String

1. Replace Blanks in String (O(n) time complexity)

   Question 9 Please implement a function to replace each blank in a string with "%20". For instance, it outputs "We%20are%20happy." if the input is "We are happy.".

   -
   - Key is to think the problem in an opposite way. Traverse the string form right to left. Count how many spaces in the string and use two pointers. One point to the rear of the string, the other point to the position based on the number of white spaces.
   -

```
26 void replaceWhiteSpaces(char* c)
27 {
28         char* backup = c;
29         int numWhiteSpaces = 0;
30         int Length = length(c);
31         while (*backup)
32         {
33                 if (*backup == ' ')
34                         numWhiteSpaces++;
35                 backup++;
36         }
37         cout << "Num of Spaces: " << numWhiteSpaces << endl;
38         backup = c;
39         char* p1,*p2;
40         p1 = backup + Length;
41         p2 = p1 + 2 * numWhiteSpaces;
42         while (Length--)
43         {
44                 if (*p1 != ' ')
45                         *(p2--) = *(p1--);
46                 else
47                 {
48                         *(p2--) = '0';
49                         *(p2--) = '2';
50                         *(p2--) = '%';
51                         p1--;
52                 }
53         }
54         c = p1;
55 }
```

Search in 2-D array

1. **Question 7** In a 2-D matrix, every row is increasingly sorted from left to right, and the last number in each row is not greater than the first number of the next row. A sample matrix follows. Please implement a function to check whether a number is in such a matrix or not. It returns `true` if it tries to find the number 7 in the sample matrix, but it returns `false` if it tries to find the number 12.

   1 3 5
   7 9 11
   13 15 17

   - Treat the 2-D array as a 1-D array, and apply binary search algorithm. Remember to use mid / col to determine row, and mid % col to determine col.
   -
```
bool search(int a[3][3],int row,int col,int val)
{
        int low = 0;
        int high = row * col - 1;
        int mid;
        while (low <= high)
        {
                mid = low + (high - low) / 2;
                int i = mid / col;
                int j = mid % col;
                if (a[i][j] < val)
                        low = mid + 1;
                else if(a[i][j] > val)
                        high = mid - 1;
                else
                        return true;
        }
        return false;
}
```

2. **Question 8** In a 2-D matrix, every row is increasingly sorted from left to right, and every column is increasingly sorted from top to bottom. Please implement a function to check whether a number is in such a matrix or not. For example, all rows and columns are increasingly sorted in the following matrix. It returns `true` if it tries to find number 7, but it returns `false` if it tries to find number 5.

   1 2 8 9
   2 4 9 12
   4 7 10 13
   6 8 11 15

   - Start from the top right position of the array. Compare the value with the target value. If larger than the target value, neglect the entire column. For the array containing the remaining left three columns, ….if less than the target value, remove the row. … Iteratively.

```
89 bool searchInDiagonal(int a[4][5],int row,int col,int val)
90 {
91         int r = 0;
92         int c = col;
93         while (r < row && c >= 0)
94         {
95                 if (a[r][col] == val)
96                         return true;
97                 else if (a[r][col] > val)
98                         c--;
99                 else
100                        r++;
101        }
102        return false;
103 }
```

3. **Question 6** An array contains *n* numbers ranging from 0 to *n*-1. There are some numbers duplicated in the array. It is not clear how many numbers are duplicated or how many times a number gets duplicated. How do you find a duplicated number in the array? For example, if an array of length 7 contains the numbers {2, 3, 1, 0, 2, 5, 3}, the implemented function (or method) should return either 2 or 3.
   - According the value in the array, once encounter a number, we check if the value is equal to the current index. If so, then check next one, otherwise, use the value as index and check that value with itself. If equal, then duplication found. Otherwise, swap those two values….until the entire array has been accessed.

```
71
72 void findDuplicates(int* array,int size,vector<int>& v)
73 {
74         int i = 0;
75         while (i < size)
76         {
77                 while (i < size && array[i] != i)
78                 {
79                         if (array[i] == array[array[i]])
80                         {
81                                 if (find(v.begin(),v.end(),array[i]) == v.end())
82                                         v.push_back(array[i]);
83                                 i++;
84                         }
85                         else
86                         {
87                                 int temp,index1,index2;
88                                 index1 = i;
89                                 index2 = array[i];
90                                 temp = array[index1];
91                                 array[index1] = array[index2];
92                                 array[index2] = temp;
93                         }
94                 }
95                 i++;
96         }
97 }
```

     - i

4. **Question 10** Given two sorted arrays, denoted as *array1* and *array2*, please merge them into *array1* and keep the merged array sorted. Suppose there is sufficient vacant memory at the end of *array1* to accommodate elements of *array2*.

