程序员面试题精选及答案集

请粉以下公共号表示支持

作者: 顾颖琼 博士

联系 email: guyingqiong@hotmail.com

警告！

本书旨在帮助对电脑编程有兴趣的有为青年通过面试，拿到丰厚薪水，不保证该有为青年被录取后的工作表现.

作者简介:

顾颖琼，祖籍上海，1978 年出生。24 岁赴美求学，2007 年获 得圣母大学数学物理博士学位，现为电脑资深工程师。他研究生院所研究的核潮汐模型第一次将量子物理中的转动和振动有机的结合起来 。顾颖琼博士是美国电气和电子工程师资深会员，Trinity college 理学院院长。

顾颖琼博士先后在微软公司、Ebay 公司担任资深职位。 他善于发现问题，解决问题的风格为人称道。他擅长面向对象的语言设计,以及和工程师以及管理者进行多层次的交流,

顾颖琼博士非常关心热爱电脑编程的有为青年的成长，致力于帮助他们成为国际化的人才。

1. Serialize and Deserialize a Binary Tree To String

Design an algorithm and write code to serialize and deserialize a binary tree. Writing the tree to a file is called 'serialization' and reading back from the file to reconstruct the exact same binary tree is 'deserialization'.

JAVA :

```
class TreeNode{
   int val;
   TreeNode left, right;
   TreeNode(int val){
        this.val = val;
    }
}

public String serialize(TreeNode root){
    StringBuilder sb = new StringBuilder();
    serialize(root, sb);
    return sb.toString();
}

private void serialize(TreeNode x, StringBuilder sb){
    if (x == null) {
        sb.append("# ");
    } else {
        sb.append(x.val + " ");
        serialzie(x.left, sb);
        serialzie(x.right, sb);
    }
}

public TreeNode deserialize(String s){
    if (s == null || s.length() == 0) return null;
    StringTokenizer st = new StringTokenizer(s, " ");
    return deserialize(st);
}

private TreeNode deserialize(StringTokenizer st){
    if (!st.hasMoreTokens())
        return null;
    String val = st.nextToken();
    if (val.equals("#"))
        return null;
    TreeNode root = new TreeNode(Integer.parseInt(val));
    root.left = deserialize(st);
    root.right = deserialize(st);
    return root;
}
```

2. **Convert a binary search tree to a sorted double-linked list. We can only change the target of pointers, but cannot create any new nodes.**

For example, if we input a binary search tree as shown on the left side of the Figure 1, the output double-linked list is shown on the right side.

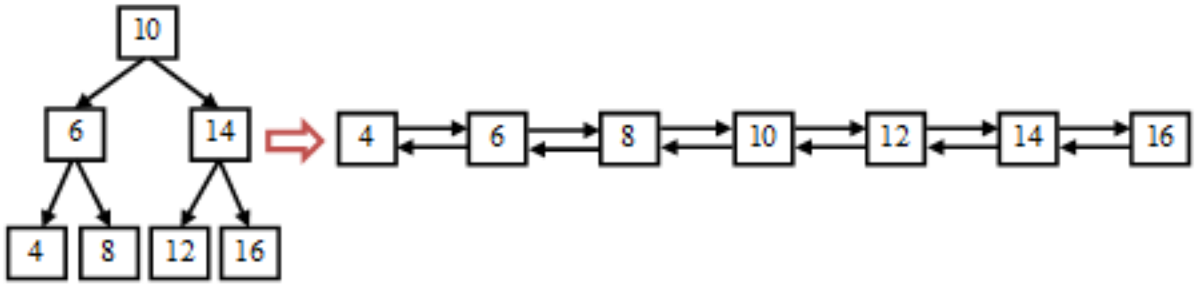


Figure 1 The conversion between a binary search tree and a sorted double-linked list

A node of binary search tree is defined in C/C++ is as:

```
struct BinaryTreeNode
{
   int                  m_nValue;
   BinaryTreeNode*      m_pLeft;
   BinaryTreeNode*      m_pRight;
};
```

Analysis: In a binary tree, each node has two pointers to its children. In a double-linked list, each node also has two pointers, one pointing to the previous node and the other pointing to the next one. Additionally, binary search tree is a sorted data structure. In a binary search tree, value in parent is always greater than value of its left child and less than value of its right child. Therefore, we can adjust a pointer to its left child in binary search tree to its previous node in a double-linked list, and adjust a pointer to its right child to its next node.

It is required that the converted list should be sorted, so we can adopt in-order traversal. That is because according to the definition of in-order traversal we traverse from nodes with less value to nodes with greater value.

C/C++

We can write the following code based on the recursive solution:

```
BinaryTreeNode* Convert(BinaryTreeNode* pRootOfTree)
{
   BinaryTreeNode *pLastNodeInList = NULL;
   ConvertNode(pRootOfTree, &pLastNodeInList);

   // pLastNodeInList points to the tail of double-linked list,
   // but we need to return its head
   BinaryTreeNode *pHeadOfList = pLastNodeInList;
   while(pHeadOfList != NULL && pHeadOfList->m_pLeft != NULL)
      pHeadOfList = pHeadOfList->m_pLeft;

   return pHeadOfList;
}
```

```
void ConvertNode(BinaryTreeNode* pNode, BinaryTreeNode** pLastNodeInList)
{
   if(pNode == NULL)
      return;

   BinaryTreeNode *pCurrent = pNode;

   if (pCurrent->m_pLeft != NULL)
      ConvertNode(pCurrent->m_pLeft, pLastNodeInList);

   pCurrent->m_pLeft = *pLastNodeInList;
   if(*pLastNodeInList != NULL)
      (*pLastNodeInList)->m_pRight = pCurrent;

   *pLastNodeInList = pCurrent;

   if (pCurrent->m_pRight != NULL)
      ConvertNode(pCurrent->m_pRight, pLastNodeInList);
}
```

3. Paths with Specified Sum in Binary Tree
Question: All nodes along children pointers from root to leaf nodes form a path in a binary tree. Given a binary tree and a number, please print out all of paths where the sum of all nodes value is same as the given number. The node of binary tree is defined as:

```
struct BinaryTreeNode
{
   int                m_nValue;
   BinaryTreeNode*      m_pLeft;
   BinaryTreeNode*      m_pRight;
};
```

For instance, if inputs are the binary tree in Figure 1 and a number 22, two paths with be printed: One is the path contains node 10 and 12, and the other contains 10, 5 and 7.

