integer is " << integer\_Break(n) << endl;

n = 12;

cout << "\nMaximum product of " << n << " after breaking the integer is " << integer\_Break(n) << endl;

return 0;

}

# 5. Write a program in C++ to display the pattern like a diamond

Sample Output:

Input number of rows (half of the diamond): 5



#include <iostream>

using namespace std;

int main()

{

int i,j,r;

cout << "\n\n Display the pattern like a diamond:\n";

cout << "----------------------------------------\n";

cout << " Input number of rows (half of the diamond): ";

cin >> r;

for(i=0;i<=r;i++)

{

for(j=1;j<=r-i;j++)

cout<<" ";

for(j=1;j<=2\*i-1;j++)

cout<<"\*";

cout<<endl;

}

for(i=r-1;i>=1;i--)

{

for(j=1;j<=r-i;j++)

cout<<" ";

for(j=1;j<=2\*i-1;j++)

cout<<"\*";

cout<<endl;;

}

}

# 6. Write a program in C++ to print the Floyd's Triangle.

Sample Output:

Input number of rows: 5

1

01

101

0101

10101

#include <iostream>

using namespace std;

int main()

{

int i,j,n,p,q;

cout << "\n\n Print the Floyd's Triangle:\n";

cout << "--------------------------------\n";

cout << " Input number of rows: ";

cin >> n;

for(i=1;i<=n;i++)

{

if(i%2==0)

{

p=1;q=0;

}

else

{

p=0;q=1;

}

for(j=1;j<=i;j++)

if(j%2==0)

cout<<p;

else

cout<<q;

cout<<endl;

}

}

# 7. Write a program in C++ to make such a pattern like a pyramid with numbers increased by 1

Sample Output:

Input number of rows: 4

1

2 3

4 5 6

7 8 9 10

#include <iostream>

#include <string>

using namespace std;

int main()

{

int i,j,spc,rows,k,t=1;

cout << "\n\n Display such a pattern like a pyramid with numbers increased by 1:\n";

cout << "-----------------------------------------------------------------------\n";

cout << " Input number of rows: ";

cin >> rows;

spc=rows+4-1;

for(i=1;i<=rows;i++)

{

for(k=spc;k>=1;k--)

{

cout<<" ";

}

for(j=1;j<=i;j++)

cout<<t++<<" ";

cout<<endl;

spc--;

}

}

# 8. Write a program in C++ to display Pascal's triangle like pyramid.

Sample Output:

Input number of rows: 5

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

#include <iostream>

using namespace std;

int main()

{

int row,c=1,blk,i,j;

cout << "\n\n Display the Pascal's triangle:\n";

cout << "-----------------------------------\n";

cout << " Input number of rows: ";

cin >> row;

for(i=0;i<row;i++)

{

for(blk=1;blk<=row-i;blk++)

cout<<" ";

for(j=0;j<=i;j++)

{

if (j==0||i==0)

c=1;

else

c=c\*(i-j+1)/j;

cout<<c<<" ";

}

cout<<endl;

}

}

# 9. Write a program in C++ to display such a pattern for n number of rows using number. Each row will contain odd numbers of number. The first and last number of each row will be 1 and middle column will be the row number.

Sample Output:

Input number of rows: 5

1

121

12321

1234321

123454321

#include <iostream>

using namespace std;

int main()

{

int i,j,n;

cout << "\n\n Display a pattern using odd number of numbers, the first and last number of each row is 1:\n";

cout << "-----------------------------------------------------------------------------------------------\n";

cout << " Input number of rows: ";

cin >> n;

for(i=0;i<=n;i++)

{

/\* print blank spaces \*/

for(j=1;j<=n-i;j++)

cout<<" ";

/\* Display number in ascending order upto middle\*/

for(j=1;j<=i;j++)

cout<<j;

/\* Display number in reverse order after middle \*/

for(j=i-1;j>=1;j--)

cout<<j;

cout<<endl;

}

}

# 10. Write a program in C++ to display the pattern like pyramid using the alphabet

Sample Output:

Input the number of Letters (less than 26) in the Pyramid: 5

A

A B A

A B C B A

A B C D C B A

A B C D E D C B A

#include <iostream>

using namespace std;

int main()

{

int i, j;

char alph = 'A';

int n, blk;

int ctr = 1;

cout << "\n\n Display the pattern like pyramid using the alphabet:\n";

cout << "---------------------------------------------------------\n";

cout << " Input the number of Letters (less than 26) in the Pyramid: ";

cin >> n;

for (i = 1; i <= n; i++) {

for (blk = 1; blk <= n - i; blk++)

cout << " ";

for (j = 0; j <= (ctr / 2); j++)

{

cout << alph++ << " ";

}

alph = alph - 2;

for (j = 0; j < (ctr / 2); j++)

{

cout << alph-- << " ";

}

ctr = ctr + 2;

alph = 'A';

cout << endl;

}

}

# 11. Write a program in C++ to print a pyramid of digits as shown below for n number of lines.

Sample Output:

Input the number of rows: 5

1

232

34543

4567654

567898765

#include <iostream>

using namespace std;

int main()

{

int i, j, spc, n;

cout << "\n\n Display the pattern like pyramid using digits:\n";

cout << "---------------------------------------------------\n";

cout << " Input the number of rows: ";

cin >> n;

for (i = 1; i <= n; i++)

{

spc = n - i;

while (spc-- > 0)

cout << " ";

for (j = i; j < 2 \* i - 1; j++)

cout << j;

for (j = 2 \* i - 1; j > i - 1; j--)

cout << j;

cout << endl;

}

}

# 12. Write a program in C++ to print a pattern like highest numbers of columns appear in first row.

Sample Output:

Input the number of rows: 5

12345

2345

345

45

5

#include <iostream>

using namespace std;

int main()

{

int i, j, n;

cout << "\n\n Display the pattern like highest numbers of columns appear in first row:\n";

cout << "------------------------------------------------------------------------------\n";

cout << " Input the number of rows: ";

cin >> n;

for (i = 1; i <= n;)

{

cout << i;

for (j = i + 1; j <= n;)

{

cout << j;

j = j + 1;

}

cout << endl;

i = i + 1;

}

}

# 13. Write a program in C++ to display such a pattern for n number of rows using number. Each row will contain odd numbers of number.The first and last number of each row will be 1 and middle column will be the row number. n numbers of columns will appear in 1st row.

Sample Output:

Input number of rows: 7

1234567654321

12345654321

123454321

1234321

12321

121

1

#include <iostream>

using namespace std;

int main()

{

int i,j,n;

cout << "\n\n Display a pattern using odd number of numbers, the n numbers of columns will appear in 1st row:\n";

cout << "----------------------------------------------------------------------------------------------------\n";

cout << " Input number of rows: ";

cin >> n;

for(i=n;i>=1;i--)

{

/\* print blank spaces \*/

for(j=1;j<=n+5-i;j++)

cout<<" ";

/\* Display number in ascending order upto middle\*/

for(j=1;j<=i;j++)

cout<<j;

/\* Display number in reverse order after middle \*/

for(j=i-1;j>=1;j--)

cout<<j;

cout<<endl;

}

}

# 14. Write a program in C++ to convert a hexadecimal number to octal number.

Sample Output:

Input any 32-bit Hexadecimal Number: 5f The equivalant octal number is: 137

#include<iostream>

#include<stdlib.h>

#include<math.h>

using namespace std;

unsigned long Hex\_To\_Dec(char hex[])

{

char \*hexstring;

int length = 0;

const int hexbase = 16;

unsigned long dnum = 0;

int i;

for (hexstring = hex; \*hexstring != '\0'; hexstring++)

{

length++;

}

hexstring = hex;

for (i = 0; \*hexstring != '\0' && i < length; i++, hexstring++)

{

if (\*hexstring >= 48 && \*hexstring <= 57)

{

dnum += (((int)(\*hexstring)) - 48) \* pow(hexbase, length - i - 1);

}

else if ((\*hexstring >= 65 && \*hexstring <= 70))

{

dnum += (((int)(\*hexstring)) - 55) \* pow(hexbase, length - i - 1);

}

else if (\*hexstring >= 97 && \*hexstring <= 102)

{

dnum += (((int)(\*hexstring)) - 87) \* pow(hexbase, length - i - 1);

}

else {

cout<<" The given hexadecimal number is invalid. \n";

}

}

return dnum;

}

int main()

{

unsigned long dnum;

char hex[9];

int dec\_num, rem=1, m, n;

int oct\_num[100],quot;

dec\_num=0;

cout << "\n\n Convert any hexadecimal number to octal number:\n";

cout << "----------------------------------------------------\n";

cout << " Input any 32-bit Hexadecimal Number: ";

cin>>hex;

dnum = Hex\_To\_Dec(hex);

quot = dnum;

cout<<" The equivalent octal number is: ";

quot = dnum;

while(quot != 0)

{

oct\_num[m++] = quot % 8;

quot = quot/8;

}

for(n=m-1; n>=0; n--)

{

cout<<oct\_num[n];

}

cout<<"\n";

}

# 15. Write a program in C++ to convert hexadecimal number to binary number.

Sample Output:

Input any 32-bit Hexadecimal Number: 5f

The equivalant binary number is: 1011111

#include<iostream>

#include<stdlib.h>

#include<math.h>

using namespace std;

unsigned long Hex\_To\_Dec(char hex[])

{

char \*hexstring;

int length = 0;

const int hexbase = 16;

unsigned long dnum = 0;

int i;

for (hexstring = hex; \*hexstring != '\0'; hexstring++)

{

length++;

}

hexstring = hex;

for (i = 0; \*hexstring != '\0' && i < length; i++, hexstring++)

{

if (\*hexstring >= 48 && \*hexstring <= 57)

{

dnum += (((int)(\*hexstring)) - 48) \* pow(hexbase, length - i - 1);

}

else if ((\*hexstring >= 65 && \*hexstring <= 70))

{

dnum += (((int)(\*hexstring)) - 55) \* pow(hexbase, length - i - 1);

}

else if (\*hexstring >= 97 && \*hexstring <= 102)

{

dnum += (((int)(\*hexstring)) - 87) \* pow(hexbase, length - i - 1);

}

else {

cout<<" The given hexadecimal number is invalid. \n";

}

}

return dnum;

}

int main()

{

unsigned long dnum;

char hex[9];

int dec\_num, rem=1, m, n;

int bin\_num[100],quot;

dec\_num=0;

cout << "\n\n Convert any hexadecimal number to binary number:\n";

cout << "------------------------------------------------------\n";

cout << " Input any 32-bit Hexadecimal Number: ";

cin>>hex;

dnum = Hex\_To\_Dec(hex);

quot = dnum;

cout<<" The equivalent binary number is: ";

while(quot != 0)

{

bin\_num[m++] = quot % 2;

quot = quot/2;

}

for(n=m-1; n>=0; n--)

{

dec\_num=(dec\_num\*10)+bin\_num[n];

}

cout<<dec\_num<<endl;

cout<<endl;

}

# 16. Write a program in C++ to convert a hexadecimal number to decimal number.

Sample Output:

Input any 32-bit Hexadecimal Number: 25

The value in decimal number is: 37

#include<iostream>

#include<stdlib.h>

#include<math.h>

using namespace std;

unsigned long Hex\_To\_Dec(char hex[])

{

char \*hexstring;

int length = 0;

const int hexbase = 16;

unsigned long dnum = 0;

int i;

for (hexstring = hex; \*hexstring != '\0'; hexstring++)

{

length++;

}

hexstring = hex;

for (i = 0; \*hexstring != '\0' && i < length; i++, hexstring++)

{

if (\*hexstring >= 48 && \*hexstring <= 57)

{

dnum += (((int)(\*hexstring)) - 48) \* pow(hexbase, length - i - 1);

}

else if ((\*hexstring >= 65 && \*hexstring <= 70))

{

dnum += (((int)(\*hexstring)) - 55) \* pow(hexbase, length - i - 1);

}

else if (\*hexstring >= 97 && \*hexstring <= 102)

{

dnum += (((int)(\*hexstring)) - 87) \* pow(hexbase, length - i - 1);

}

else {

cout<<" The given hexadecimal number is invalid. \n";

}

}

return dnum;

}

int main()

{

unsigned long dnum;

char hex[9];

cout << "\n\n Convert any hexadecimal number to decimal number:\n";

cout << "------------------------------------------------------\n";

cout << " Input any 32-bit Hexadecimal Number: ";

cin>>hex;

dnum = Hex\_To\_Dec(hex);

cout<<" The value in decimal number is: "<<dnum<<"\n";

}

# 17. Write a program that will print the first N numbers for a specific base

Sample Output:

Print the first N numbers for a specific base:

The number 11 in base 10 = 1\*(10^1)+1\*(10^0)=11

Similarly the number 11 in base 7 = 1\*(7^1)+1\*(7^0)=8

----------------------------------------------------------------

Input the number of term: 15

Input the base: 9

The numbers in base 9 are:

1 2 3 4 5 6 7 8 10 11 12 13 14 15 16

#include <iostream>

using namespace std;

int main()

{

int trm, bs, r, q, i, num;

cout << "\n\n Print the first N numbers for a specific base:\n";

cout << " The number 11 in base 10 = 1\*(10^1)+1\*(10^0)=11" << endl;

cout << " Similarly the number 11 in base 7 = 1\*(7^1)+1\*(7^0)=8" << endl;

cout << "----------------------------------------------------------------\n";

cout << " Input the number of term: ";

cin >> trm;

cout << " Input the base: ";

cin >> bs;

cout << " The numbers in base " << bs << " are: " << endl;

for (i = 1; i <= trm; i++)

{

r = i % bs;

q = i / bs;

num = q \* 10 + r;

cout << num << " ";

}

cout << endl;

}

# 18. Write a program in C++ to find two's complement of a binary number.

Sample Output:

Input a 8 bit binary value: 01101110

The original binary = 01101110

After ones complement the value = 10010001

After twos complement the value = 10010010

#include <iostream>

#define SZ 8

using namespace std;

int main()

{

char bn[SZ + 1], onComp[SZ + 1], twComp[SZ + 1];

int i, carr = 1;

int er = 0;

cout << "\n\n Find two's complement of a binary value:\n";

cout << "----------------------------------------------\n";

cout << " Input a " << SZ << " bit binary value: ";

cin >> bn;

for (i = 0; i < SZ; i++)

{

if (bn[i] == '1')

{

onComp[i] = '0';

}

else if (bn[i] == '0')

{

onComp[i] = '1';

}

else

{

cout << "Invalid Input. Input the value of assign bits." << endl;

er = 1;

break;

}

}

onComp[SZ] = '\0';

for (i = SZ - 1; i >= 0; i--)

{

if (onComp[i] == '1' && carr == 1)

{

twComp[i] = '0';

}

else if (onComp[i] == '0' && carr == 1)

{

twComp[i] = '1';

carr = 0;

}

else

{

twComp[i] = onComp[i];

}

}

twComp[SZ] = '\0';

if (er == 0)

{

cout << " The original binary = " << bn << endl;

cout << " After ones complement the value = " << onComp << endl;

cout << " After twos complement the value = " << twComp << endl;

}

}

# 19. Find all n–digit numbers with a given sum where n varies from 1 to 9 and sum <= 81 (Maximum possible sum in a 9–digit number).