   - According to the replace white spaces problem in a string, we can still iterate from right to left. Thus, the complexity is: O(m + n).
   -
```
38 void merge(int** array, int size1, int* array2, int size2)
39 {
40         *array = realloc(*array,size1,size1 + size2);
41         int* array1 = *array;
42         int pos1 = size1 - 1;
43         int pos2 = pos1 + size2;
44         cout << "pos2: " << pos2 << endl;
45         int pos3 = size2 - 1;
46         while (pos1 >= 0 && pos3 >= 0)
47         {
48                 cout << "array1[pos1]: " << array1[pos1] << endl;
49                 cout << "array2[pos3]: " << array2[pos3] << endl;
50                 if (array1[pos1] < array2[pos3])
51                         array1[pos2--] = array2[pos3--];
52                 else
53                         array1[pos2--] = array1[pos1--];
54         }
55 }
```

Tree

1. **Question 18** Given a node in a binary tree, please implement a function to retrieve its next node in the in-order traversal sequence. There is a pointer to the parent node in each tree node.

   - The tree in Figure 3-12 is a binary tree whose in-order traversal sequence is *d, b, h, e, i, a, f, c, g*. Let's take it as an example to analyze how to get the next node in a binary tree.

     If a node has a right child, its next node is the most left child in its right subtree. That is to say, it moves to the right child and then traverses along the links to the left child as much as possible. For example, the next node of node *b* is node *h*, and the next node of node *a* is node *f*.

     If a node does not have a right child, its next node is its parent if it is the left child of its parent. For instance, the next node of node *d* is node *b*, and the next node of node *f* is node *c*.

     It is more complex to get the next node of a node that does not have a right child and is the right child of its parent. It traverses along the links to parents until it reaches a node that is the left child of its parent. The parent is the next node if such a node exists.

     In order to get the next node of node *i*, it traverses along the link to the parent and reaches node *e* at first. Since node *e* is not the left child of its parent, it continues to traverse and reaches at node *b*, which is the left child of its parent. Therefore, the parent of node *b*, which is node *a*, is the next node after node *i*.

     It is a similar process to get the next node of node g. It first traverses the link to the parent and reaches node *c*. It continues to traverse because node *c* is not a left child of its parent, and it reaches node *a*. Because node *a* does not have a parent, node *g* is the last in the binary tree and it does not have a next node.
   - Two cases: If the node don't have right child, scan through its parent until the parent's right child is not the node itself. (use parent and pCurrent)(exit condition: while (parent != NULL && pParent->rightChild == pCurrent); The other condition is that if the node has right child, then the next node must exist in right subtree. right node and leftmost node of

the right subtree)

```
BinaryTreeNode* GetNext(BinaryTreeNode* pNode) {
    if(pNode == NULL)
        return NULL;

    BinaryTreeNode* pNext = NULL;
    if(pNode->m_pRight != NULL) {
        BinaryTreeNode* pRight = pNode->m_pRight;
        while(pRight->m_pLeft != NULL)
            pRight = pRight->m_pLeft;

        pNext = pRight;
    }
    else if(pNode->m_pParent != NULL) {
        BinaryTreeNode* pCurrent = pNode;
        BinaryTreeNode* pParent = pNode->m_pParent;
        while(pParent != NULL && pCurrent == pParent->m_pRight) {
            pCurrent = pParent;
            pParent = pParent->m_pParent;
        }

        pNext = pParent;
    }

    return pNext;
}
```