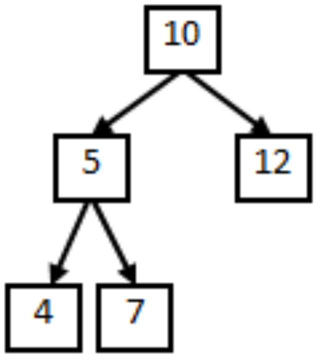


Figure : A binary tree with two paths where the sum of all nodes value is 22: One is the path contains node 10 and 12, and the other contains node 10, 5 and 7

Analysis: Path in a binary tree is a new concept for many candidates, so it is not a simple question for them. We may try to find the hidden rules with concrete examples. Let us take the binary tree in Figure 1 as an example.

Since paths always start from a root node, we traverse from a root node in a binary tree. We have three traversal orders, pre-order, in-order and post-order, and we firstly visit a root node with pre-order traversal.
Here is some sample code for this problem c/c++:

```
void FindPath(BinaryTreeNode* pRoot, int expectedSum)
{
   if(pRoot == NULL)
      return;

   std::vector<int> path;
   int currentSum = 0;
   FindPath(pRoot, expectedSum, path, currentSum);
}

void FindPath
(
   BinaryTreeNode*   pRoot,
   int               expectedSum,
   std::vector<int>& path,
   int               currentSum
)
{
   currentSum += pRoot->m_nValue;
   path.push_back(pRoot->m_nValue);

     // Print the path is the current node is a leaf
     // and the sum of all nodes value is same as expectedSum
   bool isLeaf = pRoot->m_pLeft == NULL && pRoot->m_pRight == NULL;
```

```
    if(currentSum == expectedSum && isLeaf)
    {
        printf("A path is found: ");
        std::vector<int>::iterator iter = path.begin();
        for(; iter != path.end(); ++ iter)
            printf("%d\t", *iter);

        printf("\n");
    }

    // If it is not a leaf, continue visition its children
    if(pRoot->m_pLeft != NULL)
        FindPath(pRoot->m_pLeft, expectedSum, path, currentSum);
    if(pRoot->m_pRight != NULL)
        FindPath(pRoot->m_pRight, expectedSum, path, currentSum);

    // Before returning back to its parent, remove it from path,
    path.pop_back();
}
```

4. Post-order Traversal Sequences of Binary Search Trees

Problem: Determine whether an input array is a post-order traversal sequence of a binary tree or not. If it is, return true; otherwise return false. Assume all numbers in an input array are unique.

For example, if the input array is {5, 7, 6, 9, 11, 10, 8}, true should be returned, since it is a post-order traversal sequence of the binary search tree in Figure 1. If the input array is {7, 4, 6, 5}, false should be returned since there are no binary search trees whose post-order traversal sequence is such an array.

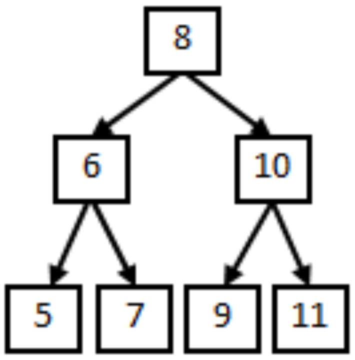


Figure : A binary search tree with post-order traversal sequence {5, 7, 6, 9, 11, 10, 8}

Analysis: The last number is a post-order traversal sequence is the value of root node. Other numbers in a sequence can be partitioned into two parts: The left numbers, which are less than the value of root node, are value of nodes in left sub-tree; the following numbers, which are greater than the value of root node, are value of nodes in right sub-tree.

Take the input {5, 7, 6, 9, 11, 10, 8} as an example, the last number 8 in this sequence is value of root node. The first 3 numbers (5, 7 and 6), which are less than 8, are value of nodes in left sub-tree. The following 3 numbers (9, 11 and 10), which are greater than 8, are value of nodes in right sub-tree.
It is not difficult to write code after we get the strategy above. Some sample code is shown below:

```
bool VerifySquenceOfBST(int sequence[], int length)
{
    if(sequence == NULL || length <= 0)
        return false;

    int root = sequence[length - 1];

    // nodes in left sub-tree are less than root node
    int i = 0;
    for(; i < length - 1; ++ i)
    {
        if(sequence[i] > root)
            break;
    }

    // nodes in right sub-tree are greater than root node
    int j = i;
    for(; j < length - 1; ++ j)
    {
        if(sequence[j] < root)
            return false;
    }

    // Is left sub-tree a binary search tree?
    bool left = true;
    if(i > 0)
        left = VerifySquenceOfBST(sequence, i);

    // Is right sub-tree a binary search tree?
    bool right = true;
    if(i < length - 1)
        right = VerifySquenceOfBST(sequence + i, length - i - 1);

    return (left && right);
}
```

5. Binary Search Tree Verification
Question: How to verify whether a binary tree is a binary search tree?

For example, the tree in Figure is a binary search tree.

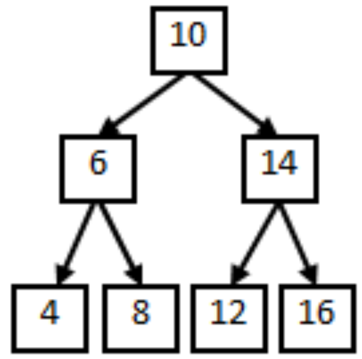


Figure 1: A binary search tree

A node in binary tree is defined as:

```
struct BinaryTreeNode
{
    int                    nValue;
    BinaryTreeNode*        pLeft;
    BinaryTreeNode*        pRight;
};
```

Analysis: Binary search tree is an important data structure. It has a specific character: Each node is greater than or equal to nodes in its left sub-tree, and less than or equal to nodes in its right sub-tree.