3–digit numbers with sum 6 are

105 114 123 132 141 150 204 213 222 231 240 303 312 321 330 402 411 420 501 510 600

5–digit numbers with sum 42 are

69999 78999 79899 79989 79998 87999 88899 88989 88998 89799 89889 89898 89979 89988 89997 96999 97899 97989 97998 98799 98889 98898 98979 98988 98997 99699 99789 99798 99879 99888 99897 99969 99978 99987 99996

#include <stdio.h>

// Function to find all n–digit numbers with a sum of digits equal

// to `target` in a bottom-up manner

void findNdigitNums(char result[], int index, int n, int target)

{

// if the number is less than n–digit and its sum of digits is

// less than the given sum

if (index < n && target >= 0)

{

char d = '0';

// special case: number cannot start from 0

if (index == 0) {

d = '1';

}

// consider every valid digit and put it in the current

// index and recur for the next index

while (d <= '9')

{

result[index] = d;

int digit = (d - '0');

findNdigitNums(result, index + 1, n, target - digit);

d++;

}

}

// if the number becomes n–digit and its sum of digits is

// equal to the given sum, print it

else if (index == n && target == 0) {

printf("%s ", result);

}

}

int main()

{

int n = 3; // n–digit

int target = 6; // given sum

// character array to store the result

char result[n + 1];

result[n] = '\0'; // null terminate the array

findNdigitNums(result, 0, n, target);

return 0;

}

# 20. Given a string, find the first non-repeating character in it by doing only a single traversal of it.

Input:string is ABCDBAGHC

Output:The first non-repeating character in the string is D

#include <iostream>

#include <unordered\_map>

using namespace std;

// Function to find the first non-repeating character in

// the string by doing only a single traversal of it

int findNonRepeatingChar(string str)

{

// map to store character count and the index of its

// last occurrence in the string

unordered\_map<char, pair<int, int>> map;

for (int i = 0; i < str.length(); i++)

{

map[str[i]].first++;

map[str[i]].second = i;

}

// stores index of the first non-repeating character

int min\_index = -1;

// traverse the map and find a character with count 1 and

// a minimum index of the string

for (auto it: map)

{

int count = it.second.first;

int firstIndex = it.second.second;

if (count == 1 && (min\_index == -1 || firstIndex < min\_index)) {

min\_index = firstIndex;

}

}

return min\_index;

}

int main()

{

string str = "ABCDBAGHC";

int index = findNonRepeatingChar(str);

if (index != -1) {

cout << "The first non-repeating character in the string is " << str[index];

} else {

cout << "The string has no non-repeating character";

}

return 0;

}

# 21. Given a string, remove adjacent duplicates characters from it. In other words, remove all consecutive same characters except one.

Input: AABBBCDDD

Output: ABCD

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Function to remove adjacent duplicates characters from a string

void removeDuplicates(char s[])

{

int n = strlen(s);

char prev = '\0';

int k = 0;

// loop through the string

for (int i = 0; i < n; i++)

{

// if the current char is different from the previous char

if (prev != s[i])

{

// set distinct chars at index `k` and increment it

s[k++] = s[i];

}

// update previous char to current char for the next iteration of the loop

prev = s[i];

}

// null terminate the resultant string

s[k] = '\0';

}

int main(void)

{

char s[] = "AAABBCDDD";

removeDuplicates(s);

printf("%s", s);

return 0;

}

# 22. Calculate the total number of ways to achieve a given sum with n throws of dice having k faces.

**Input:**

The total number of throws n is 2

The total number of faces k is 6 (i.e., each dice has values from 1 to 6)

The desired sum is 10

**Output:**

The total number of ways is 3.

The possible throws are (6, 4), (4, 6), (5, 5)

| #include <stdio.h>    // Function to calculate the total number of ways to achieve a given  // sum with `n` throws of dice having `k` faces  int count(int n, int k, int target)  {  // if the desired sum is reached with `n` dices  if (n == 0) {  return (target == 0) ? 1 : 0;  }    // the desired sum can't be reached with the current configuration  if (target < 0 || k \* n < target || n > target) {  return 0;  }  int result = 0;  // recur for all possible solutions  for (int i = 1; i <= k; i++) {  result += count(n - 1, k, target - i);  }  return result;  }    int main(void)  {  int n = 4; // n throws  int k = 6; // values 1 to 6  int target = 15; // desired sum  printf("The total number of ways is %d", count(n, k, target));  return 0;  } **22.** **A numeric array of length N is given. We need to design a function that finds all positive numbers in the array that have their opposites in it as well. Describe approaches for solving optimal worst case and optimal average case performance, respectively.** **Level: Hard**  **Answer:**  Let us first design an approach with optimal worst case time.  We need to compare numbers to see if they have their opposites in the array. The trivial solution of comparing all numbers has a consistent time of O(N^2). The great number of comparisons involved should suggest trying to establish a total order operator that allows us to use sorting for solving the problem. If we define a comparison operator that places all instances of a number right after all instances of its opposite, we would have exactly pair of consecutive opposite numbers for each number that has its opposite in the array.  An example of what we want to achieve:  Array: -7 4 -3 2 2 -8 -2 3 3 7 -2 3 -2  Sorted: -2 -2 -2 2 2 -3 3 3 4 -7 7 -8  We see that after our special sorting method, we have [-2, 2], [-3, 3] and [-7, 7] combinations happening consecutively exactly once. Implementing this comparison is simple and it can be implemented as follows.  FUNCTION compare(a, b)  IF a != b and a != -b  RETURN abs(a) < abs(b)  ELSE  RETURN a < b  If the numbers aren’t equal or opposite, we sort them by their absolute value, but if they are, we sort them by their sign. Finally, a solution based on this is now very simple:  FUNCTION find\_numbers\_with\_opposites(numbers)  answer = List  sorted\_numbers = sort\_by(numbers, compare)  FOR n IN [1..sorted\_numbers.length()]  IF sorted\_numbers[n] > 0 AND sorted\_numbers[n - 1] == -sorted\_numbers[n]  answer.push(n)  END IF  END FOR  RETURN answer  This implementation has a worst case runtime complexity of O(N log N), with the sorting algorithm being the bottleneck.  Optimal average case time complexity of O(N) can be achieved using Hash Tables. We map numbers to their absolute values, and check if their opposites are already in the Hash Table.  FUNCTION find\_numbers\_with\_opposites(numbers)  table = HashTable<number, number>  answer = List  FOR number IN numbers  IF number == 0  CONTINUE  END IF  key = abs(number)  IF key not in table  table[key] = number  ELSE IF table[key] = -number  answer.push(key)  table[key] = 0  END IF  END FOR  We change the value in the table to something that will never be equal to any of the numbers in the array so we don’t return duplicate results from duplicate matches.  All HashTable operations have an average time complexity of O(1), and our complexity is a result of performing operations N times. 23. Design an algorithm that finds the number of ways in which you can traverse N meters by doing jumps of 1, 2, 3, 4, or 5 meter lengths. Assume that N can be a very large number. What is the resulting complexity? **Answer:**  We can use dynamic programming for solving this problem. Let’s use n[k] to represent the number of ways we can reach distance k. That distance can be reached by jumping from one of the 5 previous distances. Thus the number of ways in which we can reach this distance is the sum of the ways in which we can reach the previous 5 distances:  n[k] = n[k-1] + n[k-2] + n[k-3] + n[k-4] + n[k-5]  The solution is a simple for loop.  FUNCTION ways(N)  Array n[N+1]  n[0] = 1  FOR k IN [1..N]  n[k] = 0  FOR d IN [1..min(5, k)+1]  n[k] += n[k - d]  END FOR  END FOR  RETURN n[N]  This solution has a time complexity of O(N). But, we can have even better performance. The given sum can be represented as a 1x5 matrix of ones multiplied by a 5x1 matrix of previous elements. If we use the same approach for shifting, we can get the relation B[k] = A \* B[k-1], where:  B[k] =  [n[k-4] ]  [n[k-3] ]  [n[k-2] ]  [n[k-1] ]  [n[k] ]  A =  [0 1 0 0 0]  [0 0 1 0 0]  [0 0 0 1 0]  [0 0 0 0 1]  [1 1 1 1 1]  If B[0] = [0 0 0 0 1]’, then [0 0 0 0 1] \* B[k] = n[k]. Now, due to B[k] = A \* B[k-1], B[k] = A^T \* B[0]. With that, the solution of our problem can be represented as a relation n[N] = [0 0 0 0 1] \* A^N \* [0 0 0 0 1]’. If we use the previously mentioned optimal approach for calculating pow(A, N), this solution has an O(log N) time complexity. We have to keep in mind that this does have a high constant bound to this complexity, since matrix multiplication takes time. But, for a large enough N, this solution is optimal. 24. We are given an array of numbers. How would we find the sum of a certain subarray? How could we query an arbitrary number of times for the sum of any subarray? If we wanted to be able to update the array in between sum queries, what would be the optimal solution then? What’s the preprocessing and query complexity for each solution. **Answer:**  For the sake of notation, let us represent the length of the array as N.  The first problem consists of calculating the sum of the array. There is no preprocessing involved and we do one summing operation of O(N) complexity.  The second problem needs to calculates sums multiple times. Thus, it would be wise to perform preprocessing to reduce the complexity of each query. Let’s replace the create an array of subsums s[0:N+1] for the array a[0:N], that is:  s[0] = 0  FOR k in [1..N+1]  s[k] = s[k-1] + a[k-1]  END FOR  Now each element k stores the sum of a[0:k]. To query the sum of a subarray a[p:q], to take the sum of all elements until q s[q] and subtract the sum of all elements before p s[p], that is subsum(p, q) = s[q] - s[p]  The preprocessing for this method takes O(N) time, but each query takes O(1) time to perform.  The hardest problem is responding to an arbitrary number of data updates and queries. First, let us look at the previous solutions. The first solution has O(1) insertion complexity, but O(N) query complexity. The second solution has the opposite, O(N) insertion and O(1) queries. Neither of these approaches is ideal for the general case. Ideally, we want to achieve a low complexity for both operations.  A **Fenwick tree (or binary indexed tree)** is ideal for this problem. We maintain an array tree[0:N+1] of values, where every N-th item stores the sum(a[M:N]), and where M is equal to N with the least significant 1 in its binary representation replaced by 0. So for example, N = 19 = b10011, M = 18=b10010; N = 20 = b10100, M=16 = b10000. Now we can easily calculate the sum by following M until we reach 0. Updates are done in the opposite direction.  A = Array[N]  Tree = Array[N+1]  FUNCTION sum(end)  result = 0  WHILE end > 0  result += tree[end]  last\_one = end & -end  end -= last\_one  END WHILE  RETURN result  FUNCTION update(index, value)  increment = value - a[index]  WHILE index < tree.length()  tree[index] += increment  last\_one = index & -index  index += last\_one  END WHILE  FUNCTION query(p, q)  RETURN sum(q) - sum(p)  Both operations require O(log N) complexity, which is better than either previous approach. 25. You need to design a scheduler that to schedule a set of tasks. A number of the tasks need to wait for some other tasks to complete prior to running themselves. What algorithm could we use to design the schedule and how would we implement it? **Level: Hard**  **Answer:**  What we need to do is a topological sort. We connect a graph of all the task dependencies. We then mark the number of dependencies for each node and add nodes with zero dependencies to a queue. As we take nodes from that queue, we remove a dependency from all of its children. As nodes reach zero dependencies, we add them to the queue.  After the algorithm executes, if the list doesn’t have as many elements as the number of tasks, we have circular dependencies. Otherwise, we have a solution:  FUNCTION schedule(nodes)  dependencies = Array[nodes.length()]  FOR node IN nodes  FOR child IN node.children  dependencies[child] += 1  END FOR  END FOR  queue = Queue  solution = List  FOR n IN [0..nodes.length()]  IF dependencies[n] == 0  queue.push(n)  END FOR  END FOR  WHILE !queue.empty()  node = queue.pop()  solution.push(node)  FOR n IN nodes[node].children  dependencies[n] -= 1  IF dependencies[n] == 0  queue.push(n)  END IF  END FOR  END WHILE  IF solution.length() != nodes.length()  RETURN ERROR(“Problem contained circular dependencies”)  ELSE  RETURN solution  END IF |
| --- |
| 26. You have a set of date intervals represented by StartDate and EndDate. How would you efficiently calculate the longest timespan covered by them? What will be the time complexity?  **Level: Hard**  **Answer:**  FUNCTION MaxTimespan(StartDates, EndDates)  Sort(StartDates, EndDates)  ActStartDate = StartDates[1]  ActEndDate = EndDates[1]  ActTimespan = ActEndDate - ActStartDate  FOR i in 2..StartDates.Length  IF StartDates[i] BETWEEN ActStartDate AND ActEndDate  ActEndDate = MAX(ActEndDate, EndDates[i])  ActTimespan = MAX(ActTimespan, ActEndDate - ActStartDate)  ELSE  ActStartDate = StartDates[i]  ActEndDate = EndDates[i]  END IF  END FOR  RETURN ActTimespan  The overall time complexity is O(NlogN) because:   1. Sort intervals by start date. Time complexity is O(NlogN). 2. Take first interval as actual range. Loop over intervals and if the current StartDate is within the actual range, extend EndDate of the actual range if 3. needed and extend maximal timespan achieved so far if needed. Otherwise, use current interval as new actual range. Time complexity is O(N).  **27.**  **How to get the non-matching characters in a string?** **Answer:** **To get the non-matching characters in a string, the below steps are followed:**   1. Hash Map data structure is taken which works with the key-value pair. 2. Loop the string, character by character, and verify if that character of the string exists in the hash map or not. 3. If the result is true, the counter for the character in the hash map is increased or else then put a count as 1. 4. Once the loop ends, then the Hash map is traversed and print the characters with a count equal to 1.   **Code snippet:**   | HashMap<Character, Integer> mp = new HashMap<> ();  for (int j = 0; j<text.length (); j++) {  char ch = text.charAt(j);  if(mp.containsKey(ch)){  int cnt = mp.get(ch);  mp.put(ch, ++cnt);  }else{  mp.put(ch, 1);  }  }  Set<Character> charct = map.keySet();  for (Character ch: charct){  int c= mp.get(ch);  if(c==1){  System.out.println(ch+ " - " + c);  }  } | | --- |  28. How to calculate the number of vowels and consonants in a string? **Answer: To calculate the number of vowels and consonants in a string, the below steps are followed:**   1. Get the string on which count has to be performed. 2. Run a loop from 0 to the length of the string. 3. Take a single character at a time and verify if they are a part of the group of vowels. 4. If the result is true, increase the count of vowels or else increment the count of consonants.   **Code snippet:**   | for (int k = 0; k < text.length(); k++) {  char c = text.charAt(k);  if (c == 'a' || c == 'e' || c == 'i' ||  c == 'o' || c == 'u')  owls += vowls  else  consonts += consonts  }  System.out.println("Vowel count is " + vowls);  System.out.println("Consonant count is: " + consonts); | | --- | |

# 29. How do you prove that the two strings are anagrams?

**Answer:** Two strings are called anagrams if they accommodate a similar group of characters in a varied sequence.

**To check if two strings are anagrams, the below steps are followed:**

1. Initialize two strings in two variables.
2. Check if the length of the two strings is similar, if not then the strings are not an anagram.
3. If the result is true, take the two strings and store them in a character array.
4. Sort the two character arrays, then check if the two sorted arrays are alike.
5. If the result is true, the two strings are anagram else, not anagram.