-

Array

1. Find the smallest k elements in an array of n size
    - maintain a minimum heap, so that the important elements are all under the ‘bottom’ while those larger elements are on top. Read k elements first, create a maximum heap, read the remaining elements one by one and compare the element with the top of the heap which is the largest elements in the heap. if the element is larger than the top element in the heap, then do nothing, otherwise replace and adjust the heap again….until done,

```
27 void heaptify(int* array,int size,int pos)
28 {
29         int val = array[pos];
30         int child = 2 * pos + 1;
31         while (child < size)
32         {
33                 if (child + 1 < size && array[child + 1] > array[child]) child++;
34                 if (array[child] > val)
35                 {
36                         array[pos] = array[child];
37                         pos = child;
38                         child = 2 * pos + 1;
39                 }
40                 else
41                         break;
42         }
43         array[pos] = val;
44 }
45
46 void creatHeap(int* array,int size)
47 {
48         for (int i = (size - 2) / 2;i >= 0;i--)
49                 heaptify(array,size,i);
50 }
51
52 void findMinK(int* array,int size,int k)
53 {
54         if (k > size)
55                 return;
56         creatHeap(array,k);
57         for (int i = k;i < size;i++)
58         {
59                 if (array[0] > array[i])
60                 {
61                         array[0] = array[i];
62                         heaptify(array,k,0);
63                 }
64         }
65         cout << "Min " << k << " elements are: " << endl;
66         for (int i = 0;i < k;i++)
67                 cout << array[i] << " ";
68 }
```

2. Extract the n largest elements from an array of size m
    - Create a max heap - O(m)
    - Extract the top element from the heap O(log(m)), repeat it for n times - nlog(m);
    - Total time complexity is: O(m + nlog(m));
    - Create a min heap of size n
    - Read in the first n elements and create a min heap - O(n)
    - Read in next element and compare it with the top element, if top < read-in element, then replace and heaptify - O(log(n)), repeat it for (m - n) times.
    - O(n + (m-n) * log(n));//n is constant, thus it is O(m)

Stacks and Queue

1. Describe how you could use a single array to implement three stacks

   Approach 1:

   Divide the array in three equal parts and allow the individual stack to grow in that limited space (note: "[" means inclusive, while "(" means exclusive of the end point).

   - for stack 1, we will use [0, n/3)
   - for stack 2, we will use [n/3, 2n/3)
   - for stack 3, we will use [2n/3, n)

   This solution is based on the assumption that we do not have any extra information about the usage of space by individual stacks and that we can't either modify or use any extra space. With these constraints, we are left with no other choice but to divide equally.

   Approach 2:

   In this approach, any stack can grow as long as there is any free space in the array.

   We sequentially allocate space to the stacks and we link new blocks to the previous block. This means any new element in a stack keeps a pointer to the previous top element of that particular stack.

   In this implementation, we face a problem of unused space. For example, if a stack deletes some of its elements, the deleted elements may not necessarily appear at the end of the array. So, in that case, we would not be able to use those newly freed spaces.

   To overcome this deficiency, we can maintain a free list and the whole array space would be given initially to the free list. For every insertion, we would delete an entry from the free list. In case of deletion, we would simply add the index of the free cell to the free list.

   In this implementation we would be able to have flexibility in terms of variable space utilization but we would need to increase the space complexity.

Algorithm

Question 25 There is a stair with *n* levels. A frog can jump up 1, 2, …, *n*-1, *n* levels at each step on the stair. How many approaches are there for this frog to jump from the bottom of the stair to the top?
For example, there are four approaches for the frog to jump on a stair with three levels: (1) it jumps with three steps, one level for each step; (2) it jumps with two steps, one level for the first step and two levels for the second step; (3) it jumps with two steps, two levels for the first step and one level for the second step; or (4) it jumps in only one step from the bottom to the top directly.

1.
   1. f(n) = f(n - 1) + f(n - 2) + f(n - 3) + …. f(1) + f(0)
   2. f(n - 1) = f(n - 2) + f(n - 3) + …. f(1) + f(0)
   3. f(n) = 2 * f(n - 1)
   4. f(n) = 2 ^ (n - 1)//The fastest way to calculate this is using the bit operations: 1 << (n - 1);

■ **Question 26** Rectangles with size 2×1 are utilized to cover other rectangles, horizontally or vertically. How many approaches are available to cover a 2×8 rectangle with eight 2×1 rectangles (Figure 4-2)?