The following sample code is implemented based on this pre-order traversal solution:

```
bool isBST_Solution1(BinaryTreeNode* pRoot)
{
    int min = numeric_limits<int>::min();
    int max = numeric_limits<int>::max();
    return isBSTCore_Solution1(pRoot, min, max);
}

bool isBSTCore_Solution1(BinaryTreeNode* pRoot, int min, int max)
{
    if(pRoot == NULL)
        return true;

    if(pRoot->nValue < min || pRoot->nValue > max)
        return false;

    return isBSTCore_Solution1(pRoot->pLeft, min, pRoot->nValue)
        && isBSTCore_Solution1(pRoot->pRight, pRoot->nValue, max);
```

}

6. Post-order Traversal Sequences of Binary Search Trees

Problem: Determine whether an input array is a post-order traversal sequence of a binary tree or not. If it is, return true; otherwise return false. Assume all numbers in an input array are unique.

For example, if the input array is {5, 7, 6, 9, 11, 10, 8}, true should be returned, since it is a post-order traversal sequence of the binary search tree in Figure 1. If the input array is {7, 4, 6, 5}, false should be returned since there are no binary search trees whose post-order traversal sequence is such an array.

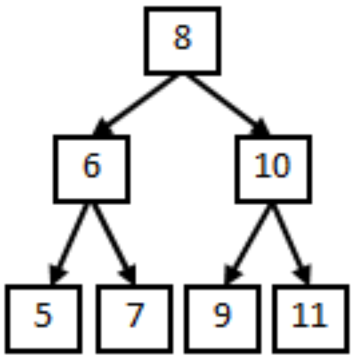


Figure : A binary search tree with post-order traversal sequence {5, 7, 6, 9, 11, 10, 8}

Analysis: The last number is a post-order traversal sequence is the value of root node. Other numbers in a sequence can be partitioned into two parts: The left numbers, which are less than the value of root node, are value of nodes in left sub-tree; the following numbers, which are greater than the value of root node, are value of nodes in right sub-tree.

Take the input {5, 7, 6, 9, 11, 10, 8} as an example, the last number 8 in this sequence is value of root node. The first 3 numbers (5, 7 and 6), which are less than 8, are value of nodes in left sub-tree. The following 3 numbers (9, 11 and 10), which are greater than 8, are value of nodes in right sub-tree.

}

We can continue to construct the left sub-tree and right sub-tree according to the two sub-arrays with the same strategy. In the subsequence {5, 7, 6}, the last number 6 is the root value of the left sub-tree. The number 5 is the value of left child since it is less than root value 6, and 7 is the value of right child since it is greater than 6. Meanwhile, the last number 10 in subsequence {9, 11, 10} is the root value of right sub-tree. The number 9 is value of left child, and 11 is value of right child accordingly.

It is not difficult to write code after we get the strategy above. Some sample code is shown below:

```
bool VerifySquenceOfBST(int sequence[], int length)
{
   if(sequence == NULL || length <= 0)
      return false;

   int root = sequence[length - 1];

   // nodes in left sub-tree are less than root node
   int i = 0;
   for(; i < length - 1; ++ i)
   {
      if(sequence[i] > root)
         break;
   }

   // nodes in right sub-tree are greater than root node
   int j = i;
   for(; j < length - 1; ++ j)
   {
      if(sequence[j] < root)
         return false;
   }

   // Is left sub-tree a binary search tree?
   bool left = true;
   if(i > 0)
      left = VerifySquenceOfBST(sequence, i);

   // Is right sub-tree a binary search tree?
   bool right = true;
   if(i < length - 1)
      right = VerifySquenceOfBST(sequence + i, length - i - 1);

   return (left && right);
}
```

7. Closest Node in a Binary Search Tree
Problem: Given a binary search tree and a value *k*, please find a node in the binary search tree whose value is closest to *k*.

For instance, in the binary search tree in Figure 1, the node closest to 29 is the node with value 28.

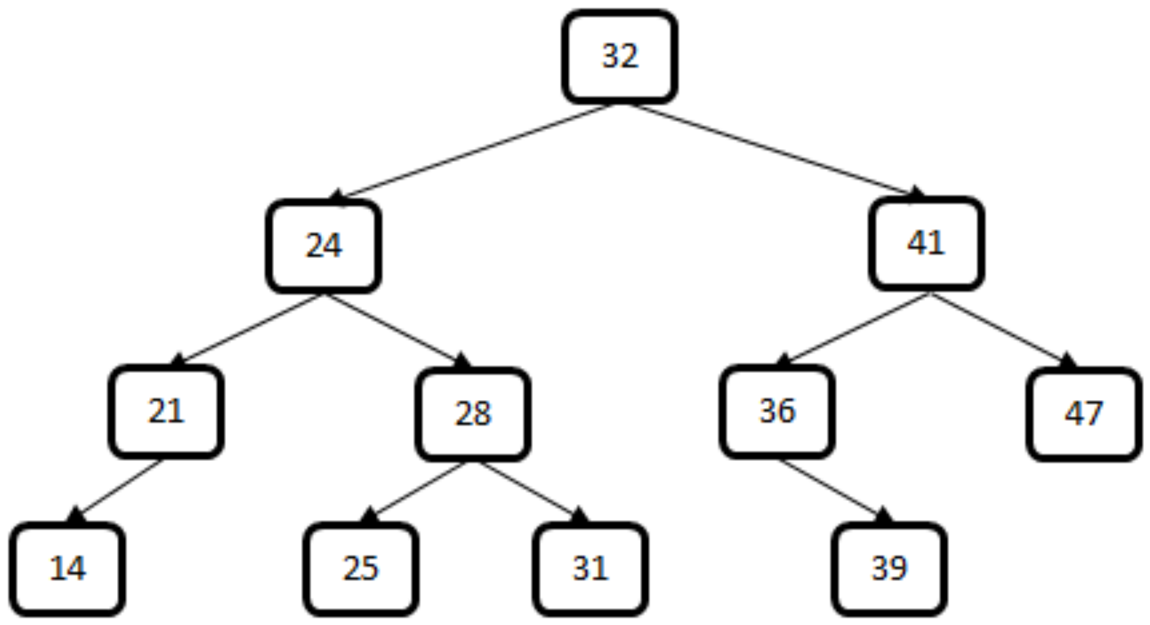


Figure : A sample binary search tree, in which the closest node to 29 is the node with value 28

Analysis: Let's analyze this problem step by step, taking the binary search in Figure 1 and 29 as an example.
We start from the root node with value 32, and the distance to 29 is 3. Since the value 32 is greater than 29, and all values in the right sub-tree are greater than 32, distances to 29 in the right sub-tree should be greater than 3. We move to the left sub-tree.