**Code snippet:**

| if (str1.length() != str2.length()) {  System.out.println(str1 + " and " +str2 + " not anagrams string");  }else{  char[] anagram1 = str1.toCharArray();  char[] anagram2 = str2.toCharArray();  Arrays.sort(anagram1);  Arrays.sort(anagram2);  anagrmstat = Arrays.equals(anagram1, anagram2);  }  if (anagrmstat == true) {  System.out.println(str1 + " and " +str2 + " anagrams string");  }else{  System.out.println(str1 + " and " +str2 + " not anagrams string");  }  } |
| --- |

# 30. Print all triplets that form a geometric. progression

**Given a sorted array of distinct positive integers, print all triplets that forms a geometric progression with an integral common ratio.**

**input: A[] = { 1, 2, 6, 10, 18, 54 }   
Output:  
2 6 18  
6 18 54   
  
Input: A[] = { 2, 8, 10, 15, 16, 30, 32, 64 }   
Output:  
2 8 32  
8 16 32  
16 32 64**

**Input: A[] = { 1, 2, 6, 18, 36, 54 }  
Output:  
2 6 18  
1 6 36  
6 18 54**

**Input: A[] = { 1, 2, 4, 16 }  
Output:  
1 2 4  
1 4 16**

**Input: A[] = {1, 2, 3, 6, 18, 22};  
Output:  
2 6 18**

| **#include <stdio.h>**    **// Function to print all triplets that forms geometric progression**  **// in a given sorted array**  **void findAllTriplets(int A[], int n)**  **{**  **if (n < 3) {**  **return;**  **}**    **// One by one, fix every element as a middle element**  **for (int j = 1; j < n - 1; j++)**  **{**  **// Initialize `i` and `k` for current `j`**  **int i = j - 1, k = j + 1;**    **// Find all `i` and `k` such that `(i, j, k)` form a GP triplet**  **while (1)**  **{**  **// If `A[j]/A[i] = r`, and `A[k]/A[j] = r`, and `r` is an integer,**  **// `(i, j, k)` forms a GP**  **while (i >= 0 && k < n && (A[j] % A[i] == 0) &&**  **(A[k] % A[j] == 0) && (A[j] / A[i] == A[k] / A[j]))**  **{**  **// print the triplet**  **printf("%d %d %d\n", A[i], A[j], A[k]);**    **// since the array is sorted and elements are distinct**  **k++ , i--;**  **}**    **if (i < 0 || k >= n) {**  **break;**  **}**    **// If `A[j]` is multiple of `A[i]` and `A[k]` is multiple of `A[j]`,**  **// then `A[j] / A[i] != A[k] / A[j]`; compare their values to**  **// move to the next `k` or previous `i`**  **if (A[j] % A[i] == 0 && A[k] % A[j] == 0)**  **{**  **if (A[j] / A[i] < A[k] / A[j]) {**  **i--;**  **}**  **else {**  **k++;**  **}**  **}**    **// Otherwise, if `A[j]` is a multiple of `A[i]`, try next `k`.**  **// Else, try the previous `i`.**  **else if (A[j] % A[i] == 0) {**  **k++;**  **}**  **else {**  **i--;**  **}**  **}**  **}**  **}**    **int main()**  **{**  **int A[] = { 1, 2, 6, 10, 18, 54 };**  **int n = sizeof(A) / sizeof(A[0]);**    **findAllTriplets(A, n);**    **return 0;**  **}** |
| --- |

**31. Check if a subarray with 0 sum exists or not**

**Given an integer array, check if it contains a subarray having zero-sum.  
Input: { 3, 4, -7, 3, 1, 3, 1, -4, -2, -2 }**

**Output: Subarray with zero-sum exists**

**The subarrays with a sum of 0 are:**

**{ 3, 4, -7 }**

**{ 4, -7, 3 }**

**{ -7, 3, 1, 3 }**

**{ 3, 1, -4 }**

**{ 3, 1, 3, 1, -4, -2, -2 }**

**{ 3, 4, -7, 3, 1, 3, 1, -4, -2, -2 }**

**#include <iostream>**

**#include <unordered\_set>**

**using namespace std;**

**// Function to check if subarray with zero-sum is present in a given array or not**

**bool hasZeroSumSubarray(int nums[], int n)**

**{**

**// create an empty set to store the sum of elements of each**

**// subarray `nums[0…i]`, where `0 <= i < n`**

**unordered\_set<int> set;**

**// insert 0 into the set to handle the case when subarray with**

**// zero-sum starts from index 0**

**set.insert(0);**

**int sum = 0;**

**// traverse the given array**

**for (int i = 0; i < n; i++)**

**{**

**// sum of elements so far**

**sum += nums[i];**

**// if the sum is seen before, we have found a subarray with zero-sum**

**if (set.find(sum) != set.end()) {**

**return true;**

**}**

**else {**

**// insert sum so far into the set**

**set.insert(sum);**

**}**

**}**

**// we reach here when no subarray with zero-sum exists**

**return false;**

**}**

**int main()**

**{**

**int nums[] = { 4, 2, -3, -1, 0, 4 };**

**int n = sizeof(nums)/sizeof(nums[0]);**

**hasZeroSumSubarray(nums, n) ?**

**cout << "Subarray exists":**

**cout << "Subarray does not exist";**

**return 0;**

**}**

**32. Print all subarrays with 0 sum**

**Given an integer array, print all subarrays with zero-sum.  
Input: { 4, 2, -3, -1, 0, 4 }**

**Subarrays with zero-sum are**

**{ -3, -1, 0, 4 }**

**{ 0 }**

**Input: { 3, 4, -7, 3, 1, 3, 1, -4, -2, -2 }**

**Subarrays with zero-sum are**

**{ 3, 4, -7 }**

**{ 4, -7, 3 }**

**{ -7, 3, 1, 3 }**

**{ 3, 1, -4 }**

**{ 3, 1, 3, 1, -4, -2, -2 }**

**{ 3, 4, -7, 3, 1, 3, 1, -4, -2, -2 }**

**#include <iostream>**

**#include <unordered\_map>**

**using namespace std;**

**// Function to print all subarrays with a zero-sum**

**// in a given array**

**void printAllSubarrays(int nums[], int n)**

**{**

**// consider all subarrays starting from `i`**

**for (int i = 0; i < n; i++)**

**{**

**int sum = 0;**

**// consider all subarrays ending at `j`**

**for (int j = i; j < n; j++)**

**{**

**// sum of elements so far**

**sum += nums[j];**

**// if the sum is seen before, we have found a subarray**

**// with zero-sum**

**if (sum == 0) {**

**cout << "Subarray [" << i << "…" << j << "]\n";**

**}**

**}**

**}**

**}**

**int main()**

**{**

**int nums[] = { 3, 4, -7, 3, 1, 3, 1, -4, -2, -2 };**

**int n = sizeof(nums)/sizeof(nums[0]);**

**printAllSubarrays(nums, n);**

**return 0;**

**}**

**33: Find maximum length subarray having a given sum**

**Given an integer array, find the maximum length subarray having a given sum.  
nums[] = { 5, 6, -5, 5, 3, 5, 3, -2, 0 }**

**target = 8**

**Subarrays with sum 8 are**

**{ -5, 5, 3, 5 }**

**{ 3, 5 }**

**{ 5, 3 }**

**The longest subarray is { -5, 5, 3, 5 } having length 4**

| **#include <stdio.h>**    **// Naive function to find the maximum length subarray with sum `S` present**  **// in a given array**  **void findMaxLenSubarray(int nums[], int n, int S)**  **{**  **// `len` stores the maximum length of subarray with sum `S`**  **int len = 0;**    **// stores ending index of the maximum length subarray having sum `S`**  **int ending\_index = -1;**    **// consider all subarrays starting from `i`**  **for (int i = 0; i < n; i++)**  **{**  **int target = 0;**    **// consider all subarrays ending at `j`**  **for (int j = i; j < n; j++)**  **{**  **// sum of elements in the current subarray**  **target += nums[j];**    **// if we have found a subarray with sum `S`**  **if (target == S)**  **{**  **// update length and ending index of max length subarray**  **if (len < j - i + 1)**  **{**  **len = j - i + 1;**  **ending\_index = j;**  **}**  **}**  **}**  **}**    **// print the subarray**  **printf("[%d, %d]", ending\_index - len + 1, ending\_index);**  **}**    **int main(void)**  **{**  **int nums[] = { 5, 6, -5, 5, 3, 5, 3, -2, 0 };**  **int target = 8;**    **int n = sizeof(nums)/sizeof(nums[0]);**    **findMaxLenSubarray(nums, n, target);**    **return 0;**  **}** |
| --- |

**34: Find the largest subarray having an equal number of 0’s and 1’s**

**Given a binary array containing 0’s and 1’s, find the largest subarray with equal numbers of 0’s and 1’s.  
Input: { 0, 0, 1, 0, 1, 0, 0 }**

**Output: Largest subarray is { 0, 1, 0, 1 } or { 1, 0, 1, 0 }**

**#include <iostream>**

**#include <unordered\_map>**

**using namespace std;**

**// Function to find the largest subarray having an equal number**

**// of 0's and 1's**

**void findLargestSubarray(int nums[], int n)**

**{**

**// create an empty map to store the ending index of the first subarray**

**// having some sum**

**unordered\_map<int, int> map;**

**// insert (0, -1) pair into the set to handle the case when a**

**// subarray with zero-sum starts from index 0**

**map[0] = -1;**

**// `len` stores the maximum length of subarray with zero-sum**

**int len = 0;**

**// stores ending index of the largest subarray having zero-sum**

**int ending\_index = -1;**

**int sum = 0;**

**// Traverse through the given array**

**for (int i = 0; i < n; i++)**

**{**

**// sum of elements so far (replace 0 with -1)**

**sum += (nums[i] == 0)? -1 : 1;**

**// if the sum is seen before**

**if (map.find(sum) != map.end())**

**{**

**// update length and ending index of largest subarray having zero-sum**

**if (len < i - map[sum])**

**{**

**len = i - map[sum];**

**ending\_index = i;**

**}**

**}**

**// if the sum is seen for the first time, insert the sum with its**

**// index into the map**

**else {**

**map[sum] = i;**

**}**

**}**

**// print the subarray if present**

**if (ending\_index != -1) {**

**cout << "[" << ending\_index - len + 1 << ", " << ending\_index << "]";**

**}**

**else {**

**cout << "No subarray exists";**

**}**

**}**

**int main()**

**{**

**int nums[] = { 0, 0, 1, 0, 1, 0, 0 };**

**int n = sizeof(nums) / sizeof(nums[0]);**

**findLargestSubarray(nums, n);**

**return 0;**

**}**

**35: Sort an array of 0’s, 1’s, and 2’s (Dutch National Flag Problem)**

**Given an array containing only 0’s, 1’s, and 2’s, sort it in linear time and using constant space.  
Input: { 0, 1, 2, 2, 1, 0, 0, 2, 0, 1, 1, 0 }**

**Output: { 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2 }**

**#include <stdio.h>**

**// Utility function to swap elements `A[i]` and `A[j]` in an array**

**void swap(int A[], int i, int j)**

**{**

**int temp = A[i];**

**A[i] = A[j];**

**A[j] = temp;**

**}**

**// Linear time partition routine to sort an array containing 0, 1, and 2.**

**// It is similar to 3–way partitioning for the Dutch national flag problem.**

**void threeWayPartition(int A[], int end)**

**{**

**int start = 0, mid = 0;**

**int pivot = 1;**

**while (mid <= end)**

**{**

**if (A[mid] < pivot) // current element is 0**

**{**

**swap(A, start, mid);**

**++start, ++mid;**

**}**

**else if (A[mid] > pivot) // current element is 2**

**{**

**swap(A, mid, end);**

**--end;**

**}**

**else { // current element is 1**

**++mid;**

**}**

**}**

**}**

**int main()**

**{**

**int A[] = { 0, 1, 2, 2, 1, 0, 0, 2, 0, 1, 1, 0 };**

**int n = sizeof(A)/sizeof(A[0]);**

**threeWayPartition(A, n - 1);**

**for (int i = 0; i < n; i++) {**

**printf("%d ", A[i]);**

**}**

**return 0;**

**}**

**36: Merge two arrays by satisfying given constraints**

**Given two sorted arrays X[] and Y[] of size m and n each where m >= n and X[] has exactly n vacant cells, merge elements of Y[] in their correct position in array X[], i.e., merge (X, Y) by keeping the sorted order.**

**Input:**

**X[] = { 0, 2, 0, 3, 0, 5, 6, 0, 0 }**

**Y[] = { 1, 8, 9, 10, 15 }**

**The vacant cells in X[] is represented by 0**

**Output:**

**X[] = { 1, 2, 3, 5, 6, 8, 9, 10, 15 }**

**#include <stdio.h>**

**// Function to merge `X[0… m]` and `Y[0… n]` into `X[0… m+n+1]`**

**void merge(int X[], int Y[], int m, int n)**

**{**

**// size of `X[]` is `k+1`**

**int k = m + n + 1;**

**// run if X[] or Y[] has elements left**

**while (m >= 0 && n >= 0)**

**{**

**// put the next greater element in the next free position in `X[]` from the end**

**if (X[m] > Y[n]) {**

**X[k--] = X[m--];**

**}**

**else {**

**X[k--] = Y[n--];**

**}**

**}**

**// copy the remaining elements of `Y[]` (if any) to `X[]`**

**while (n >= 0) {**

**X[k--] = Y[n--];**

**}**

**// fill `Y[]` with all zeroes**

**for (int i = 0; i < n; i++) {**

**Y[i] = 0;**

**}**

**}**

**// The function moves non-empty elements in `X[]` in the**

**// beginning and then merge them with `Y[]`**

**void rearrange(int X[], int Y[], int m, int n)**

**{**

**// return if `X` is empty**

**if (m == 0) {**

**return;**

**}**

**// moves non-empty elements of `X[]` at the beginning**

**int k = 0;**

**for (int i = 0; i < m; i++)**

**{**

**if (X[i] != 0) {**

**X[k++] = X[i];**

**}**

**}**

**// merge `X[0 … k-1]` and `Y[0 … n-1]` into `X[0 … m-1]`**

**merge(X, Y, k - 1, n - 1);**

**}**

**int main()**

**{**

**// vacant cells in `X[]` is represented by 0**

**int X[] = { 0, 2, 0, 3, 0, 5, 6, 0, 0 };**

**int Y[] = { 1, 8, 9, 10, 15 };**

**int m = sizeof(X) / sizeof(X[0]);**

**int n = sizeof(Y) / sizeof(Y[0]);**

**/\* Validate input before calling `rearrange()`**

**1. Both arrays `X[]` and `Y[]` should be sorted (ignore 0's in `X[]`)**

**2. Size of array `X[]` >= size of array `Y[]` (i.e., `m >= n`)**

**3. Total number of vacant cells in array `X[]` = size of array `Y[]` \*/**

**// merge `Y[]` into `X[]`**

**rearrange(X, Y, m, n);**

**// print merged array**

**for (int i = 0; i < m; i++) {**

**printf("%d ", X[i]);**

**}**

**return 0;**

**}**

**37: Shuffle an array using Fisher–Yates shuffle algorithm**

**Given an integer array,** [**in-place**](https://www.techiedelight.com/in-place-vs-out-of-place-algorithms/) **shuffle it. The algorithm should produce an unbiased permutation, i.e., every permutation is equally likely.**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <time.h>**

**// Utility function to swap elements `A[i]` and `A[j]` in an array**

**void swap(int A[], int i, int j)**

**{**

**int temp = A[i];**

**A[i] = A[j];**

**A[j] = temp;**

**}**

**// Function to shuffle an array `A[]` of `n` elements**

**void shuffle(int A[], int n)**

**{**

**// read array from the highest index to lowest**

**for (int i = n - 1; i >= 1; i--)**

**{**

**// generate a random number `j` such that `0 <= j <= i`**

**int j = rand() % (i + 1);**

**// swap the current element with the randomly generated index**

**swap(A, i, j);**

**}**

**}**

**int main(void)**

**{**

**int A[] = { 1, 2, 3, 4, 5, 6 };**

**int n = sizeof(A) / sizeof(A[0]);**

**// seed for random input**

**srand(time(NULL));**

**shuffle(A, n);**

**// print the shuffled array**

**for (int i = 0; i < n; i++) {**

**printf("%d ", A[i]);**

**}**

**return 0;**

**}**

**38: Rearrange an array with alternate high and low elements**

**Given an integer array, rearrange it such that every second element becomes greater than its left and right elements. Assume no duplicate elements are present in the array.  
Input: {1, 2, 3, 4, 5, 6, 7}**

**Output: {1, 3, 2, 5, 4, 7, 6}**

**Input: {9, 6, 8, 3, 7}**

**Output: {6, 9, 3, 8, 7}**

**Input: {6, 9, 2, 5, 1, 4}**

**Output: {6, 9, 2, 5, 1, 4}**

**#include <stdio.h>**

**// Utility function to swap elements `arr[i]` and `arr[j]` in an array**

**void swap(int arr[], int i, int j)**

**{**

**int temp = arr[i];**

**arr[i] = arr[j];**

**arr[j] = temp;**

**}**

**// Function to rearrange the array such that every second element**

**// of the array is greater than its left and right elements**

**void rearrangeArray(int arr[], int n)**

**{**

**// start from the second element and increment index**

**// by 2 for each iteration of the loop**

**for (int i = 1; i < n; i += 2)**

**{**

**// if the previous element is greater than the current element,**

**// swap the elements**

**if (arr[i - 1] > arr[i]) {**

**swap(arr, i - 1, i);**

**}**

**// if the next element is greater than the current element,**

**// swap the elements**

**if (i + 1 < n && arr[i + 1] > arr[i]) {**

**swap(arr, i + 1, i);**

**}**

**}**

**}**

**int main(void)**

**{**

**int arr[] = { 9, 6, 8, 3, 7 };**

**int n = sizeof(arr) / sizeof(arr[0]);**

**rearrangeArray(arr, n);**

**// print output array**

**for (int i = 0; i < n; i++) {**

**printf("%d ", arr[i]);**

**}**

**return 0;**

**}**

**39: Find the largest subarray formed by consecutive integers**

**Given an integer array, find the largest subarray formed by consecutive integers. The subarray should contain all distinct values  
Input: { 2, 0, 2, 1, 4, 3, 1, 0 }**

**Output: The largest subarray is { 0, 2, 1, 4, 3 }**

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**// Function to check if subarray `A[i…j]` is formed by consecutive integers.**