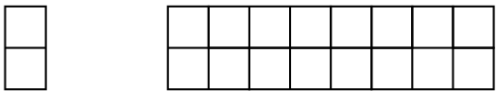


Figure 4-2. A 2×1 rectangle and a 2×8 rectangle

The number of choices to cover a 2×8 rectangle is denoted as $f(8)$. There are two choices to cover with the first 2×1 rectangle. One is to lay it vertically to the left side of the 2×8 rectangle, and the number of choices to cover the remaining 2×7 rectangle is $f(7)$. The other choice is to lay the first 2×1 at the top left corner horizontally. It has to lay another 2×1 rectangle at the lower left corner. The number of choices to cover the remaining 2×6 rectangle is $f(6)$.

We can see that $f(8)=f(7)+f(6)$, so it is also a Fibonacci Sequence.

2.
3. Given a unsorted array, find the number at a given position when the array is sorted.
   - Partition:
     - ```
       int partition(int* array,int l,int h)
       {
               int key = array[l];
               while (l < h)
               {
                       while (l < h && key <= array[h]) h--;
                       if (l < h)
                               array[l++] = array[h];
                       while (l < h && key > array[l]) l++;
                       if (l < h)
                               array[h--] = array[l];
               }
               array[l] = key;
               return l;
       }
       ```
   - Find:
       ```

```
38 //Find the number at pos when sorted.
39 void findPos(int* array,int size,int pos)
40 {
41         if (!array || size <= 0)
42                 return;
43         int low = 0;
44         int high = size - 1;
45         int index = partition(array,low,high);
46         for (int i = 0;i < size;i++)
47                 cout << array[i] << " ";
48         cout << endl;
49         while (index != pos)
50         {
51                 cout << "index: " << index << endl;
52                 if (index > pos)
53                         high = index - 1;
54                 else
55                         low = index + 1;
56                 index = partition(array,low,high);
57         }
58         cout << "pos: " << pos << ": " <<  array[index] << endl;
59 }
```

4. Imagine a robot sitting on the upper left hand corner of an NxN grid. The robot can only move in two directions: right and down. How many possible paths are there for the robot?

   FOLLOW UP

   Imagine certain squares are "off limits", such that the robot can not step on them. Design an algorithm to get all possible paths for the robot.

   - From the upper left point to the bottom right point, the total steps will consist of N right steps and N down steps. Thus, the number of possible paths can be represented as C(N,N / 2) / A(N / 2,N / 2);
   - The other way to think about this problem is that: For a position (N,M). The possible number of paths for this point is: f(N,M) which is equal to f(N - 1, M) + f(N,M - 1)…Thus, this is like a recursion..until the base case where N = 0 or M = 0 (bottom row and rightmost row where only one possible path (straight)). We can easily write a function:

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

   - For printing the possible paths. The decisions of moving right or moving down are all binary, which means the decision can be represented as a binary number. This binary number (N digits) must have exactly N / 2 1s and N / 2 0s. The possible paths are just the all permutations of this binary number.

```
//The differences between this one and findPath in the tree is that once we step onto the condition where x is n or y is n, that is the end condition, since this point
//we don't have to probe anymore beacuse there are no more possiblitie existing. But for tree, after finding a path, there might be still some paths after that point
//because some numbers are negative. We can only pop an element after we check both of the subtrees. For this one, we can pop it as soon as we find it.
void generatePaths(int n,int x,int y,vector<Point>& vec)
{
	vec.push_back(Point(x,y));
	static int count = 0;
	if (x == n || y == n)
	{
		count++;
		cout << "Found: " << count << endl;
		for (int i = 0;i < vec.size();i++)
			cout << "(" << vec[i].x << "," << vec[i].y << ")" << " ";
		if (x == n)
			while (y < n) cout << "(" << x << "," << ++y << ")" << " ";
		else
			while (x < n) cout << "(" << ++x << "," << y << ")" << " ";
		cout << endl;
		vec.pop_back();
		return;
	}
	generatePaths(n,x + 1,y,vec);
	generatePaths(n,x,y + 1,vec);
	vec.pop_back();
}
```

5. Write a method that returns all subsets of a set.
   - Consider a set {1,2,3} and its subset: {{},{1},{2},{3},{1,2},{1,3},{2,3},{1,2,3}}
   - for empty set, all three numbers are not selected, which can be represented as 000
   - for {1}, 2 and 3 are not selected, which can be represented as 100
   - for {2}, ….010
   - ..
   - for {1,2,3},… 111
   - In this case, we know that all subsets are actually from 000 - 111
   - Bear this idea in mind, we can write a program:

```
void subset(int* array,int size,vector<vector<int> >& vec)
{
	int numSets = pow(2,size);
	for (int count = 0;count < numSets;count++)
	{
		vector<int> temp;
		for (int i = 0;i < size;i++)
		{
			if (count & (1 << i))
				temp.push_back(array[i]);
		}
		vec.push_back(temp);
	}
}
```