The next node to be visited contains value 24, and the distance to 29 is 5. Since 5 is greater than the previous closest distance 3, the closest node up to now remains the node with value 32. Additionally, the current value 24 is less than 29, and all values in the left sub-tree are less than 24, so distances to 29 in the left sub-tree will be greater than 5. We move on to visit the right sub-tree.

The next node to be visited contains value 28, and the distance to 29 is 1. Since 1 is less than the previous closest distance 3, the closest node is updated to the node with value 28. Additionally, the value 28 is less than 29, and all values in the left sub-tree areless than 28, so distances to 29 in the left sub-tree will be greater than 1. Let’s continue to visit the right sub-tree.

Finally we reach a left node with value 31. The distance to 29 is 2 and it is greater than the previous closest distance 1, so the closest node to 29 is still the node with value 28.

According to the step-by-step analysis above, we could implement the following code:

```
BinaryTreeNode* getClosestNode(BinaryTreeNode* pRoot, int value)
{
    BinaryTreeNode* pClosest = NULL;
    int minDistance = 0x7FFFFFFF;
    BinaryTreeNode* pNode = pRoot;
    while(pNode != NULL)
    {
        int distance = abs(pNode->m_nValue - value);
        if(distance < minDistance)
        {
            minDistance = distance;
            pClosest = pNode;
        }

        if(distance == 0)
            break;

        if(pNode->m_nValue > value)
            pNode = pNode->m_pLeft;
        else if(pNode->m_nValue < value)
            pNode = pNode->m_pRight;
    }

    return pClosest;
}
```

8. Nodes with Sum in a Binary Search Tree
Problem: Given a binary search tree, please check whether there are two nodes in it whose sum equals a given value.

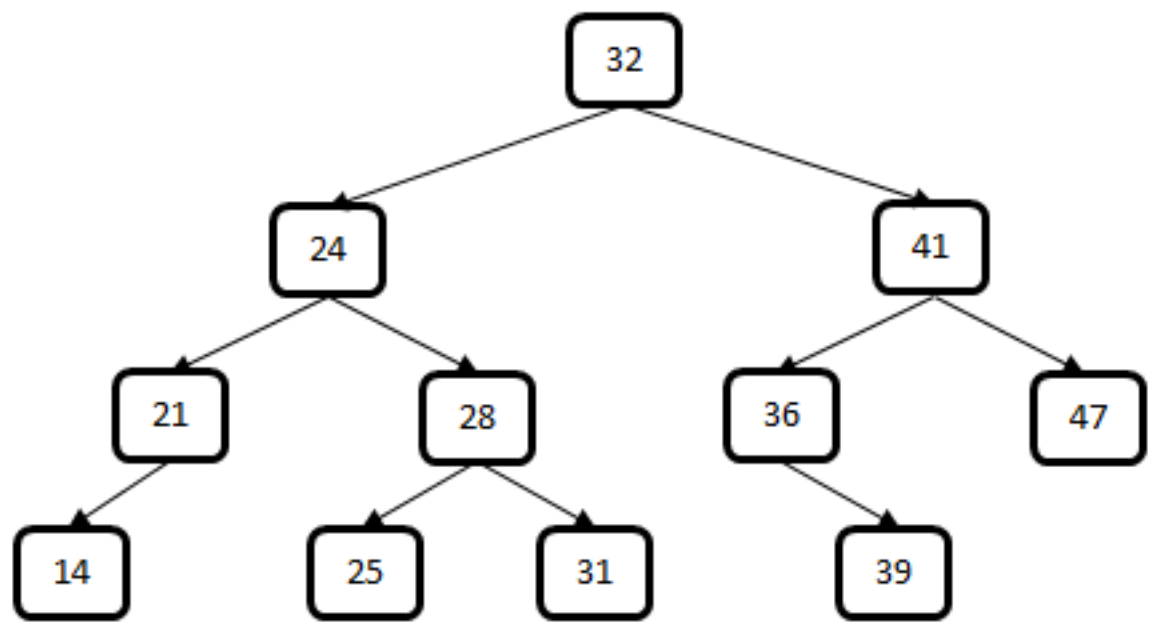


Figure : A sample binary search tree

For example, if the given sum is 66, there are two nodes in Figure 1 with value 25 and 41 whose sum is 66. While the given sum is 58, there are not two nodes whose sum is same as the given value.

Our solution can be implemented with the following C++ code:

```
bool hasTwoNodes(BinaryTreeNode* pRoot, int sum)
{
    stack<BinaryTreeNode*> nextNodes, prevNodes;
    buildNextNodes(pRoot, nextNodes);
    buildPrevNodes(pRoot, prevNodes);

    BinaryTreeNode* pNext = getNext(nextNodes);
    BinaryTreeNode* pPrev = getPrev(prevNodes);
    while(pNext != NULL && pPrev != NULL && pNext != pPrev)
    {
        int currentSum = pNext->m_nValue + pPrev->m_nValue;
        if(currentSum == sum)
            return true;

        if(currentSum < sum)
```

```
            pNext = getNext(nextNodes);
        else
            pPrev = getPrev(prevNodes);
    }

    return false;
}

void buildNextNodes(BinaryTreeNode* pRoot, stack<BinaryTreeNode*>& nodes)
{
    BinaryTreeNode* pNode = pRoot;
    while(pNode != NULL)
    {
        nodes.push(pNode);
        pNode = pNode->m_pLeft;
    }
}

void buildPrevNodes(BinaryTreeNode* pRoot, stack<BinaryTreeNode*>& nodes)
{
    BinaryTreeNode* pNode = pRoot;
    while(pNode != NULL)
    {
        nodes.push(pNode);
        pNode = pNode->m_pRight;
    }
}

BinaryTreeNode* getNext(stack<BinaryTreeNode*>& nodes)
{
    BinaryTreeNode* pNext = NULL;
    if(!nodes.empty())
    {
        pNext = nodes.top();
        nodes.pop();

        BinaryTreeNode* pRight = pNext->m_pRight;
        while(pRight != NULL)
        {
            nodes.push(pRight);
            pRight = pRight->m_pLeft;
        }
    }

    return pNext;
```

```
}

BinaryTreeNode* getPrev(stack<BinaryTreeNode*>& nodes)
{
   BinaryTreeNode* pPrev = NULL;
   if(!nodes.empty())
   {
      pPrev = nodes.top();
      nodes.pop();

      BinaryTreeNode* pLeft = pPrev->m_pLeft;
      while(pLeft != NULL)
      {
         nodes.push(pLeft);
         pLeft = pLeft->m_pRight;
      }
   }

   return pPrev;
}
```