**// Here, `min` and `max` denote the minimum and maximum element in the subarray.**

**bool isConsecutive(int A[], int i, int j, int min, int max)**

**{**

**// for an array to contain consecutive integers, the difference**

**// between the maximum and minimum element in it should be exactly `j-i`**

**if (max - min != j - i) {**

**return false;**

**}**

**// create a visited array (we can also use a set)**

**vector<bool> visited(j - i + 1);**

**// traverse the subarray and check if each element appears only once**

**for (int k = i; k <= j; k++)**

**{**

**// if the element is seen before, return false**

**if (visited[A[k] - min]) {**

**return false;**

**}**

**// mark the element as seen**

**visited[A[k] - min] = true;**

**}**

**// we reach here when all elements in the array are distinct**

**return true;**

**}**

**// Find the largest subarray formed by consecutive integers**

**void findMaxSubarray(int A[], int n)**

**{**

**int len = 1;**

**int start = 0, end = 0;**

**// consider each subarray formed by `A[i…j]`**

**// `i` denotes the beginning of the subarray**

**for (int i = 0; i < n - 1; i++)**

**{**

**// stores the minimum and maximum element in subarray `A[i…j]`**

**int min\_val = A[i], max\_val = A[i];**

**// `j` denotes the end of the subarray**

**for (int j = i + 1; j < n; j++)**

**{**

**// update the minimum and maximum elements of the subarray**

**min\_val = min(min\_val, A[j]);**

**max\_val = max(max\_val, A[j]);**

**// check if subarray `A[i…j]` is formed by consecutive integers**

**if (isConsecutive(A, i, j, min\_val, max\_val))**

**{**

**if (len < max\_val - min\_val + 1)**

**{**

**len = max\_val - min\_val + 1,**

**start = i, end = j;**

**}**

**}**

**}**

**}**

**// print the maximum length subarray**

**cout << "The largest subarray is [" << start << ", " << end << "]";**

**}**

**int main()**

**{**

**int A[] = { 2, 0, 2, 1, 4, 3, 1, 0 };**

**int n = sizeof(A) / sizeof(A[0]);**

**findMaxSubarray(A, n);**

**return 0;**

**}**

**40: Replace every array element with the product of every other element without using a division operator**

**Given an integer array, replace each element with the product of every other element without using the division operator.  
Input: { 1, 2, 3, 4, 5 }**

**Output: { 120, 60, 40, 30, 24 }**

**Input: { 5, 3, 4, 2, 6, 8 }**

**Output: { 1152, 1920, 1440, 2880, 960, 720 }**

| **#include <stdio.h>**    **// Function to replace each array element with every other**  **// element's product without using the division operator**  **void findProduct(int arr[], int n)**  **{**  **// base case**  **if (n == 0) {**  **return;**  **}**    **// use two auxiliary arrays**  **int left[n], right[n];**    **// `left[i]` stores the product of all elements in subarray[0…i-1]**  **left[0] = 1;**  **for (int i = 1; i < n; i++) {**  **left[i] = arr[i - 1] \* left[i - 1];**  **}**    **// `right[i]` stores the product of all elements in subarray[i+1…n-1]**  **right[n - 1] = 1;**  **for (int j = n - 2; j >= 0; j--) {**  **right[j] = arr[j + 1] \* right[j + 1];**  **}**    **// replace each element with the product of its left and right subarray**  **for (int i = 0; i < n; i++) {**  **arr[i] = left[i] \* right[i];**  **}**  **}**    **int main(void)**  **{**  **int arr[] = { 5, 3, 4, 2, 6, 8 };**  **int n = sizeof(arr) / sizeof(arr[0]);**    **findProduct(arr, n);**    **// print the modified array**  **for (int i = 0; i < n; i++) {**  **printf("%d ", arr[i]);**  **}**    **return 0;**  **}** |
| --- |

**41: Longest Bitonic Subarray Problem**

**The Longest Bitonic Subarray (LBS) problem is to find a subarray of a given sequence in which the subarray’s elements are first sorted in increasing order, then in decreasing order, and the subarray is as long as possible. Strictly ascending or descending subarrays are also accepted.  
Longest bitonic subarray of the sequence { 3, 5, 8, 4, 5, 9, 10, 8, 5, 3, 4 } is { 4, 5, 9, 10, 8, 5, 3 }**

**For sequences sorted in increasing or decreasing order, the output is the same as the input sequence, i.e.,**

**[1, 2, 3, 4, 5] ——> [1, 2, 3, 4, 5]**

**[5, 4, 3, 2, 1] ——> [5, 4, 3, 2, 1]**

**#include <stdio.h>**

**// Function to find the length of the longest bitonic subarray in an array**

**void findBitonicSubarray(int arr[], int n)**

**{**

**if (n == 0) {**

**return;**

**}**

**// `I[i]` store the length of the longest increasing subarray,**

**// ending at `arr[i]`**

**int I[n];**

**I[0] = 1;**

**for (int i = 1; i < n; i++)**

**{**

**I[i] = 1;**

**if (arr[i - 1] < arr[i]) {**

**I[i] = I[i - 1] + 1;**

**}**

**}**

**// `D[i]` store the length of the longest decreasing subarray,**

**// starting with `arr[i]`**

**int D[n];**

**D[n - 1] = 1;**

**for (int i = n - 2; i >= 0; i--)**

**{**

**D[i] = 1;**

**if (arr[i] > arr[i + 1]) {**

**D[i] = D[i + 1] + 1;**

**}**

**}**

**// consider each element as a peak and calculate LBS**

**int lbs\_len = 1;**

**int beg = 0, end = 0;**

**for (int i = 0; i < n; i++)**

**{**

**if (lbs\_len < I[i] + D[i] - 1)**

**{**

**lbs\_len = I[i] + D[i] - 1;**

**beg = i - I[i] + 1;**

**end = i + D[i] - 1;**

**}**

**}**

**// print the longest bitonic subarray**

**printf("The length of the longest bitonic subarray is %d\n", lbs\_len);**

**printf("The longest bitonic subarray indices is [%d, %d]", beg, end);**

**}**

**int main(void)**

**{**

**int A[] = { 3, 5, 8, 4, 5, 9, 10, 8, 5, 3, 4 };**

**int n = sizeof(A) / sizeof(A[0]);**

**findBitonicSubarray(A, n);**

**return 0;**

**}**

**42: Find the maximum difference between two array elements that satisfies the given constraints**

**Given an integer array, find the maximum difference between two elements in it such that the smaller element appears before the larger element.  
Input: { 2, 7, 9, 5, 1, 3, 5 }**

**Output: The maximum difference is 7.**

**The pair is (2, 9)**

| **#include <stdio.h>**  **#include <limits.h>**    **// Utility function to find a maximum of two numbers**  **int max(int x, int y) {**  **return (x > y) ? x : y;**  **}**    **// Naive function to find the maximum difference between two elements in**  **// an array such that the smaller element appears before the larger element**  **int getMaxDiff(int arr[], int n)**  **{**  **int diff = INT\_MIN;**    **if (n == 0) {**  **return diff;**  **}**    **for (int i = 0; i < n - 1; i++) {**  **for (int j = i + 1; j < n; j++) {**  **if (arr[j] > arr[i]) {**  **diff = max(diff, arr[j] - arr[i]);**  **}**  **}**  **}**    **return diff;**  **}**    **int main()**  **{**  **int arr[] = { 2, 7, 9, 5, 1, 3, 5 };**  **int n = sizeof(arr) / sizeof(arr[0]);**    **int result = getMaxDiff(arr, n);**  **if (result != INT\_MIN) {**  **printf("The maximum difference is %d", result);**  **}**    **return 0;**  **}** |
| --- |

**43: Print continuous subarray with maximum sum**

**Given an integer array, find and print a contiguous subarray with the maximum sum in it.  
Input: {-2, 1, -3, 4, -1, 2, 1, -5, 4}**

**Output: The contiguous subarray with the largest sum is {4, -1, 2, 1}**

**Input: {8, -7, -3, 5, 6, -2, 3, -4, 2}**

**Output: The contiguous subarray with the largest sum is {5, 6, -2, 3}**

**#include <iostream>**

**#include <climits>**

**using namespace std;**

**// Function to print contiguous subarray with the largest sum**

**// in a given set of integers**

**void kadane(int arr[], int n)**

**{**

**// base case**

**if (n <= 0) {**

**return;**

**}**

**// stores maximum sum subarray found so far**

**int max\_so\_far = INT\_MIN;**

**// stores the maximum sum of subarray ending at the current position**

**int max\_ending\_here = 0;**

**// stores endpoints of maximum sum subarray found so far**

**int start = 0, end = 0;**

**// stores starting index of a positive-sum sequence**

**int beg = 0;**

**// traverse the given array**

**for (int i = 0; i < n; i++)**

**{**

**// update the maximum sum of subarray "ending" at index `i`**

**max\_ending\_here = max\_ending\_here + arr[i];**

**// if the maximum sum becomes less than the current element,**

**// start from the current element**

**if (max\_ending\_here < arr[i])**

**{**

**max\_ending\_here = arr[i];**

**beg = i;**

**}**

**// update result if the current subarray sum is found to be greater**

**if (max\_so\_far < max\_ending\_here)**

**{**

**max\_so\_far = max\_ending\_here;**

**start = beg;**

**end = i;**

**}**

**}**

**cout << "The contiguous subarray with the largest sum is ";**

**for (int i = start; i <= end; i++) {**

**cout << arr[i] << " ";**

**}**

**}**

**int main()**

**{**

**int arr[] = { -2, 1, -3, 4, -1, 2, 1, -5, 4 };**

**int n = sizeof(arr)/sizeof(arr[0]);**

**kadane(arr, n);**

**return 0;**

**}**

**44: Find all distinct combinations of a given length – I**

**Given an integer array, find all distinct combinations of a given length k.  
Input: {2, 3, 4}, k = 2**

**Output: {2, 3}, {2, 4}, {3, 4}**

**Input: {1, 2, 1}, k = 2**

**Output: {1, 2}, {1, 1}, {2, 1}**

| **#include <iostream>**  **#include <set>**  **#include <vector>**  **#include <experimental/iterator>**  **using namespace std;**    **// Function to print all distinct combinations of length `k`**  **void findCombinations(vector<int> const &arr, int i, int k,**  **set<vector<int>> &subarrays, vector<int> &out)**  **{**  **// invalid input**  **if (arr.size() == 0 || k > arr.size()) {**  **return;**  **}**    **// base case: combination size is `k`**  **if (k == 0) {**  **subarrays.insert(out);**  **return;**  **}**    **// start from the next index till the last index**  **for (int j = i; j < arr.size(); j++)**  **{**  **// add current element `arr[j]` to the solution and recur for next index**  **// `j+1` with one less element `k-1`**  **out.push\_back(arr[j]);**  **findCombinations(arr, j + 1, k - 1, subarrays, out);**  **out.pop\_back(); // backtrack**  **}**  **}**    **template <typename T>**  **void printVector(vector<T> const &input)**  **{**  **cout << "[";**  **copy(begin(input),**  **end(input),**  **experimental::make\_ostream\_joiner(cout, ", "));**  **cout << "]\n";**  **}**    **int main()**  **{**  **vector<int> arr = { 1, 2, 3 };**  **int k = 2;**    **// set to store all combinations**  **set<vector<int>> subarrays;**    **// vector to store a combination**  **vector<int> out;**    **// process elements from left to right**  **findCombinations(arr, 0, k, subarrays, out);**    **// print subarrays**  **for (auto const &vec: subarrays) {**  **printVector(vec);**  **}**    **return 0;**  **}** |
| --- |

**45: Find all distinct combinations of a given length – II**

**Given an integer array, find all distinct combinations of a given length k  
Input: {2, 3, 4}, k = 2**

**Output: {2, 3}, {2, 4}, {3, 4}**

**Input: {1, 2, 1}, k = 2**

**Output: {1, 2}, {1, 1}, {2, 1}**

| **#include <iostream>**  **#include <set>**  **#include <vector>**  **#include <experimental/iterator>**  **using namespace std;**    **// Function to print all distinct combinations of length `k`**  **void findCombinations(vector<int> const &arr, int i, int k,**  **set<vector<int>> &subarrays, vector<int> &out)**  **{**  **// do nothing for empty input**  **if (arr.size() == 0) {**  **return;**  **}**    **// base case: combination size is `k`**  **if (k == 0) {**  **subarrays.insert(out);**  **return;**  **}**    **// return if no more elements are left**  **if (i == arr.size()) {**  **return;**  **}**    **// include the current element in the current combination and recur**  **out.push\_back(arr[i]);**  **findCombinations(arr, i + 1, k - 1, subarrays, out);**    **// exclude the current element from the current combination**  **out.pop\_back(); // backtrack**    **// exclude the current element from the current combination and recur**  **findCombinations(arr, i + 1, k, subarrays, out);**  **}**    **template <typename T>**  **void printVector(vector<T> const &input)**  **{**  **cout << "[";**  **copy(begin(input),**  **end(input),**  **experimental::make\_ostream\_joiner(cout, ", "));**  **cout << "]\n";**  **}**    **int main()**  **{**  **vector<int> arr = { 1, 2, 3 };**  **int k = 2;**    **// set to store all combinations**  **set<vector<int>> subarrays;**    **// vector to store a combination**  **vector<int> out;**    **// process elements from left to right**  **findCombinations(arr, 0, k, subarrays, out);**    **// print subarrays**  **for (auto const &vec: subarrays) {**  **printVector(vec);**  **}**    **return 0;**  **}** |
| --- |

**46: Find the maximum sequence of continuous 1’s formed by replacing at-most `k` zeroes by ones**

**Given a binary array, find the maximum sequence of continuous 1’s that can be formed by replacing at most k zeroes by ones.  
Input: A[] = { 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0 }**

**For k = 0,**

**The length of the longest sequence is 4 (from index 6 to 9)**

**For k = 1,**

**The length of the longest sequence is 7 (from index 3 to 9)**

**For k = 2,**

**The length of the longest sequence is 10 (from index 0 to 9)**

**For k = 3,**

**The length of the longest sequence is 11 (from index 0 to 10)**

**#include <stdio.h>**

**// Function to find the maximum sequence of continuous 1's by replacing**

**// at most `k` zeroes by 1 using sliding window technique**

**void findLongestSequence(int arr[], int n, int k)**

**{**

**int left = 0; // represents the current window's starting index**

**int count = 0; // stores the total number of zeros in the current window**

**int window = 0; // stores the maximum number of continuous 1's found**

**// so far (including `k` zeroes)**

**int leftIndex = 0; // stores the left index of maximum window found so far**

**// maintain a window `[left…right]` containing at most `k` zeroes**

**for (int right = 0; right < n; right++)**

**{**

**// if the current element is 0, increase the count of zeros in the**

**// current window by 1**

**if (arr[right] == 0) {**

**count++;**

**}**

**// the window becomes unstable if the total number of zeros in it becomes**

**// more than `k`**

**while (count > k)**

**{**

**// if we have found zero, decrement the number of zeros in the**

**// current window by 1**

**if (arr[left] == 0) {**

**count--;**

**}**

**// remove elements from the window's left side till the window**

**// becomes stable again**

**left++;**

**}**

**// when we reach here, window `[left…right]` contains at most**

**// `k` zeroes, and we update max window size and leftmost index**

**// of the window**

**if (right - left + 1 > window)**

**{**

**window = right - left + 1;**

**leftIndex = left;**

**}**

**}**

**// no sequence found**

**if (window == 0) {**

**return;**

**}**

**// print the maximum sequence of continuous 1's**

**printf("The longest sequence has length %d from index %d to %d",**

**window, leftIndex, (leftIndex + window - 1));**

**}**

**int main()**

**{**

**int arr[] = { 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0 };**

**int k = 2;**

**int n = sizeof(arr) / sizeof(arr[0]);**

**findLongestSequence(arr, n, k);**

**return 0;**

**}**

**47: Longest Increasing Subsequence using LCS**

**The longest increasing subsequence problem is to find a subsequence of a given sequence in which the subsequence’s elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible. This subsequence is not necessarily contiguous or unique.**

**For example, the longest increasing subsequence is [0, 2, 6, 9, 11, 15] in the following subsequence:  
[0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15]  
  