   - time complexity: O(2 ^ n)

6. Write a method to compute all permutations of a string
   - Take an example of “abc”
   - Divide it into two parts: a and remainder bc.
   - Calculate the permutation of the remainder {bc, cb}
   - Insert the first part into any position of the remainder permutations. that is: {abc,bac,bca} {acb,cab,cba} 6 total

```
vector<string> permutation1(string s)
{
        vector<string> vec;
        if (s.empty())  {vec.push_back("");return vec;}
        char first = s[0];
        vector<string> subpermutation = permutation1(s.substr(1));
        for (int i = 0;i < subpermutation.size();i++)
        {
                for (int j = 0;j <= subpermutation[i].size();j++)
                {
                        string temp = subpermutation[i];
                        temp.insert(temp.begin() + j,first);
                        vec.push_back(temp);
                }
        }
        return vec;
}
```

- time Complexity: O(n!)
- This approach has problem when the string has duplicates. One way to solve it is to use a static variable (hash map) to keep track if the permutation is duplicate.

7. Implement an algorithm to print all valid (e.g., properly opened and closed) combinations of n-pairs of parentheses.
   - We can keep track of all available parenthesis in stock and match left and right parenthesis by tracking the number of parenthesis.
   - As long as the left parenthesis is available, put that in. The number of left parenthesis will always be equal or larger than right parenthesis at any point of the execution of the program.

```
void printParenthese1(int n,int left,int right,string s)
{
        if (left > n) return;
        if (left == n && right == n)
                cout << s << endl;
        if (left < n)
                printParenthese1(n,left + 1,right,s + "(");
        if (right < left)
                printParenthese1(n,left,right + 1,s + ")");
}
```

   - Time complexity O(n).

8. Given an infinite number of quarters (25 cents), dimes (10 cents), nickels (5 cents) and pennies (1 cent), write code to calculate the number of ways of representing n cents.
   - Use recursive methods.

```
void generateChanges1(int n,int nextPickable,int num1,int num5,int num10,int num25)
{
        if (n < 0)
                return;
        if (n == 0)
        {
                cout << "found!" << endl;
                cout << "1: " << num1 << " 5: " << num5 << " 10: " << num10 << " 25: " << num25 <<
endl;
        }
        if (nextPickable >= 25)
                generateChanges1(n - 25,25,num1,num5,num10,num25 + 1);
        if (nextPickable >= 10)
                generateChanges1(n - 10,10,num1,num5,num10 + 1,num25);
        if (nextPickable >= 5)
                generateChanges1(n - 5,5,num1,num5 + 1,num10,num25);
        if (nextPickable >= 1)
                generateChanges1(n - 1,1,num1 + 1,num5,num10,num25);
}
```

```
int numOfWayChanges(int n,int next)
{
        static int count = 0;
        if (n < 0) return 0;
        if (n == 0) {count++;return count;}
        if (next >= 25) numOfWayChanges(n - 25,25);
        if (next >= 10) numOfWayChanges(n - 10,10);
        if (next >= 5) numOfWayChanges(n - 5,5);
        if (next >= 1) numOfWayChanges(n - 1,1);
        return count;
}
```

- We have to sort the possible combinations by starting from the big coins to small coins.

9. You are given two sorted arrays, A and B, and A has a large enough buffer at the end to hold B. Write a method to merge B into A in sorted order.
    - Start from the right, use three pointers, one points to the end of the array A, one points to the end of the array B. one points to the end of the new array (array A).

```
void mergeTwoSortedArray(int* a,int sizeA,int* b,int sizeB)
{
        int pointer1 = sizeA - 1;
        int pointer2 = sizeB - 1;
        int pointer3 = pointer1 + sizeB;
        while (pointer1 >=0 && pointer2 >= 0)
        {
                if (a[pointer1] < b[pointer2])
                        a[pointer3--] = b[pointer2--];
                else
                        a[pointer3--] = a[pointer1--];
        }
        while (pointer2 >= 0)
                a[pointer3--] = b[pointer2--];
}
```

10. Given a sorted array of n integers that has been rotated an unknown number of times, give an O(log n) algorithm that finds an element in the array. You may assume that the array was originally sorted in increasing order.