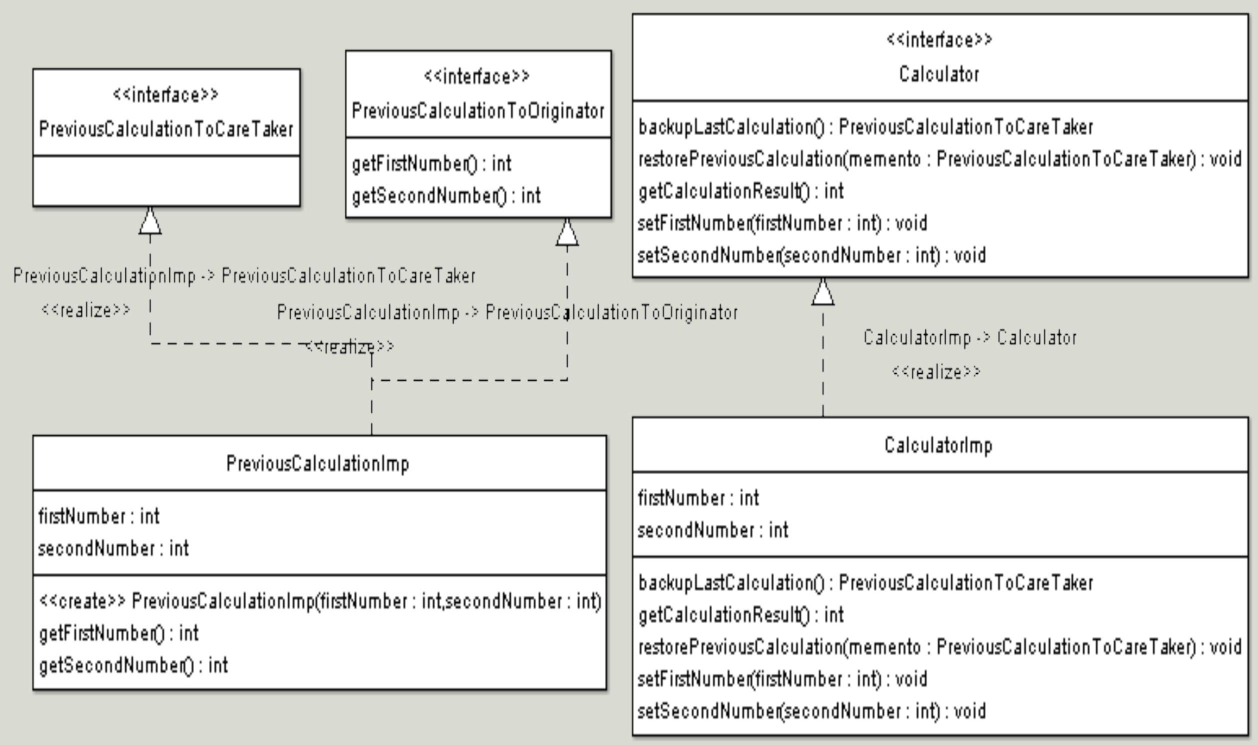
This solution costs O(*n*) time. The space complexity is O(height of tree), which is O(log*n*) on average, and O(*n*) in the worst cases.

9. write a calculator program to solve simple arithmetic problems.

Memento Pattern - Calculator Example - Java Sourcecode

Java Source Code Example for the Memento Pattern - Calculator

This simple example is a calculator that finds the result of addition of two numbers, with the additional option to undo last operation and restore previous result.



<<interface>>
PreviousCalculationToCareTaker
<<interface>>
PreviousCalculationToOriginator
getFirstNumber() : int
getSecondNumber() : int
<<interface>>
Calculator
backupLastCalculation() : PreviousCalculationToCareTaker
restorePreviousCalculation(memento : PreviousCalculationToCareTaker) : void
getCalculationResult() : int
setFirstNumber(firstNumber : int) : void
setSecondNumber(secondNumber : int) : void
PreviousCalculationImp -> PreviousCalculationToCareTaker
<<realize>>
PreviousCalculationImp -> PreviousCalculationToOriginator
<<realize>>
CalculatorImp -> Calculator
<<realize>>
PreviousCalculationImp
firstNumber : int
secondNumber : int
<<create>> PreviousCalculationImp(firstNumber : int,secondNumber : int)
getFirstNumber() : int
getSecondNumber() : int
CalculatorImp
firstNumber : int
secondNumber : int
backupLastCalculation() : PreviousCalculationToCareTaker
getCalculationResult() : int
restorePreviousCalculation(memento : PreviousCalculationToCareTaker) : void
setFirstNumber(firstNumber : int) : void
setSecondNumber(secondNumber : int) : void

The code below shows the memento object interface to caretaker. Note that this interface is a placeholder only and has no methods to honor encapsulation in that the memento is opaque to the caretaker.

```
package memento;
/**  *  Memento interface to CalculatorOperator (Caretaker)  */
public interface PreviousCalculationToCareTaker {
            // no operations permitted for the caretaker
}
```

The code below shows the Memento to Originator interface; note that this interface provides the necessary methods for the originator to restore its original state.

```
package memento;
/**  * Memento Interface to Originator  *   * This interface allows the originator to restore its sta
public interface PreviousCalculationToOriginator {
public int getFirstNumber();
public int getSecondNumber();
 }
```

The code below shows the memento implementation, note that the memento must implement two interfaces, the one to the caretaker as well as the one to the originator.

```
package memento;
 /**  * Memento Object Implementation
•   * Note that this object implements both interfaces to Originator and CareTaker  */
•   public class PreviousCalculationImp implements PreviousCalculationToCareTaker,
•           PreviousCalculationToOriginator {
•   private int firstNumber;
•   private int secondNumber;
•   public PreviousCalculationImp(int firstNumber, int secondNumber) {
•           this.firstNumber =  firstNumber;                this.secondNumber       =       secondNumber;
            }                   @Override           public int getFirstNumber() {                   return
    firstNumber;   }
•           @Override           public int getSecondNumber() {                  return          secondNumber;
            } }
```