This subsequence has length 6; the input sequence has no 7–member increasing subsequences. The longest increasing subsequence in this example is not unique. For instance, [0, 4, 6, 9, 11, 15] or [0, 4, 6, 9, 13, 15] are other increasing subsequences of equal length in the same input sequence.**

**#include <iostream>**

**#include <vector>**

**#include <unordered\_map>**

**#include <string>**

**#include <algorithm>**

**using namespace std;**

**// Function to find the length of the longest common subsequence of**

**// array `X[0…m-1]` and `Y[0…n-1]`**

**int LCSLength(vector<int> const &X, vector<int> const &Y, int m, int n,**

**unordered\_map<string, int> &lookup)**

**{**

**// return if the end of either array is reached**

**if (m == 0 || n == 0) {**

**return 0;**

**}**

**// construct a unique map key from dynamic elements of the input**

**string key = to\_string(m) + "|" + to\_string(n);**

**// if the subproblem is seen for the first time, solve it and**

**// store its result in a map**

**if (lookup.find(key) == lookup.end())**

**{**

**// if the last element of `X` and `Y` matches**

**if (X[m - 1] == Y[n - 1]) {**

**lookup[key] = LCSLength(X, Y, m - 1, n - 1, lookup) + 1;**

**}**

**else {**

**// otherwise, if the last element of `X` and `Y` don't match**

**lookup[key] = max(LCSLength(X, Y, m, n - 1, lookup),**

**LCSLength(X, Y, m - 1, n, lookup));**

**}**

**}**

**// return the subproblem solution from the map**

**return lookup[key];**

**}**

**// Function to remove duplicates from a sorted array**

**int removeDuplicates(vector<int> &Y)**

**{**

**int k = 0;**

**for (int i = 1; i < Y.size(); i++)**

**{**

**if (Y[i] != Y[k]) {**

**Y[++k] = Y[i];**

**}**

**}**

**// return length of subarray containing all distinct characters**

**return k + 1;**

**}**

**// Iterative function to find the length of the longest increasing subsequence (LIS)**

**// of a given array using the longest common subsequence (LCS)**

**int findLIS(vector<int> const &X)**

**{**

**int n = X.size();**

**// base case**

**if (n == 0) {**

**return 0;**

**}**

**// create a copy of the original array**

**vector<int> Y(X);**

**// sort the copy of the original array**

**sort(Y.begin(), Y.end());**

**// remove all the duplicates from `Y`**

**int m = removeDuplicates(Y);**

**// create a map to store solutions to subproblems**

**unordered\_map<string, int> lookup;**

**// return LCS of both**

**return LCSLength(X, Y, n, m, lookup);**

**}**

**int main()**

**{**

**vector<int> X = { 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15 };**

**cout << "The length of the LIS is " << findLIS(X);**

**return 0;**

**}**

**48: Find minimum sum subarray of size `k`**

**Given an integer array, find the minimum sum subarray of size k, where k is a positive integer  
Input: {10, 4, 2, 5, 6, 3, 8, 1}, k = 3**

**Output: Minimum sum subarray of size 3 is (1, 3)**

**#include <stdio.h>**

**#include <limits.h>**

**// Find the minimum sum subarray of a given size `k`**

**void findSubarray(int arr[], int n, int k)**

**{**

**// base case**

**if (n == 0 || n <= k) {**

**return;**

**}**

**// stores the sum of elements in the current window**

**int window\_sum = 0;**

**// stores the sum of minimum sum subarray found so far**

**int min\_window = INT\_MAX;**

**// stores ending index of the minimum sum subarray found so far**

**int last = 0;**

**for (int i = 0; i < n; i++)**

**{**

**// add the current element to the window**

**window\_sum += arr[i];**

**// if the window size is more than equal to `k`**

**if (i + 1 >= k)**

**{**

**// update the minimum sum window**

**if (min\_window > window\_sum)**

**{**

**min\_window = window\_sum;**

**last = i;**

**}**

**// remove a leftmost element from the window**

**window\_sum -= arr[i + 1 - k];**

**}**

**}**

**printf("The minimum sum subarray is (%d, %d)", last - k + 1, last);**

**}**

**int main(void)**

**{**

**int arr[] = { 10, 4, 2, 5, 6, 3, 8, 1 };**

**int k = 3;**

**int n = sizeof(arr)/sizeof(arr[0]);**

**findSubarray(arr, n, k);**

**return 0;**

**}**

**49: Find a subarray having the given sum in an integer array**

**Given an integer array, find a subarray having a given sum in it.  
Input: nums[] = {2, 6, 0, 9, 7, 3, 1, 4, 1, 10}, target = 15**

**Output: {6, 0, 9}**

**Input: nums[] = {0, 5, -7, 1, -4, 7, 6, 1, 4, 1, 10}, target = 15**

**Output: {1, -4, 7, 6, 1, 4} or {4, 1, 10}**

**Input: nums[] = {0, 5, -7, 1, -4, 7, 6, 1, 4, 1, 10}, target = -3**

**Output: {1, -4}**

| **#include <stdio.h>**    **// Function to print subarray having a given sum using**  **// sliding window technique**  **void findSubarray(int nums[], int n, int target)**  **{**  **// maintains the sum of the current window**  **int windowSum = 0;**    **// maintain a window [low, high-1]**  **int low = 0, high = 0;**    **// consider every subarray starting from the `low` index**  **for (low = 0; low < n; low++)**  **{**  **// if the current window's sum is less than the given sum,**  **// then add elements to the current window from the right**  **while (windowSum < target && high < n)**  **{**  **windowSum += nums[high];**  **high++;**  **}**    **// if the current window's sum is equal to the given sum**  **if (windowSum == target)**  **{**  **printf("Subarray found [%d–%d]\n", low, high - 1);**  **return;**  **}**    **// At this point, the current window's sum is more than the given sum.**  **// Remove the current element (leftmost element) from the window**  **windowSum -= nums[low];**  **}**  **}**    **int main(void)**  **{**  **// an array of positive integers**  **int nums[] = { 2, 6, 0, 9, 7, 3, 1, 4, 1, 10 };**  **int target = 15;**    **int n = sizeof(nums)/sizeof(nums[0]);**    **findSubarray(nums, n, target);**    **return 0;**  **}** |
| --- |

**50: Find the smallest subarray length whose sum of elements is greater than `k`**

**Given an array of positive integers, find the smallest subarray’s length whose sum of elements is greater than a given number k.  
Input: {1, 2, 3, 4, 5, 6, 7, 8}, k = 20**

**Output: The smallest subarray length is 3**

**Explanation: The smallest subarray with sum > 20 is {6, 7, 8}**

**Input: {1, 2, 3, 4, 5, 6, 7, 8}, k = 7**

**Output: The smallest subarray length is 1**

**Explanation: The smallest subarray with sum > 7 is {8}**

**Input: {1, 2, 3, 4, 5, 6, 7, 8}, k = 21**

**Output: The smallest subarray length is 4**

**Explanation: The smallest subarray with sum > 21 is {5, 6, 7, 8}  
Input: {1, 2, 3, 4, 5, 6, 7, 8}, k = 40**

**Output: No subarray exists**

| **#include <stdio.h>**  **#include <limits.h>**    **// Utility function to find a minimum of two numbers**  **int min(int x, int y) {**  **return (x < y) ? x : y;**  **}**    **// Function to find the length of the smallest subarray whose sum**  **// of elements is greater than the given number**  **int findSmallestSubarrayLen(int arr[], int n, int k)**  **{**  **// stores the current window sum**  **int windowSum = 0;**    **// stores the result**  **int len = INT\_MAX;**    **// stores the window's starting index**  **int left = 0;**    **// maintain a sliding window `[left…right]`**  **for (int right = 0; right < n; right++)**  **{**  **// include the current element in the window**  **windowSum += arr[right];**    **// the window becomes unstable if its sum becomes more than `k`**  **while (windowSum > k && left <= right)**  **{**  **// update the result if the current window's length is less than the**  **// minimum found so far**  **len = min(len, right - left + 1);**    **// remove elements from the window's left side till the window**  **// becomes stable again**  **windowSum -= arr[left];**  **left++;**  **}**  **}**    **// invalid input**  **if (len == INT\_MAX) {**  **return 0;**  **}**    **// return result**  **return len;**  **}**    **int main()**  **{**  **// an array of positive numbers**  **int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8 };**  **int k = 21;**    **int n = sizeof(arr) / sizeof(arr[0]);**    **// find the length of the smallest subarray**  **int len = findSmallestSubarrayLen(arr, n, k);**    **if (len != INT\_MAX) {**  **printf("The smallest subarray length is %d", len);**  **}**  **else {**  **printf("No subarray exists");**  **}**    **return 0;**  **}** |
| --- |

**51: Find the smallest window in an array sorting which will make the entire array sorted**

**Given an integer array, find the smallest window sorting which will make the entire array sorted in increasing order.  
Input: { 1, 2, 3, 7, 5, 6, 4, 8 }**

**Output: Sort the array from index 3 to 6**

**Input: { 1, 3, 2, 7, 5, 6, 4, 8 }**

**Output: Sort the array from index 1 to 6**

**#include <iostream>**

**#include <climits>**

**using namespace std;**

**// Function to find the smallest window in an array, sorting which will**

**// make the entire array sorted**

**void findSubarray(int arr[], int n)**

**{**

**int leftIndex = -1, rightIndex = -1;**

**// traverse from left to right and keep track of maximum so far**

**int max\_so\_far = INT\_MIN;**

**for (int i = 0; i < n; i++)**

**{**

**if (max\_so\_far < arr[i]) {**

**max\_so\_far = arr[i];**

**}**

**// find the last position that is less than the maximum so far**

**if (arr[i] < max\_so\_far) {**

**rightIndex = i;**

**}**

**}**

**// traverse from right to left and keep track of the minimum so far**

**int min\_so\_far = INT\_MAX;**

**for (int i = n - 1; i >= 0; i--)**

**{**

**if (min\_so\_far > arr[i]) {**

**min\_so\_far = arr[i];**

**}**

**// find the last position that is more than the minimum so far**

**if (arr[i] > min\_so\_far) {**

**leftIndex = i;**

**}**

**}**

**if (leftIndex == -1) {**

**cout << "Array is already sorted";**

**return;**

**}**

**cout << "Sort array from index " << leftIndex << " to " << rightIndex;**

**}**

**int main()**

**{**

**int arr[] = { 1, 3, 2, 7, 5, 6, 4, 8 };**

**int n = sizeof(arr)/sizeof(arr[0]);**

**findSubarray(arr, n);**

**return 0;**

**}**

**52: Find maximum sum path involving elements of given arrays**

**Given two sorted arrays of integers, find a maximum sum path involving elements of both arrays whose sum is maximum. We can start from either array, but we can switch between arrays only through its common elements.  
Input:**

**X = { 3, 6, 7, 8, 10, 12, 15, 18, 100 }**

**Y = { 1, 2, 3, 5, 7, 9, 10, 11, 15, 16, 18, 25, 50 }**

**The maximum sum path is:**

**1 —> 2 —> 3 —> 6 —> 7 —> 9 —> 10 —> 12 —> 15 —> 16 —> 18 —> 100**

**The maximum sum is 199**

| **#include <stdio.h>**    **// Utility function to find the minimum of two integers**  **int max (int x, int y) {**  **return (x > y) ? x : y;**  **}**    **// Function to find the maximum sum path in two given arrays.**  **// The code is similar to the merge routine of the merge sort algorithm**  **int findMaxSum(int X[], int Y[], int m, int n)**  **{**  **int sum = 0;**  **int sum\_x = 0, sum\_y = 0;**    **// `i` and `j` denotes the current index of `X` and `Y`, respectively**  **int i = 0, j = 0;**    **// loop till `X` and `Y` are empty**  **while (i < m && j < n)**  **{**  **// to handle the duplicate elements in `X`**  **while (i < m-1 && X[i] == X[i+1]) {**  **sum\_x += X[i++];**  **}**    **// to handle the duplicate elements in `Y`**  **while (j < n-1 && Y[j] == Y[j+1]) {**  **sum\_y += Y[j++];**  **}**    **// if the current element of `Y` is less than the current element of `X`**  **if (Y[j] < X[i])**  **{**  **sum\_y += Y[j];**  **j++;**  **}**    **// if the current element of `X` is less than the current element of `Y`**  **else if (X[i] < Y[j])**  **{**  **sum\_x += X[i];**  **i++;**  **}**    **else // if (X[i] == Y[j])**  **{**  **// consider the maximum sum and include the current cell's value**  **sum += max(sum\_x, sum\_y) + X[i];**    **// move both indices by 1 position**  **i++, j++;**    **// reset both sums**  **sum\_x = 0, sum\_y = 0;**  **}**  **}**    **// process the remaining elements of `X` (if any)**  **while (i < m) {**  **sum\_x += X[i++];**  **}**    **// process the remaining elements of `Y` (if any)**  **while (j < n) {**  **sum\_y += Y[j++];**  **}**    **sum += max(sum\_x, sum\_y);**  **return sum;**  **}**    **int main()**  **{**  **int X[] = { 3, 6, 7, 8, 10, 12, 15, 18, 100 };**  **int Y[] = { 1, 2, 3, 5, 7, 9, 10, 11, 15, 16, 18, 25, 50 };**    **int m = sizeof(X)/sizeof(X[0]);**  **int n = sizeof(Y)/sizeof(Y[0]);**    **printf("The maximum sum is %d", findMaxSum(X, Y, m, n));**    **return 0;**  **}** |
| --- |

**53: Find maximum profit earned by buying and selling shares any number of times**

**Given a list containing future prediction of share prices, find the maximum profit earned by buying and selling shares any number of times with the constraint, a new transaction can only start after the previous transaction is complete, i.e., we can only hold at most one share at a time.  
Stock Prices: {1, 5, 2, 3, 7, 6, 4, 5}**

**Total profit earned is 10**

**Buy on day 1 and sell on day 2**

**Buy on day 3 and sell on day 5**

**Buy on day 7 and sell on day 8**

**Stock Prices: {10, 8, 6, 5, 4, 2}**

**Total profit earned is 0**

| **#include <stdio.h>**    **// Function to find the maximum profit earned by buying and**  **// selling shares any number of times**  **int findMaxProfit(int price[], int n)**  **{**  **// keep track of the maximum profit gained**  **int profit = 0;**    **// initialize the local minimum to the first element's index**  **int j = 0;**    **// start from the second element**  **for (int i = 1; i < n; i++)**  **{**  **// update the local minimum if a decreasing sequence is found**  **if (price[i - 1] > price[i]) {**  **j = i;**  **}**    **// sell shares if the current element is the peak,**  **// i.e., (`previous <= current > next`)**  **if (price[i - 1] <= price[i] &&**  **(i + 1 == n || price[i] > price[i + 1]))**  **{**  **profit += (price[i] - price[j]);**  **printf("Buy on day %d and sell on day %d\n", j + 1, i + 1);**  **}**  **}**    **return profit;**  **}**    **int main()**  **{**  **int price[] = { 1, 5, 2, 3, 7, 6, 4, 5 };**  **int n = sizeof(price) / sizeof(price[0]);**    **printf("\nTotal profit earned is %d", findMaxProfit(price, n));**    **return 0;**  **}** |
| --- |

**54: Find minimum platforms needed to avoid delay in the train arrival**

**Given a schedule containing the arrival and departure time of trains in a station, find the minimum number of platforms needed to avoid delay in any train’s arrival.  
Trains arrival = { 2.00, 2.10, 3.00, 3.20, 3.50, 5.00 }**