    EXAMPLE:

    Input: find 5 in array (15 16 19 20 25 1 3 4 5 7 10 14)

    Output: 8 (the index of 5 in the array)

    - Use modified binary search. the mid point partitions the array into two parts, one part is sorted and the other part is not sorted. We can firstly determine if the element exist in the sorted part, if so, then easy.. just binary search. otherwise, we have to search the unsorted part and partition it into two parts again…

```
int findElementInRotatedArray1(int* array,int size,int data)
{
        int low = 0;
        int high = size - 1;
        while (low <= high)
        {
                int mid = low + (high - low) / 2;
                if (data == array[mid])
                        return mid;
                if (array[low] < array[mid])//the pre-part is ordered
                {
                        if (data < array[mid] && data >= array[low])//exist in ordered part
                                high = mid - 1;
                        else
                                low = mid + 1;//exist in non-ordered part.
                }
                else//the after-part is ordered
                {
                        if (data > array[mid] && data <= array[high])
                                low = mid + 1;//exit in the ordered part
                        else
                                high = mid - 1;//exist in the non-ordered part.
                }
        }
        return -1;
}
```

    - Time complexity : O(logn) worst case, all duplicates: O(n).

11. Quick Sort Using Partition

```
int partition(int* array,int low,int high)
{
        int i = low;
        int j = high;
        int pivot = i + (j - i) / 2;
        int val = array[pivot];
        while (i <= j)
        {
                while (i <= j && array[i] < val) i++;
                if (i <= j) {array[pivot] = array[i];pivot = i++;}
                while (i <= j && array[j] > val) j--;
                if (i <= j) {array[pivot] = array[j];pivot = j--;}
        }
        array[pivot] = val;
        return pivot;
}

void qsortWithPartition(int* array,int low,int high)
{
        int i = low;
        int j = high;
        int pivot = partition(array,i,j);
        cout << "-------------" << endl;
        cout << "pivot: " << pivot << endl;
        cout << "val: " << array[pivot] << endl;
        for (int i = 0;i < 10;i++)
                cout << array[i] << " ";
        cout << endl;
        cout << "-------------" << endl;
        if (pivot > low)
                qsortWithPartition(array,low,pivot - 1);
        if (pivot < high)
                qsortWithPartition(array,pivot + 1,high);
}
```

12. Given a sorted array of strings which is interspersed with empty strings, write a method to find the location of a given string.

    Example: find "ball" in ["at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""] will return 4
    Example: find "ballcar" in ["at","","","","","","ball","car","","","dad","","",""] will return -1

```
/*
Given a sorted array of strings which is interspersed with empty strings, write a meth- od to find the location of a given string
Example: find “ball” in [“at”, “”, “”, “”, “ball”, “”, “”, “car”, “”, “”, “dad”, “”, “”] will return 4 Example: find “ballcar” in [“at”, “”, “”, “”, “”, “ball”, “car”, “”, “”, “
dad”, “”, “”] will return -1
*/
//Key is that once encounter a white space, proceed to the right index, if doesn't exist on the rihgt, proceed to the left.
int findStringInSortedArray(const vector<string>& vec,string target,int low,int high)
{
	while (low <= high)
	{
		int mid = low + (high - low) / 2;
		cout << "mid: " << mid << " low: " << low << " high: " << high << endl;
		if (vec[mid] == target) return mid;
		if (vec[mid] == "")
		{
			int temp = mid;
			while (mid <= high && vec[mid] == "") mid++;
			cout << "here mid: " << mid << endl;
			if (mid <= high)
			{
				if (vec[mid] == target) return mid;
				if (compareStringLess(vec[mid],target))
					low = mid + 1;
				else
					high = mid - 1;
			}
			else
				high = temp - 1;
		}
		else if (compareStringLess(vec[mid],target))
			low = mid + 1;
		else
			high = mid - 1;
	}
	return -1;
}
```

13. A circus is designing a tower routine consisting of people standing atop one another's shoulders. For practical and aesthetic reasons, each person must be both shorter and lighter than the person below him or her. Given the heights and weights of each person in the circus, write a method to compute the largest possible number of people in such a tower.