The code below shows the calculator interface which is the originator interface

```
package memento;  /**  * Originator Interface  */
 public interface Calculator {
// Create Memento               public PreviousCalculationToCareTaker backupLastCalculation();
// setMemento       public      void      restorePreviousCalculation(PreviousCalculationToCareTa
memento);                       // Actual Services Provided by the originator    public
getCalculationResult();         public void setFirstNumber(int firstNumber);    public                 v
setSecondNumber(int secondNumber); }
```

The code below shows the Calculator implementation which is the originator implementation. Note that the backupLastCalculation method corresponds to createMemento() method discussed previously, in this method the memento object is created and all originator state is saved to the memento. Also note that the method restorePreviousCalculation() method corresponds to setMemento() method . Inside this method the logic to restore the previous state is executed.

```
package memento;
 /**  * Originator Implementation  */
public class CalculatorImp implements Calculator {
            private int firstNumber;
private int secondNumber;
@Override  public PreviousCalculationToCareTaker backupLastCalculation() {
            // create a memento object used for restoring two numbers
return new PreviousCalculationImp(firstNumber,secondNumber); }
@Override           public int getCalculationResult() {
// result is adding two numbers                     return firstNumber + secondNumber; }
@Override           public void restorePreviousCalculation(PreviousCalculationToCareTaker mer
@Override           public void setFirstNumber(int firstNumber) {
this.firstNumber =  firstNumber;        }
@Override           public void setSecondNumber(int secondNumber) {
```

```
            this.secondNumber = secondNumber;              } }
```

The code below shows the calculator driver which simulates a user using the calculator to add numbers, the user calculates a result, then enters wrong numbers, he is not satisfied with the result and he hits Ctrl + Z to undo last operation and restore previous result.

```
package memento;  /**  * CareTaker object   */ public class CalculatorDriver {
            public static void main(String[] args) {
// program starts                    Calculator calculator = new CalculatorImp();
            calculator.setFirstNumber(10);
calculator.setSecondNumber(100);
// find result
System.out.println(calculator.getCalculationResult());
// Store result of this calculation in case of error
PreviousCalculationToCareTaker memento = calculator.backupLastCalculation();
                        // calculate result
            System.out.println(calculator.getCalculationResult());
            // user hits CTRL + Z to undo last operation and see last result
            calculator.restorePreviousCalculation(memento);
// result restored
System.out.println(calculator.getCalculationResult());                } }
```

10. **Problem:** Get the Kth node from end of a linked list. It counts from 1 here, so the 1st node from end is the tail of list.

For instance, given a linked list with 6 nodes, whose value are 1, 2, 3, 4, 5, 6, its 3rd node from end is the node with value 4.

A node in the list is defined as:

```
struct ListNode
{
   int       m_nValue;
   ListNode* m_pNext;
};
```

Analysis: In a list with n nodes, its kth node from end should be the (n-k+1)th node from its head. Therefore, if we know the number of nodes n in a list, we can get the required node with n-k+1 steps from its head. How to get the number n? It is easy if we scan the whole list from beginning to end.

The solution above needs to scan a list twice: We get the total number of nodes with the first scan, and reach the kth node from end with the second scan. Unfortunately, interviewers usually expect a solution which only scans a list once.

We have a better solution to get the kth node from end with two pointers. Firstly we move a pointer (denoted as P1) k-1 steps beginning from the head of a list. And then we move another pointer (denoted as P2) beginning from the head, and continue moving the P1 forward at same speed. Since the distance of these two pointers is always k-1, P2 reaches the kth node from end when P1 reaches the tail of a list. It scans a list only once, and it is more efficient than the previous solution.

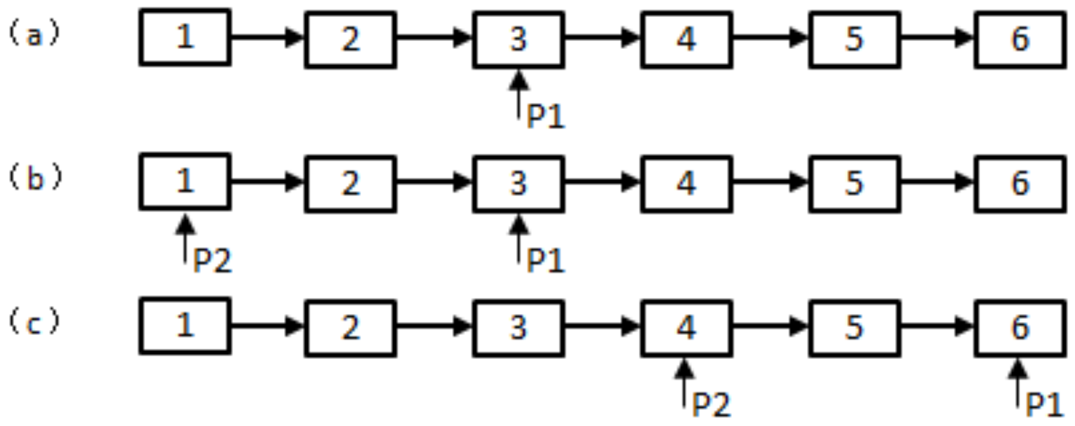


Figure 1: Get the 3rd node from end of a list with 6 nodes

It simulates the process to get the 3rd node from end of a list with 6 nodes in Figure 1. We firstly move P1 2 steps (2=3-1) to reach the 3rd node (Figure 1-a). Then P2 points to the head of a list (Figure 1-b). We move two pointers at the same speed, when the P1 reaches the tail, what P2 points is the 3rd node from end (Figure 1-c).