**Trains departure = { 2.30, 3.40, 3.20, 4.30, 4.00, 5.20 }**

**The minimum platforms needed is 2**

**The train arrived at 2.00 on platform 1**

**The train arrived at 2.10 on platform 2**

**The train departed at 2.30 from platform 1**

**The train arrived at 3.00 on platform 1**

**The train departed at 3.20 from platform 1**

**The train arrived at 3.20 on platform 1**

**The train departed at 3.40 from platform 2**

**The train arrived at 3.50 on platform 2**

**The train departed at 4.00 from platform 2**

**The train departed at 4.30 from platform 1**

**The train arrived at 5.00 on platform 1**

**The train departed at 5.20 from platform 1**

**#include <iostream>**

**#include <vector>**

**#include <algorithm>**

**using namespace std;**

**typedef vector<double> Trains;**

**// Function to find the minimum number of platforms needed**

**// to avoid delay in any train arrival**

**int findMinPlatforms(Trains arrival, Trains departure) // no-ref, no-const**

**{**

**// sort arrival time of trains**

**sort(arrival.begin(), arrival.end());**

**// sort departure time of trains**

**sort(departure.begin(), departure.end());**

**// maintains the count of trains**

**int count = 0;**

**// stores minimum platforms needed**

**int platforms = 0;**

**// take two indices for arrival and departure time**

**int i = 0, j = 0;**

**// run till all trains have arrived**

**while (i < arrival.size())**

**{**

**// if a train is scheduled to arrive next**

**if (arrival[i] < departure[j])**

**{**

**// increase the count of trains and update minimum**

**// platforms if required**

**platforms = max(platforms, ++count);**

**// move the pointer to the next arrival**

**i++;**

**}**

**// if the train is scheduled to depart next i.e.**

**// `departure[j] < arrival[i]`, decrease trains' count**

**// and move pointer `j` to the next departure.**

**// If two trains are arriving and departing simultaneously, i.e.**

**// `arrival[i] == departure[j]`, depart the train first**

**else {**

**count--, j++;**

**}**

**}**

**return platforms;**

**}**

**int main()**

**{**

**Trains arrival = { 2.00, 2.10, 3.00, 3.20, 3.50, 5.00 };**

**Trains departure = { 2.30, 3.40, 3.20, 4.30, 4.00, 5.20 };**

**cout << "The minimum platforms needed is " << findMinPlatforms(arrival, departure);**

**return 0;**

**}**

**55: Decode an array constructed from another array**

**Given an array constructed from another array A by taking the sum of every distinct pair in it, decode it to get the original array A back.  
Input: { 3, 4, 5, 5, 6, 7 }**

**Output: { 1, 2, 3, 4 }**

**Input: { 3, 4, 5, 6, 5, 6, 7, 7, 8, 9 }**

**Output: { 1, 2, 3, 4, 5 }**

**Input: { 3 }**

**Output: { 1, 2 } or { 2, 1 } or { 0, 3 } or { 3, 0 }**

**Input: { 3, 4, 5 }**

**Output: { 1, 2, 3 }**

**#include <iostream>**

**#include <cmath>**

**using namespace std;**

**// Function to decode given array to get back the original array elements**

**void decode(int inp[], int m)**

**{**

**// base case**

**if (m == 0 || m == 2) {**

**return;**

**}**

**// calculate the size of the original array**

**int n = (sqrt(8\*m + 1) + 1) / 2;**

**// create an auxiliary array of size `n` to store elements**

**// of the original array**

**int A[n];**

**// calculate the first element of the original array**

**if (n == 1 || m == 1) {**

**A[0] = inp[0];**

**}**

**else if (n == 2) {**

**A[0] = inp[0] - inp[1];**

**}**

**else {**

**A[0] = (inp[0] + inp[1] - inp[n - 1]) / 2;**

**}**

**// calculate the remaining elements of the original array using**

**// the first element**

**for (int i = 1; i < n; i++) {**

**A[i] = inp[i - 1] - A[0];**

**}**

**// print the original array**

**for (int i = 0; i < n; i++) {**

**cout << A[i] << " ";**

**}**

**}**

**int main()**

**{**

**int inp[] = { 3, 4, 5, 6, 5, 6, 7, 7, 8, 9 };**

**int m = sizeof(inp)/sizeof(inp[0]);**

**decode(inp, m);**

**return 0;**

**}**

**56: Find a triplet with the given sum in an array**

**Given an unsorted integer array, find a triplet with a given sum in it.**

**Input:**

**nums = [ 2, 7, 4, 0, 9, 5, 1, 3 ]**

**target = 6**

**Output: Triplet exists.**

**The triplets with the given sum 6 are {0, 1, 5}, {0, 2, 4}, {1, 2, 3}**

**#include <iostream>**

**using namespace std;**

**// Naive recursive function to check if triplet exists in an array**

**// with the given sum**

**bool isTripletExist(int nums[], int n, int target, int count)**

**{**

**// if triplet has the desired sum, return true**

**if (count == 3 && target == 0) {**

**return true;**

**}**

**// return false if the sum is not possible with the current configuration**

**if (count == 3 || n == 0 || target < 0) {**

**return false;**

**}**

**// recur with including and excluding the current element**

**return isTripletExist(nums, n - 1, target - nums[n - 1], count + 1) ||**

**isTripletExist(nums, n - 1, target, count);**

**}**

**int main()**

**{**

**int nums[] = { 2, 7, 4, 0, 9, 5, 1, 3 };**

**int target = 6;**

**int n = sizeof(nums) / sizeof(nums[0]);**

**isTripletExist(nums, n, target, 0) ? cout << "Triplet exists":**

**cout << "Triplet doesn't exist";**

**return 0;**

**}**

**57: Reverse every consecutive `m`-elements of a subarray**

**Given an array, reverse every group of consecutive m elements in a given subarray of it.  
Consider the below array.**

**A[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },**

**m = 3**

**Then for subarray [i, j], where i and j is**

**Input: i = 1, j = 7 or 8**

**Output: [1, 4, 3, 2, 7, 6, 5, 8, 9, 10]**

**Input: i = 1, j = 9**

**Output: [1, 4, 3, 2, 7, 6, 5, 10, 9, 8]**

**Input: i = 3, j = 5**

**Output: [1, 2, 3, 6, 5, 4, 7, 8, 9, 10]**

**Input: i = 3, j = 4**

**Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]**

| **#include <stdio.h>**    **// Utility function to swap elements `A[i]` and `A[j]` in an array**  **void swap(int A[], int i, int j)**  **{**  **int temp = A[i];**  **A[i] = A[j];**  **A[j] = temp;**  **}**    **// Utility function to find a minimum of two numbers**  **int min(int x, int y) {**  **return (x < y) ? x : y;**  **}**    **// Utility function to reverse subarray `arr[i, j]`**  **void reverse\_subarray(int arr[], int i, int j)**  **{**  **while (i < j)**  **{**  **swap(arr, i, j);**  **i++, j--;**  **}**  **}**    **// Function to reverse every consecutive `m` elements of**  **// subarray `arr[beg, end]`**  **void reverse(int arr[], int beg, int end, int m)**  **{**  **// base case**  **if (m <= 1) {**  **return;**  **}**    **// return if the subarray length is less than `m`**  **if (m > end - beg + 1) {**  **return;**  **}**    **// reverse every consecutive `m` elements**  **for (int i = beg; i <= end; i = i + m)**  **{**  **// check if subarray length is at least `m`**  **if (i + m - 1 <= end) {**  **reverse\_subarray(arr, i, i + m - 1);**  **}**  **}**  **}**    **// Utility function to print given array**  **void printArray(int arr[], int n)**  **{**  **for (int i = 0; i < n; i++) {**  **printf("%d ", arr[i]);**  **}**  **}**    **int main()**  **{**  **int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };**  **int m = 3;**  **int beg = 1, end = 8;**    **int n = sizeof(arr) / sizeof(arr[0]);**    **// reverse the array**  **reverse(arr, beg, min(end, n - 1), m);**    **// print the modified array**  **printArray(arr, n);**    **return 0;**  **}** |
| --- |

**58: 4–Sum Problem | Quadruplets with a given sum  
4-sum problem: Given an unsorted integer array, check if it contains four elements tuple (quadruplets) having a given sum.  
Input:**

**nums = [ 2, 7, 4, 0, 9, 5, 1, 3 ]**

**target = 20**

**Output: Quadruplet exists.**

**Below are quadruplets with the given sum 20**

**(0, 4, 7, 9)**

**(1, 3, 7, 9)**

**(2, 4, 5, 9)**

| **#include <iostream>**  **using namespace std;**    **// Naive recursive function to check if quadruplet exists in an array**  **// with the given sum**  **bool hasQuadruplet(int nums[], int n, int target, int count)**  **{**  **// if the desired sum is reached with 4 elements, return true**  **if (target == 0 && count == 4) {**  **return true;**  **}**    **// return false if the sum is not possible with the current configuration**  **if (count > 4 || n == 0) {**  **return false;**  **}**    **// Recur with**  **// 1. Including the current element**  **// 2. Excluding the current element**    **return hasQuadruplet(nums, n - 1, target - nums[n - 1], count + 1) ||**  **hasQuadruplet(nums, n - 1, target, count);**  **}**    **int main()**  **{**  **int nums[] = { 2, 7, 4, 0, 9, 5, 1, 3 };**  **int target = 20;**    **int n = sizeof(nums) / sizeof(nums[0]);**    **hasQuadruplet(nums, n, target, 0) ? cout << "Quadruplet exists":**  **cout << "Quadruplet Doesn't Exist";**    **return 0;**  **}** |
| --- |

**59: Print all quadruplets with a given sum | 4 sum problem extended**

**Given an unsorted integer array, print all distinct four elements tuple (quadruplets) in it, having a given sum.**

**Input:**

**A[] = [2, 7, 4, 0, 9, 5, 1, 3]**

**target = 20**

**Output: Below are the quadruplets with sum 20**

**(0, 4, 7, 9)**

**(1, 3, 7, 9)**

**(2, 4, 5, 9)**

**#include <iostream>**

**#include <algorithm>**

**using namespace std;**

**// Function to print all quadruplet present in an array with a given sum**

**void quadTuple(int arr[], int n, int target)**

**{**

**// sort the array in ascending order**

**sort (arr, arr + n);**

**// check if quadruplet is formed by `arr[i]`, `arr[j]`, and a pair from**

**// subarray `arr[j+1…n)`**

**for (int i = 0; i <= n - 4; i++)**

**{**

**for (int j = i + 1; j <= n - 3; j++)**

**{**

**// `k` stores remaining sum**

**int k = target - (arr[i] + arr[j]);**

**// check for sum `k` in subarray `arr[j+1…n)`**

**int low = j + 1, high = n - 1;**

**while (low < high)**

**{**

**if (arr[low] + arr[high] < k) {**

**low++;**

**}**

**else if (arr[low] + arr[high] > k) {**

**high--;**

**}**

**// quadruplet with a given sum found**

**else {**

**cout << "(" << arr[i] << " " << arr[j] << " " <<**

**arr[low] << " " << arr[high] << ")\n";**

**low++, high--;**

**}**

**}**

**}**

**}**

**}**

**int main()**

**{**

**int arr[] = { 2, 7, 4, 0, 9, 5, 1, 3 };**

**int target = 20;**

**int n = sizeof(arr) / sizeof(arr[0]);**

**quadTuple(arr, n, target);**

**return 0;**

**}**

**60: Quickselect Algorithm**

**Quickselect is a selection algorithm to find the k'th smallest element in an unordered list. It is closely related to the** [**Quicksort sorting algorithm**](https://www.techiedelight.com/quicksort/)**. Like Quicksort, it is efficient traditionally and offers good average-case performance, but has a poor worst-case performance.**

**Input: [7, 4, 6, 3, 9, 1]  
k = 2**

**Output: k’th smallest array element is 3**

**Input: [7, 4, 6, 3, 9, 1]**

**k = 1**

**Output: k’th smallest array element is 1**

| **#include <stdio.h>**  **#include <stdlib.h>**    **#define SWAP(x, y) { int temp = x; x = y; y = temp; }**  **#define N (sizeof(nums)/sizeof(nums[0]))**    **// Partition using Lomuto partition scheme**  **int partition(int a[], int left, int right, int pIndex)**  **{**  **// pick `pIndex` as a pivot from the array**  **int pivot = a[pIndex];**    **// Move pivot to end**  **SWAP(a[pIndex], a[right]);**    **// elements less than the pivot will be pushed to the left of `pIndex`;**  **// elements more than the pivot will be pushed to the right of `pIndex`;**  **// equal elements can go either way**  **pIndex = left;**    **// each time we find an element less than or equal to the pivot, `pIndex`**  **// is incremented, and that element would be placed before the pivot.**  **for (int i = left; i < right; i++)**  **{**  **if (a[i] <= pivot)**  **{**  **SWAP(a[i], a[pIndex]);**  **pIndex++;**  **}**  **}**    **// move pivot to its final place**  **SWAP(a[pIndex], a[right]);**    **// return `pIndex` (index of the pivot element)**  **return pIndex;**  **}**    **// Returns the k'th smallest element in the list within `left…right`**  **// (i.e., left <= k <= right). The search space within the array is**  **// changing for each round – but the list is still the same size.**  **// Thus, `k` does not need to be updated with each round.**  **int quickselect(int nums[], int left, int right, int k)**  **{**  **// If the array contains only one element, return that element**  **if (left == right) {**  **return nums[left];**  **}**    **// select `pIndex` between left and right**  **int pIndex = left + rand() % (right - left + 1);**    **pIndex = partition(nums, left, right, pIndex);**    **// The pivot is in its final sorted position**  **if (k == pIndex) {**  **return nums[k];**  **}**    **// if `k` is less than the pivot index**  **else if (k < pIndex) {**  **return quickselect(nums, left, pIndex - 1, k);**  **}**    **// if `k` is more than the pivot index**  **else {**  **return quickselect(nums, pIndex + 1, right, k);**  **}**  **}**    **int main()**  **{**  **int nums[] = { 7, 4, 6, 3, 9, 1 };**  **int k = 2;**    **printf("k'th smallest element is %d", quickselect(nums, 0, N - 1, k - 1));**    **return 0;**  **}** |
| --- |

**61: Print all triplets that form an arithmetic progression**

**Given a sorted array of distinct positive integers, print all triplets that forms an arithmetic progression with an integral common difference.**

**Input: A[] = { 5, 8, 9, 11, 12, 15 }**

**Output:**

**5 8 11**

**9 12 15**

**Input: A[] = { 1, 3, 5, 6, 8, 9, 15 }**

**Output:**

**1 3 5**

**1 5 9**

**3 6 9**

**1 8 15**

**3 9 15**

| **#include <stdio.h>**    **// Function to print all triplets that forms arithmetic progression**  **// in a given sorted array**  **void findAllTriplets(int A[], int n)**  **{**  **// consider `A[j]` as the middle element of AP**  **for (int j = 1; j < n - 1; j++)**  **{**  **// start with the left and right index of `j`**  **int i = j - 1, k = j + 1;**    **// Find all `i` and `k` such that `(i, j, k)` form an AP triplet**  **while (i >= 0 && k < n)**  **{**  **// if `(A[i], A[j], A[k])` forms a triplet**  **if (A[i] + A[k] == 2 \* A[j])**  **{**  **// print the triplet**  **printf("%d %d %d\n", A[i], A[j], A[k]);**    **// Since the array is sorted and elements are distinct**  **k++, i--;**  **}**  **// otherwise, if `(A[i] + A[k])` is less than `2×A[j]` then**  **// try next `k`. Else, try the previous `i`.**  **else if (A[i] + A[k] < 2 \* A[j]) {**  **k++;**  **}**  **else {**  **i--;**  **}**  **}**  **}**  **}**    **int main(void)**  **{**  **int A[] = { 1, 3, 5, 6, 8, 9, 15 };**  **int n = sizeof(A) / sizeof(A[0]);**    **findAllTriplets(A, n);**    **return 0;**  **}** |
| --- |

# **62.** [Bubble Sort Algorithm in Java with Example](https://javarevisited.blogspot.com/2014/08/bubble-sort-algorithm-in-java-with.html)

Bubble Sort is the first sorting algorithm I learned during my college day, and after so many years it's the one I remember by heart. It's kind of weird that one of the most popular sorting algorithms is also one of the worst-performing sorting algorithms. Bubble sort's average-case performance is in O(n^2), which means as the size array grows, the time it takes to sort that array increases quadratically. Due to this reason, bubble sort is not used in production code, instead quick sort and merge sort are preferred over it. In fact, Java's own Arrays.sort() method, which is the [easiest way to sort an array in Java](http://javarevisited.blogspot.com/2012/01/sort-array-in-java-ascending-and.html) also uses two pivot quicksort to sort primitive arrays and a stable mergesort algorithm to sort object arrays.

The reason for the slow performance of this algorithm is an excessive comparison and swapping since it compares each element of array to another and swaps if it is on the right side.

Due to quadratic performance, bubble sort is best suited for small, almost sorted lists e.g. {1, 2, 4, 3, 5}, where it just needs to do one swapping. Ironically, the best-case performance of bubble sort, which is O(n) beats quicksort's best-case performance of O(NlogN).

Someone may argue that why teaching an algorithm that poor performance, why not teach insertion or selection sort which is as easy as bubble sort and performs better. IMHO, the easiness of the algorithm depends upon the programmer as much as on the algorithm itself.