    EXAMPLE:

    Input (ht, wt): (65, 100) (70, 150) (56, 90) (75, 190) (60, 95) (68, 110)

    Output: The longest tower is length 6 and includes from top to bottom: (56, 90) (60,95) (65,100) (68,110) (70,150) (75,190)

```
int findMaxSequence(Temp* a,int size)
{
	if (size <= 0) return 0;
	int maxSequence = 0;
	int count = 0;
	int preHeight = a[0].height;
	int preWeight = a[0].weight;
	for (int i = 0;i < size;i++)
	{
		if (a[i].height < preHeight || a[i].weight < preWeight) {count = 0;}
		if (a[i].height >= preHeight && a[i].weight >= preHeight) {count++; count > maxSequence? maxSequence = count : count = count;}
		preHeight = a[i].height; preWeight = a[i].weight;
	}
	return maxSequence;
}
```

    - Key is to sort the sequence by two fields, firstly the first field and secondly the second field.

14. Implement -,*,/ operations by using only ‘+’

```
float transToOpposite(float number)
{
        return -number;
}

bool isDifferentSign(float number1,float number2)
{
        if ((number1 > 0 && number2 < 0) || (number1 < 0 && number2 > 0)) return true;
        else return false;
}

float absolute(float number)
{
        return number < 0? transToOpposite(number) : number;
}

float minusMy(float number1,float number2)
{
        return number1 + transToOpposite(number2);
}

float times(float number1,float number2)
{
        float result = 0;
        int count = 0;
        bool flag = isDifferentSign(number1,number2);
        number1 = absolute(number1);
        number2 = absolute(number2);
        while (count < number2) {result += number1;count++;}
        if (flag) result = transToOpposite(result);
        return result;
}

float divide(float number1,float number2)
{
        float result = 0;
        float constant = number2;
        bool flag = isDifferentSign(number1,number2);
        number1= absolute(number1);
        number2 = absolute(number2);
        for (int i = number2;i < number1;i += number2,result++)
        if (flag) transToOpposite(result);
        return result;
}
```

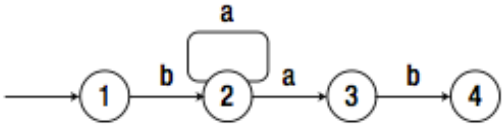
- Key is to divide tasks into sub tasks. Implement absolute, transToOpposite, isDifferentSign functions.

15. Check if a string is unique
    - Traditional way: using an array of size (256).
    - Better space-wise way: using a bit vector

We can reduce our space usage a little bit by using a bit vector. We will assume, in the below code, that the string is only lower case 'a' through 'z'. This will allow us to use just a single int

```
1   public static boolean isUniqueChars(String str) {
2       int checker = 0;
3       for (int i = 0; i < str.length(); ++i) {
4           int val = str.charAt(i) - 'a';
5           if ((checker & (1 << val)) > 0) return false;
6           checker |= (1 << val);
7       }
8       return true;
9   }
```

16. **Given millions of numbers containing duplicates and only one is unique, find it.**
17. **Find the repeating number in a list.**
    - **Use formula.**
    - **Use array index.**
18. **Find the repeating numbers in a list.**
19. Find the missing number is a list from 1 - n
    - Using XOR. -> will not get overflow because any number XORs itself will yield 0.
    - Using formula (1 + n)n / 2 -> Will get overflow
20. **Question 11** How do you implement a function to match regular expressions with '.' and '*' in patterns? The character '.' in a pattern matches a single character, and '*' matches zero or any number of characters preceding it. Matching means that a string fully matches the pattern where all characters in a string match the whole pattern. For example, the string "aaa" matches the pattern "a.a" and the pattern "ab*ac*a". However, it does not match the pattern "aa.a" nor "ab*a".



    - Say a pattern like a*b. It matches b (eg 2),ab (eg 1),aab (eg 3)
    - Then once we have matched 'a', we have two ways to do:
        - 1) keep the string unchanged and advance the pattern by 2 (e.g. 2)
        - 2) advance the string by 1 and keep the pattern unchanged (e.g. 3)
        - 3) advance string by 1 and advance pattern by 2 (e.g. 1)
    - When dealing with '.', we just consider this character as a omnipotent character that matches everything