The sample code of the solutions with two pointers is shown below:

```
ListNode* FindKthToTail(ListNode* pListHead, unsigned int k)
{
   if(pListHead == NULL || k == 0)
      return NULL;

   ListNode *pAhead = pListHead;
   ListNode *pBehind = NULL;

   for(unsigned int i = 0; i < k - 1; ++ i)
   {
      if(pAhead->m_pNext != NULL)
         pAhead = pAhead->m_pNext;
      else
      {
            return NULL;
        }
    }

    pBehind = pListHead;
    while(pAhead->m_pNext != NULL)
    {
        pAhead = pAhead->m_pNext;
        pBehind = pBehind->m_pNext;
    }

    return pBehind;
}
```

11. **Implement a function to reverse a linked list, and return the head of the reversed list. A list node is defined as below:**

```
struct ListNode
{
      int       m_nKey;
      ListNode* m_pNext;
};
```

Analysis: Lots of pointer operations are necessary to solve problems related to linked lists. Interviewers know that many candidates are prone to make mistakes on pointer operations, so they like problems of linked list to qualify candidates' programming abilities. During interviews, we had better analyze and design carefully rather than begin to code hastily. It is much better to write robust code with comprehensive analysis than write code quickly with many errors.

Direction of pointers should be adjusted in order to reverse a linked list. We may utilize figures to analyze visually the complex steps to adjust pointers. As shown in the list in Figure 1-a, node h, i and j are three adjacent nodes. Let us assume pointers of all nodes prior to h have been reversed after some operations and all m_pNext point to their previous nodes. We are going to reverse them_pNext pointer in node i. The status of list is shown in Figure 1-b.

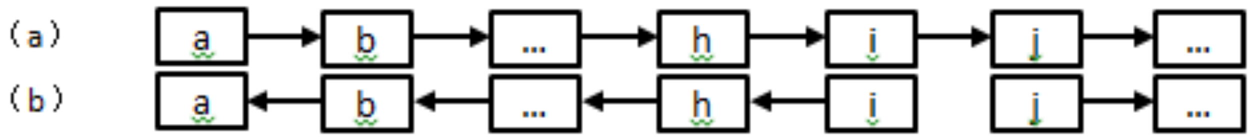


Figure 1: A list is broken when we reverse m_pNext pointers. (a) A linked list. (b) A link between node i and j is broken when all m_pNext pointers of node prior to node i point to their previous nodes.

It is noticeable that m_pNext in node i points to its previous node h, the list is broken and we cannot visit the node j anymore. We should save the node j before the m_pNext pointer of node i is adjusted to prevent the list becoming broken.

When we adjust the pointer in node i, we need to access to node h since m_pNext of node i is adjusted to point to node h. Meanwhile, we also need to access to node j because it is necessary to save it otherwise the list will be broken. Therefore, three pointers should be declared in our code, which point to the current visited node, its previous node and its next node.

Lastly we should get the head node of the reversed list. Obviously head in the reversed list should be tail of the original list. Which pointer is tail? It should be a node
whose m_pNext isNULL.
With comprehensive analysis above, we are ready to write code, which is shown below:

```
ListNode* ReverseList(ListNode* pHead)
{
   ListNode* pReversedHead = NULL;
   ListNode* pNode = pHead;
   ListNode* pPrev = NULL;
   while(pNode != NULL)
   {
      ListNode* pNext = pNode->m_pNext;

      if(pNext == NULL)
         pReversedHead = pNode;

      pNode->m_pNext = pPrev;

      pPrev = pNode;
      pNode = pNext;
   }

   return pReversedHead;
}
```

12. **How to check whether there is a loop in a linked list? For example, the list in Figure 1 has a loop.**

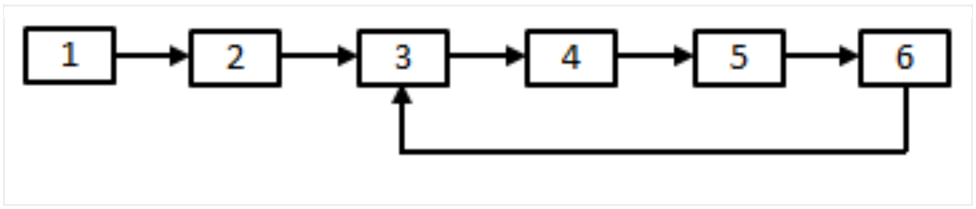


Figure 1: A list with a loop

A node in list is defined as the following structure:

```
struct ListNode
{
    int       m_nValue;
    ListNode* m_pNext;
};
```

<u>Analysis</u>: It is a popular interview question. Similar to the problem to get the Kth node from end is a list, it has a solution with two pointers.

Two pointers are initialized at the head of list. One pointer forwards once at each step, and the other forwards twice at each step. If the faster pointer meets the slower one again, there is a loop in the list. Otherwise there is no loop if the faster one reaches the end of list.

The sample code below is implemented according to this solution. The faster pointer is pFast, and the slower one is pSlow.

```
bool HasLoop(ListNode* pHead)
{
    if(pHead == NULL)
        return false;


    ListNode* pSlow = pHead->m_pNext;
    if(pSlow == NULL)
```

```
        return false;

    ListNode* pFast = pSlow->m_pNext;
    while(pFast != NULL && pSlow != NULL)
    {
        if(pFast == pSlow)
            return true;

        pSlow = pSlow->m_pNext;

        pFast = pFast->m_pNext;
        if(pFast != NULL)
            pFast = pFast->m_pNext;
    }

    return false;
}
```

13. **If there is a loop in a linked list, how to get the entry node of the loop? The entry node is the first node in the loop from head of list. For instance, the entry node of loop in the list of Figure 1 is the node with value 3.**

Analysis: Inspired by the solution of the first problem, we can also solve this problem with two pointers.

Two pointers are initialized at the head of a list. If there are *n* nodes in the loop, the first pointer forwards *n* steps firstly. And then they forward together, at same speed. When the second pointer reaches the entry node of loop, the first one travels around the loop and returns back to entry node.

Let us take the list in Figure 1 as an example. Two pointers, P1 and P2 are firstly initialized at the head node of the list (Figure 2-a). There are 4 nodes in the loop of list, so P1 moves 4 steps ahead, and reaches the node with value 5 (Figure 2-b). And then these two pointers move for 2 steps, and they meet at the node with value 3, which is the entry node of the loop.