Many programmers will find *insertion sort* easier than *bubble sor*t but again there will be a lot many who will find bubble sort easier to remember, including me. This is true, despite many of them have used insertion sort unknowingly in real life, e.g. sorting playing cards in hand.

Another reason for learning this sorting algorithm is for comparative analysis, how you improve algorithms, how you come up with different algorithms for the same problems. In short, despite of all its shortcomings, [bubble sort](http://java67.blogspot.sg/2012/12/bubble-sort-in-java-program-to-sort-integer-array-example.html) is still the most popular algorithm.

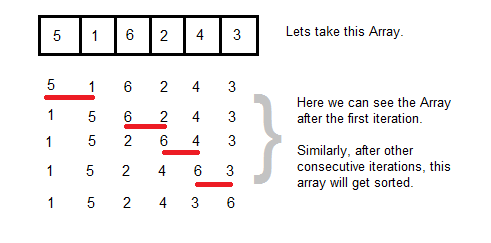
In this tutorial, we will learn *how bubble sort works*, the complexity, and performance of bubble sort algorithm, implementation, and source code in Java and a step-by-step example of bubble sort.

## **How Bubble Sort Algorithm works**

If you are the one who focus on names, then you might have got an idea how bubble sort works. Just like a bubble comes up from water, in bubble sort smallest or largest number, depending upon whether you are sorting array on ascending or descending order, bubbles up towards start or end of the array.

We need at least N pass to sort the array completely and at the end of each pass one elements are sorted in its proper position. You can take first element from array, and start comparing it with other element, [swapping](http://javarevisited.blogspot.sg/2013/02/swap-two-numbers-without-third-temp-variable-java-program-example-tutorial.html) where it's lesser than the number you are comparing.

You can start this comparison from start or from end, as we have compared elements from end in our bubble sort example. It is said that a picture is worth more than a thousand word and it's particularly true in case of understanding sorting algorithm. Let' see an *step by step example to sort array using bubble sort*, as I said after each pass largest number is sorted.

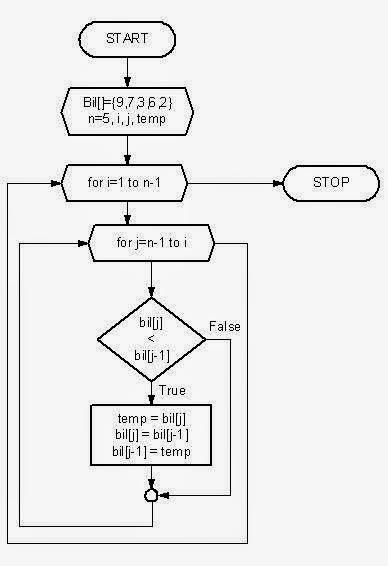


In this [array](http://javarevisited.blogspot.sg/2013/11/java-array-101-for-programmers-and.html), we start from index 0, which is 5 and starts comparing elements from start to end. So first element we compare 5 is 1, and since 5 is greater than 1 we swap them ( because ascending order sorted array will have larger number towards end).

Next we compare 5 to 6, here no swapping because 6 is greater than 5 and it's on higher index than 5. Now we compare 6 to 2, again we need swapping to move 6 towards end. At the end of this pass 6 reaches (bubbles up) at the top of the array. In next iteration 5 will be sorted on its position and after n iteration all elements will be sorted. Since we compare each element with another, we need two for loops and that result in complexity of O(n^2).

### **FlowChart of Bubble Sort Algorithm**

Another cool way to understand an algorithm is to draw it's flowchart. It will walk through each iteration in loop and how decisions are made during algorithm execution. Here is flowchart of our bubble sort algorithm, which complements our implementation of this sorting algorithm.



Here we have integer array {9, 7, 3, 6, 2} and start with four variable i, j, temp and array length which is stored in variable n. We have two for loop, outer loop runs from 1 to n-1. Our inner loop runs from n-1 to I.

Many programmer make mistake here, if you start outer loop with second element than make sure to use j>=i condition on inner loop, or if you start with first element e.g. i=0, make sure you use j>i to avoid [ArrayIndexOutOfBound exception](http://javarevisited.blogspot.sg/2014/05/exception-in-thread-main-arrayindexoutofboundsexception-java.html). Now we compare each element and swap them to move smaller element towards front of array.

As I said depending upon your navigation direction either largest element will be sorted at highest index in first pass or smallest element will be placed in lowest index. In this case, after first pass, smallest number will be sorted. This loop runs until j>=i after than it finishes and i becomes i + 1. This whole process repeats until outer loop is finished and that time your array is sorted.

In flowchart, a diamond box is used for decision making, which is equivalent of if-else statement in code. You can see here decision box is inside inner loop, which means we do N comparison in each iteration, totals to NxN comparisons.

### **Complexity and Performance of Bubble Sort Algorithm**

As I said before compared to other sorting algorithm like quicksort, merge sort or shell sort, bubble sort performs poorly. These algorithm has average case complexity of O(NLogN), while average case complexity of bubble sort O(n^2). Ironically in best case bubble sort do better than [quicksort](http://java67.blogspot.com/2014/07/quicksort-algorithm-in-java-in-place-example.html) with O(n) performance. Bubble sort is three times slower than quicksort or mergesort even for n = 100 but it's easier to implement and remember. here is the summary of bubble sort performance and complexity :

Bubble sort Worst case performance O(n^2)

Bubble sort Best case performance O(n)

Bubble sort Average case performance O(n^2)

You can further explore insertion sort and selection sort, which also does sorting in similar time complexity. By the you can not only sort the array using bubble sort but ArrayList or any other collection class as well. Though you should really use Arrays.sort() or Collections.sort() for those purpose.

### **Bubble Sort Implementation in Java**

here is the Java program to implement bubble sort algorithm using Java programming language. Don't surprise with import of java.util.Array, we have not used it's sort method here, instead it is used to [print arrays in readable format](http://javarevisited.blogspot.sg/2012/12/3-example-to-print-array-values-in-java.html). I have created a swap function to swap numbers and improve readability of code, if you don't like you can in-line the code in the swap method inside if statement of inner loop. Though I have used main method for testing, as it demonstrate better, I would suggest you to write some unit test case for your bubble sort implementation. If you don't know how to do that, you can see this [JUnit tutorial](http://javarevisited.blogspot.sg/2013/03/how-to-write-unit-test-in-java-eclipse-netbeans-example-run.html).

import java.util.Arrays;

/\*\*

\* Java program to implement bubble sort algorithm and sort integer array using

\* that method.

\*

\* @author Javin Paul

\*/

public class BubbleSort{

public static void main(String args[]) {

bubbleSort(new int[] { 20, 12, 45, 19, 91, 55 });

bubbleSort(new int[] { -1, 0, 1 });

bubbleSort(new int[] { -3, -9, -2, -1 });

}

/\*

\* This method sort the integer array using bubble sort algorithm

\*/

public static void bubbleSort(int[] numbers) {

System.out.printf("Unsorted array in Java :%s %n",

Arrays.toString(numbers));

for (int i = 0; i < numbers.length; i++) {

for (int j = numbers.length -1; j > i; j--) {

if (numbers[j] < numbers[j - 1]) {

swap(numbers, j, j-1);

}

}

}

System.out.printf("Sorted Array using Bubble sort algorithm :%s %n",

Arrays.toString(numbers));

}

/\*

\* Utility method to swap two numbers in array

\*/

public static void swap(int[] array, int from, int to){

int temp = array[from];

array[from] = array[to];

array[to] = temp;

}

}

Output

Unsorted array in Java : [20, 12, 45, 19, 91, 55]

Sorted Array using Bubble sort algorithm : [12, 19, 20, 45, 55, 91]

Unsorted array in Java : [-1, 0, 1]

Sorted Array using Bubble sort algorithm : [-1, 0, 1]

Unsorted array in Java : [-3, -9, -2, -1]

Sorted Array using Bubble sort algorithm : [-9, -3, -2, -1]

# 63. How to implement Merge Sort Algorithm in Java

The merge sort algorithm is a divide and conquers algorithm. In the divide and conquer paradigm, a problem is broken into smaller problems where each small problem still retains all the properties of the larger problem -- except its size. To solve the original problem, each piece is solved individually; then the pieces are merged back together. For example, imagine you have to sort an array of 200 elements using the [bubble sort](http://www.java67.com/2012/12/bubble-sort-in-java-program-to-sort-integer-array-example.html) algorithm. Since selection sort takes O(n^2) time, it would take about 40,000-time units to sort the array. Now imagine splitting the array into ten equal pieces and sorting each piece individually still using selection sort. Now it would take 400-time units to sort each piece; for a grand total of 10\*400 = 4000.

Once each piece is sorted, merging them back together would take about 200-time units; for a grand total of 200+4000 = 4,200. Clearly, 4,200 is an impressive improvement over 40,000.

Now think bigger. Imagine splitting the original array into groups of two and then sorting them. In the end, it would take about 1,000-time units to sort the array.

That's how merge sort works. It makes sorting a big array easy and hence it's suitable for large integer and string arrays. Time Complexity of the mergesort algorithm is the same in the best, average, and worst-case and it's equal to O(n\*log(n))

Btw, if you are new to Algorithms and Data Structure and not familiar with essential sorting and searching algorithms like quicksort, binary search, level order search, etc then I suggest you go through a good, comprehensive online course like [**Data Structures and Algorithms: Deep Dive Using Java**](https://www.java67.com/2019/07/top-10-online-courses-to-learn-data-structure-and-algorithms-in-java.html) to learn the basics first.

## **Sorting Array in Java using Merge Sort Algorithm - Example**

You have given an unordered list of integers (or any other objects e.g. String), You have to rearrange the integers or objects in their natural order.

Sample Input: {80, 50, 30, 10, 90, 60, 0, 70, 40, 20, 5}

Sample Output: {0, 10, 20, 30, 40, 50, 50, 60, 70, 80, 90}

**import** java.util.Arrays;

/\*

\* Java Program to sort an integer array using merge sort algorithm.

\*/

**public** class Main {

**public** **static** **void** main(String[] args) {

System.out.println("mergesort");

**int**[] **input** **=** { 87, 57, 370, 110, 90, 610, 02, 710, 140, 203, 150 };

System.out.println("array before sorting");

System.out.println(Arrays.**toString**(**input**));

// sorting array using MergeSort algorithm

mergesort(**input**);

System.out.println("array after sorting using mergesort algorithm");

System.out.println(Arrays.**toString**(**input**));

}

/\*\*

\* Java function to sort given array using merge sort algorithm

\*

\* @param input

\*/

**public** **static** **void** mergesort(**int**[] **input**) {

mergesort(**input**, 0, **input**.**length** **-** 1);

}

/\*\*

\* A Java method to implement MergeSort algorithm using recursion

\*

\* @param input

\* , integer array to be sorted

\* @param start

\* index of first element in array

\* @param end

\* index of last element in array

\*/

**private** **static** **void** mergesort(**int**[] **input**, **int** **start**, **int** end) {

// break problem into smaller structurally identical problems

**int** mid **=** (**start** **+** end) **/** 2;

**if** (**start** < end) {

mergesort(**input**, **start**, mid);

mergesort(**input**, mid **+** 1, end);

}

// merge solved pieces to get solution to original problem

**int** i **=** 0, **first** **=** **start**, **last** **=** mid **+** 1;

**int**[] tmp **=** **new** **int**[end **-** **start** **+** 1];

**while** (**first** <**=** mid **&&** **last** <**=** end) {

tmp[i**++**] **=** **input**[**first**] < **input**[**last**] **?** **input**[**first++**] **:** **input**[**last++**];

}

**while** (**first** <**=** mid) {

tmp[i**++**] **=** **input**[**first++**];

}

**while** (**last** <**=** end) {

tmp[i**++**] **=** **input**[**last++**];

}

i **=** 0;

**while** (**start** <**=** end) {

**input**[**start++**] **=** tmp[i**++**];

}

}

}

Output

mergesort

array before sorting

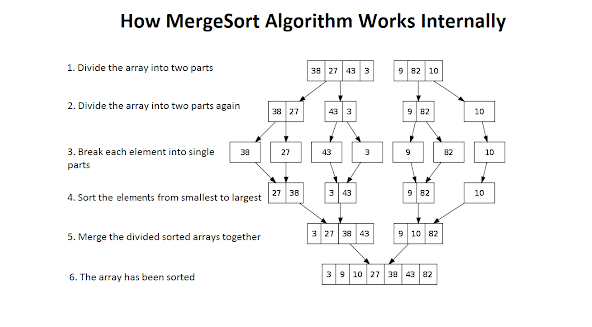
[87, 57, 370, 110, 90, 610, 2, 710, 140, 203, 150]

array after sorting using mergesort algorithm

[2, 57, 87, 90, 110, 140, 150, 203, 370, 610, 710]

You can see that the array is now sorted. The algorithm we have used is a recursive implementation of merge sort and it's also a [stable sorting algorithm](https://javarevisited.blogspot.com/2017/06/difference-between-stable-and-unstable-algorithm.html) I mean it maintains the original order of elements in case of a tie.

Anyway, if you haven't got it yet that how the merge sort algorithm works, you can also check out the [**Algorithms and Data Structures - Part 1 and 2**](https://javarevisited.blogspot.com/2018/11/top-5-data-structures-and-algorithm-online-courses.html) course on Pluralsight which explains key sorting and searching algorithms in a very nice way. It also covers essential data structures like a linked list, array, hash table, binary tree, etc.



# **64. How to implement Radix Sort in Java - Algorithm Example**

The Radix sort, like [counting sort](http://www.java67.com/2017/06/counting-sort-in-java-example.html) and [bucket sort](http://javarevisited.blogspot.sg/2017/01/bucket-sort-in-java-with-example.html#axzz58bLA7tAs), is an integer-based algorithm (I mean the values of the input array are assumed to be integers). Hence radix sort is among the fastest sorting algorithms around, in theory. It is also one of the few O(n) or linear time sorting algorithms along with the Bucket and Counting sort. The particular distinction for radix sort is that it creates a bucket for each cipher (i.e. digit); as such, similar to bucket sort, each bucket in radix sort must be a growable list that may admit different keys.

For decimal values, the number of buckets is 10, as the decimal system has 10 numerals/cyphers (i.e. 0,1,2,3,4,5,6,7,8,9). Then the keys are continuously sorted by significant digits.

Time Complexity of radix sort in the best case, average case, and worst case is O(k\*n) where k is the length of the longest number, and n is the size of the input array.

Note: if k is greater than log(n) then a n\*log(n) algorithm would be a better fit. In reality, we can always change the Radix to make k less than log(n).

Btw, if you are not familiar with time and space complexity and how to calculate or optimize it for a particular algorithm then I suggest you first go through a fundamental algorithms course like [**Data Structures and Algorithms: Deep Dive Using Java** on Udemy](https://www.java67.com/2019/07/top-10-online-courses-to-learn-data-structure-and-algorithms-in-java.html). This will not only help you to do well in interviews but also in your day-to-day job.

## 

## **Java program to implement Radix sort algorithm**

Before solving this problem or implementing a Radix Sort Algorithm, let's first get the problem statement right:

### **Problem Statement:**

Given a disordered list of integers, rearrange them in the natural order.

Sample Input: {18,5,100,3,1,19,6,0,7,4,2}

Sample Output: {0,1,2,3,4,5,6,7,18,19,100}

### **Solution**

Here is a sample program to implement the Radix sort algorithm in Java

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

**/\***

**\*** Java Program sort an integer array using radix sort algorithm.

**\*** input: [180, 50, 10, 30, 10, 29, 60, 0, 17, 24, 12]

**\*** output: [0, 10, 10, 12, 17, 24, 29, 30, 50, 60, 180]

**\***

**\*** **Time** Complexity **of** Solution:

**\*** Best Case O(k**\***n); Average Case O(k**\***n); Worst Case O(k**\***n),

**\*** **where** k **is** **the** **length** **of** **the** longest number **and** n **is** **the**

**\*** size **of** **the** input array.

**\***

**\*** Note: **if** k **is greater than** **log**(n) **then** an n**\*log**(n) algorithm would be a

**\*** better fit. **In** reality we can always change **the** radix **to** make k

**\*** less than **log**(n).

**\***

**\*/**

public class Main {

public static void main(String[] args) {

System.out.println("Radix sort in Java");

int[] input **=** { 181, 51, 11, 33, 11, 39, 60, 2, 27, 24, 12 };

System.out.println("An Integer array before sorting");

System.out.println(Arrays.toString(input));

**//** sorting array using radix Sort Algorithm

radixSort(input);

System.out.println("Sorting an int array using radix sort algorithm");

System.out.println(Arrays.toString(input));

}

**/\*\***

**\*** Java method **to** sort a **given** array using radix sort algorithm

**\***

**\*** @param input

**\*/**

public static void radixSort(int[] input) {

final int RADIX **=** 10;

**//** declare **and** initialize bucket[]

List**<**Integer**>**[] bucket **=** new ArrayList[RADIX];

**for** (int i **=** 0; i **<** bucket.**length**; i**++**) {

bucket[i] **=** new ArrayList**<**Integer**>**();

}

**//** sort

boolean maxLength **=** false;

int tmp **=** -1, placement **=** 1;

**while** (!maxLength) {

maxLength **=** true;

**//** split input **between** lists

**for** (Integer i : input) {

tmp **=** i **/** placement;

bucket[tmp % RADIX].add(i);

**if** (maxLength **&&** tmp **>** 0) {

maxLength **=** false;

}

}

**//** empty lists **into** input array

int a **=** 0;

**for** (int b **=** 0; b **<** RADIX; b**++**) {

**for** (Integer i : bucket[b]) {

input[a**++**] **=** i;

}

bucket[b].clear();

}

**//** move **to** next digit

placement **\*=** RADIX;

}

}

}

Output

Radix sort **in** Java

An Integer array **before** sorting

[181, 51, 11, 33, 11, 39, 60, 2, 27, 24, 12]

Sorting an int array using radix sort algorithm

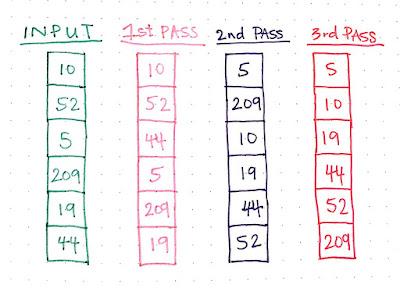
[2, 11, 11, 12, 24, 27, 33, 39, 51, 60, 181]

Here is another example of sorting a list of an integer using Radix sort, just in case If you haven't got the concept of how Radix sort works:

### **Problem Statement:**

Sort the list of numbers 10, 52, 5, 209, 19, and 44 using the Radix sort algorithm:

### **Solution:**

****

That's all about **how to sort an integer array using radix sort in Java**. Along with Counting Sort and Bucket sort, it is also an O(n) sorting algorithm. These algorithms are not general-purpose and you cannot use them to sort any object like String, Employee, etc. They are best suited for a small range of known integer values but they provide awesome performance.

Read more: <https://www.java67.com/2018/03/how-to-implement-radix-sort-in-java.html#ixzz7NWKfRmq7>

# 65. Iterative QuickSort Example in Java - without Recursion

The quicksort algorithm is one of the important sorting algorithms. Similar to merge sort, quicksort also uses divide-and-conquer hence it's easy to implement a quicksort algorithm using recursion in Java, but it's slightly more difficult to write an iterative version of quicksort. That's why Interviewers are now asking to implement QuickSort without using recursion. The interview will start with something like writing a program to sort an array using a quicksort algorithm in Java and most likely you will come up with a recursive implementation of quicksort as shown [here](http://javarevisited.blogspot.com/2014/08/quicksort-sorting-algorithm-in-java-in-place-example.html). As a follow-up, the Interviewer will now ask you to code the same algorithm using Iteration.

If you remember, while solving binary tree problems without recursion e.g. pre-order traversal without recursion ([here](http://www.java67.com/2016/07/binary-tree-preorder-traversal-in-java-without-recursion.html)) and in-order traversal without recursion ([here](http://www.java67.com/2016/07/binary-tree-preorder-traversal-in-java-without-recursion.html)), we have used **Stack** to replace recursion. You can use the same technique here to write an iterative quicksort program in Java. The Stack actually mimics the recursion.

Before going for a programming/coding interview, It's absolutely necessary to do as much practice in data structure and algorithms as possible to take advantage of all the knowledge available. You can also join a comprehensive Data Structure and Algorithms course like [**Data Structures and Algorithms: Deep Dive Using Java**](https://javarevisited.blogspot.com/2018/11/top-5-data-structures-and-algorithm-online-courses.html)on Udemy to fill the gaps in your understanding.

## **Iterative Quicksort Algorithm**

I learned about quicksort in my engineering classes, one of the few algorithms which I managed to understand then. Since it's a divide-and-conquer algorithm, you choose a pivot and divide the array.

Unlike merge sort, which is also a divide-and-conquer algorithm and where all-important work happens on combine steps, In [quicksort](https://javarevisited.blogspot.com/2014/08/quicksort-sorting-algorithm-in-java-in-place-example.html#axzz6dXsEfLvJ), the real work happens in the divide step and the combining step does nothing important.

Btw, the working of the algorithm will remain the same whether you implement an iterative solution or a recursion solution. In an iterative solution, we'll use [Stack](https://javarevisited.blogspot.com/2017/03/difference-between-stack-and-queue-data-structure-in-java.html#axzz5dxZIUUxy) instead of [Recursion](https://www.java67.com/2021/07/recursion-programming-exercises-in-java.html).

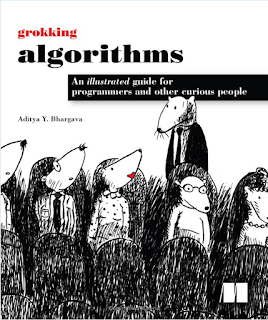
Here are the steps to implement iterative quicksort in Java:

* Push the range (0...n) into the Stack
* Partition the given array with a pivot
* Pop the top element.
* Push the partitions ( index range ) into a stack if the range has more than one element
* Do the above 3 steps, till the stack, is empty

You might know that even though writing recursive algorithms are easy they are always slower than their Iterative counterpart. So, when the Interviewer will ask you to choose a method in terms of time complexity where memory is not a concern, which version will you choose?

Well, both recursive and iterative quicksorts are O(N log N) average case and O(n^2) in the worst case but the recursive version is shorter and clearer. Iterative is faster and makes you simulate the recursion call stack.

Btw, if you want to understand more about *what does recursion has to do with the stack?* and *why does quicksort run in O(n log n) time in the average case?* I suggest reading [Grokking Algorithms](https://javarevisited.blogspot.com/2017/10/grokking-algorithms-by-aditya-bhargava-best-beginner-book.html#axzz6dXsEfLvJ), a rare algorithm book that is easy to understand with real-world examples. I just bought a copy of this book and even though I know all those algorithms, I find it quite readable and gain a new perspective. So, if you are struggling with the algorithms, this is the book you should read now.



## **QuickSort example in Java without recursion.**

Here is our sample Java program to implement Quicksort using for loop and stack, without using recursion. This is also known as the iterative quicksort algorithm.

import java.util.Arrays;

import java.util.Scanner;

import java.util.Stack;

/\*\*

\* Java Program to implement Iterative QuickSort Algorithm, without recursion.

\*

\* @author WINDOWS 8

\*/

public class Sorting {

public static void main(String args[]) {

int[] unsorted = {34, 32, 43, 12, 11, 32, 22, 21, 32};

System.out.println("Unsorted array : " + Arrays.toString(unsorted));

iterativeQsort(unsorted);

System.out.println("Sorted array : " + Arrays.toString(unsorted));

}

/\*

\* iterative implementation of quicksort sorting algorithm.

\*/

public static void iterativeQsort(int[] numbers) {

Stack stack = new Stack();

stack.push(0);

stack.push(numbers.length);

while (!stack.isEmpty()) {

int end = stack.pop();

int start = stack.pop();

if (end - start < 2) {

continue;

}

int p = start + ((end - start) / 2);

p = partition(numbers, p, start, end);

stack.push(p + 1);

stack.push(end);

stack.push(start);

stack.push(p);

}

}

/\*

\* Utility method to partition the array into smaller array, and

\* comparing numbers to rearrange them as per quicksort algorithm.

\*/

private static int partition(int[] input, int position, int start, int end) {

int l = start;

int h = end - 2;

int piv = input[position];

swap(input, position, end - 1);

while (l < h) {

if (input[l] < piv) {

l++;

} else if (input[h] >= piv) {

h--;

} else {

swap(input, l, h);

}

}

int idx = h;

if (input[h] < piv) {

idx++;

}

swap(input, end - 1, idx);

return idx;

}

/\*\*

\* Utility method to swap two numbers in given array

\*

\* @param arr - array on which swap will happen

\* @param i

\* @param j

\*/

private static void swap(int[] arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

Output:

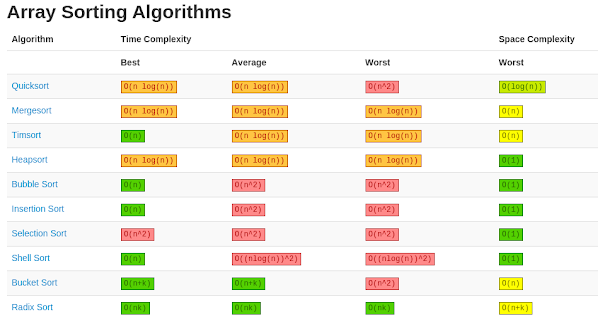
Unsorted array : [34, 32, 43, 12, 11, 32, 22, 21, 32]

Sorted array : [11, 12, 21, 22, 32, 32, 32, 34, 43]

That's all about **how to implement quicksort in Java without recursion**. Just remember, when you use for loop and stack to implement quicksort, it's known as iterative implementation and when you call the method itself, it's known as recursive implementation.

The recursive solution of quicksort is easier to write and understand but the iterative solution is much faster. Though average and worst-case time complexity of both recursive and iterative quicksorts are O(N log N) average case and O(n^2).

Btw, if you want to remember or review the time complexity of different sorting algorithms e.g. [quicksort](http://www.java67.com/2014/07/quicksort-algorithm-in-java-in-place-example.html), [merge sor](https://www.java67.com/2018/03/mergesort-in-java-algorithm-example-and.html)t, [insertion sort](http://www.java67.com/2014/09/insertion-sort-in-java-with-example.html), [radix sort](https://www.java67.com/2018/03/how-to-implement-radix-sort-in-java.html), shell sort, or [bubble sort](http://www.java67.com/2012/12/bubble-sort-in-java-program-to-sort-integer-array-example.html), here is a nice slide you can print and use:



Read more: <https://javarevisited.blogspot.com/2016/09/iterative-quicksort-example-in-java-without-recursion.html#ixzz7NWKv7etY>

# 66. **Write C++ Program to Multiply Two Matrices**

**#include <iostream>**

**using** **namespace** std;

**int** main()

{

**int** a[10][10], b[10][10], result[10][10], r1, c1, r2, c2, i, j, k;

cout<<"Enter rows and column for first matrix: ";

cin>>r1;

cin>>c1;

cout<<"Enter rows and column for second matrix: ";

cin>>r2;

cin>>c2;

// Column of first matrix should be equal to column of second matrix and

**while** (c1 != r2)

{

cout<<"Error! column of first matrix not equal to row of second.\n\n";

cout<<"Enter rows and column for first matrix: ";

cin>>r1;

cin>>c1;

cout<<"Enter rows and column for second matrix: ";

cin>>r2;

cin>>c2;

}

// Storing elements of first matrix.

cout<<"\nEnter elements of matrix 1:\n";

**for**(i=0; i<r1; ++i)

**for**(j=0; j<c1; ++j)

{

cout<<"Enter element a"<<i+1<<j+1<<": ";

cin>>a[i][j];

}

// Storing elements of second matrix.

cout<<"\nEnter elements of matrix 2:\n";

**for**(i=0; i<r2; ++i)

**for**(j=0; j<c2; ++j)

{

cout<<"Enter element b"<<i+1<<j+1<<": ";

cin>>b[i][j];

}

// Initializing all elements of result matrix to 0

**for**(i=0; i<r1; ++i)

**for**(j=0; j<c2; ++j)

{

result[i][j] = 0;

}

// Multiplying matrices a and b and

// storing result in result matrix

**for**(i=0; i<r1; ++i)

**for**(j=0; j<c2; ++j)

**for**(k=0; k<c1; ++k)

{

result[i][j]+=a[i][k]\*b[k][j];

}

// Displaying the result

cout<<"\nOutput Matrix:\n";

**for**(i=0; i<r1; ++i)

**for**(j=0; j<c2; ++j)

{

cout<<" "<<result[i][j];

**if**(j == c2-1)

cout<<"\n\n";

}

**return** 0;

}

# 67. **Write C++ Program to check whether two matrices are equal or not**

**#include <iostream>**

**using** **namespace** std;

**#define size 2 // Matrix size declaration**

**int** main()

{

**int** A[size][size];

**int** B[size][size];

**int** row, col, isEqual;

// Input elements in first matrix from user

cout<<"Enter elements in matrix A of size "<<size<<" x "<<size<<"\n";

**for**(row=0; row<size; row++)

{

**for**(col=0; col<size; col++)

{

cin>>A[row][col];

}

}

// Input elements in second matrix from user

cout<<"Enter elements in matrix B of size"<<size<<" x "<<size<<"\n";

**for**(row=0; row<size; row++)

{

**for**(col=0; col<size; col++)

{

cin>>B[row][col];

}

}

// Assumes that the matrices are equal

isEqual = 1;

**for**(row=0; row<size; row++)

{

**for**(col=0; col<size; col++)

{

//If the corresponding entries of matrices are not equal

**if**(A[row][col] != B[row][col])

{

isEqual = 0;

**break**;

}

}

}

/\*

\* Checks the value of isEqual

\* As per our assumption if isEqual contains 1 means both are equal

\* If it contains 0 means both are not equal

\*/

**if**(isEqual == 1)

{

cout<<"\nMatrix A is equal to Matrix B";

}

**else**

{

cout<<"\nMatrix A is not equal to Matrix B";

}

**return** 0;

}

# 68. **Write C++ Program to Find the Transpose of a given Matrix**

**#include <iostream>**

**using** **namespace** std;

**int** main()

{

**static** **int** array[10][10];

**int** i, j, m, n;

cout<<"Enter the order of the matrix \n";

// Inputing elements in matrix from user

cin>>m>>n;

cout<<"Enter the coefiicients of the matrix\n";

**for** (i = 0; i < m; ++i)

{

**for** (j = 0; j < n; ++j)

{

cin>>array[i][j];

}

}

//Printing the original matrix

cout<<"The given matrix is \n";

**for** (i = 0; i < m; ++i)

{

**for** (j = 0; j < n; ++j)

{

cout<<" "<<array[i][j];

}

cout<<"\n";

}

//Printing the transpose of matrix

cout<<"Transpose of matrix is \n";

**for** (j = 0; j < n; ++j)

{

**for** (i = 0; i < m; ++i)

{

cout<<" "<<array[i][j];

}

cout<<"\n";

}

**return** 0;

}

# 69. C++ Program to implement Binary Search using array

Write a C++ Program to implement Binary Search using array. Here’s simple Program to implement Binary Search using array in C++ Programming Language.

Binary search is an algorithm used to search for an element in a sorted array. In this algorithm the targeted element is compared with middle element. If both elements are equal then position of middle element is returned and hence targeted element is found.

If both elements are unequal then if targeted element is less or more than middle element we discard the lower or upper half and the search continues by finding new middle element.

/\* C++ Program to implement Binary Search using array \*/

#include<iostream>

using namespace std;

int main()

{

int search(int [],int,int);

int n,i,a[100],e,res;

cout<<"Enter size of Array :: ";

cin>>n;

cout<<"\nEnter elements to the array :: \n";

for(i=0;i<n;++i)

{

cout<<"\nEnter "<<i+1<<" element :: ";

cin>>a[i];

}

cout<<"\nEnter element to search :: ";

cin>>e;

res=search(a,n,e);

if(res!=-1)

cout<<"\nElement found at position "<<res+1<<"\n";

else

cout<<"\nElement is not found....!!!";

return 0;

}

int search(int a[],int n,int e)

{

int f,l,m;

f=0;

l=n-1;

while(f<=l)

{

m=(f+l)/2;

if(e==a[m])

return(m);

else

if(e>a[m])

f=m+1;

else

l=m-1;

}

return -1;

}