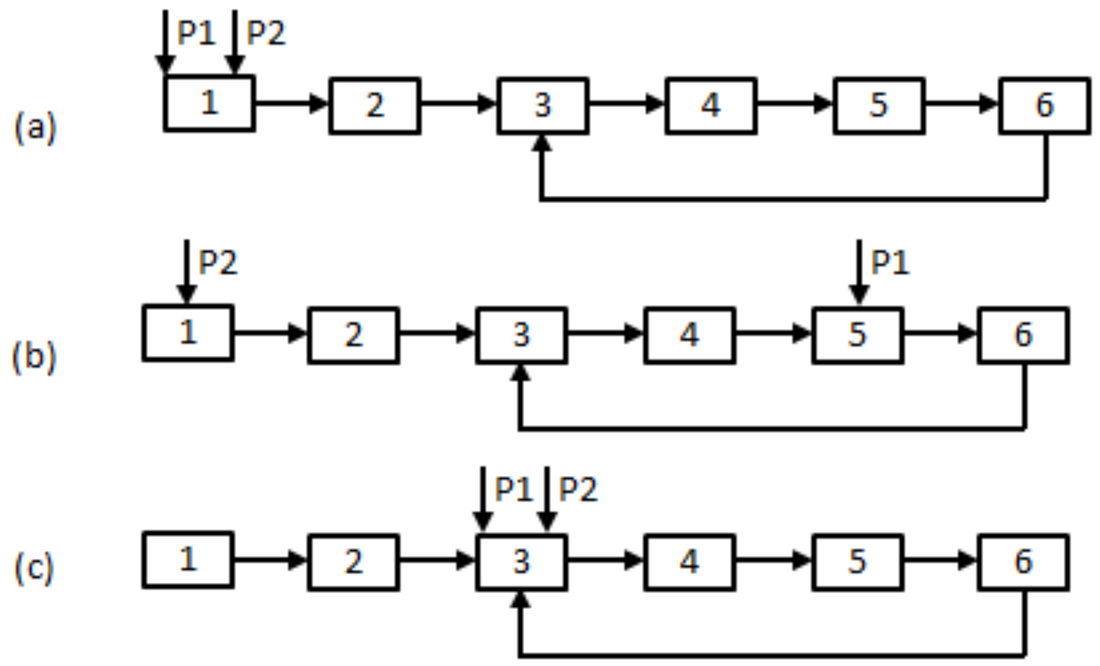


Figure 2: Process to find the entry node of a loop in a list. (a) Pointers P1 and P2 are initialized at the head of list; (b) The point P1 moves 4 steps ahead, since there are 4 nodes in the loop; (c) P1 and P2 move for two steps, and meet each other.

The only problem is how to get the numbers in a loop. Let go back to the solution of the first question. We define two pointers, and the faster one meets the slower one if there is a loop. Actually, the meeting node should be inside the loop. Therefore, we can move forward from the meeting node and get the number of nodes in the loop when we arrive at the meeting node again.

The following function MeetingNode gets the meeting node of two pointers if there is a loop in a list, which is a minor modification of the previous HasLoop:

```
ListNode* MeetingNode(ListNode* pHead)
{
   if(pHead == NULL)
      return NULL;

   ListNode* pSlow = pHead->m_pNext;
   if(pSlow == NULL)
      return NULL;

   ListNode* pFast = pSlow->m_pNext;
   while(pFast != NULL && pSlow != NULL)
   {
      if(pFast == pSlow)
```

```
            return pFast;

        pSlow = pSlow->m_pNext;

        pFast = pFast->m_pNext;
        if(pFast != NULL)
            pFast = pFast->m_pNext;
    }

    return NULL;
}
```

We can get the number of nodes in a loop of a list, and the entry node of loop after we know the meeting node, as shown below:

```
ListNode* EntryNodeOfLoop(ListNode* pHead)
{
    ListNode* meetingNode = MeetingNode(pHead);
    if(meetingNode == NULL)
        return NULL;

    // get the number of nodes in loop
    int nodesInLoop = 1;
    ListNode* pNode1 = meetingNode;
    while(pNode1->m_pNext != meetingNode)
    {
        pNode1 = pNode1->m_pNext;
        ++nodesInLoop;
    }

    // move pNode1
    pNode1 = pHead;
    for(int i = 0; i < nodesInLoop; ++i)
        pNode1 = pNode1->m_pNext;

    // move pNode1 and pNode2
    ListNode* pNode2 = pHead;
    while(pNode1 != pNode2)
    {
        pNode1 = pNode1->m_pNext;
        pNode2 = pNode2->m_pNext;
    }

    return pNode1;
}
```

14. **Nodes in a list represent a number. For example, the nodes in Figure 1 (a) and (b) represent numbers 123 and 4567 respectively. Please implement a function/method to add numbers in two lists, and store the sum into a new list.**

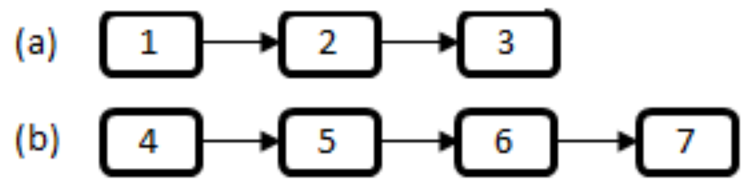


Figure 1: Two lists representing numbers. (a) A list for 123; (b) A list for 4567.

Analysis: Usually numbers are added beginning from the least significant digits (The digit 3 in the number 123, and the digit 7 in the number 4567). As shown in Figure 1, the least significant digits are at the tail of lists, and they can be accessed after the whole lists are scanned. Therefore, lists should be reversed at first, in order get the least significant digits before other digits. The two reversed lists of lists in Figure 1 are shown in Figure 2.

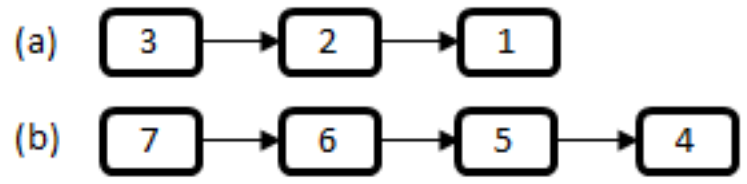


Figure 2: Two reversed lists of the lists in Figure 1.

After two lists are reversed, we can add nodes along the links between nodes, and then reversed the result list after all nodes are added. Therefore, the overall structure to add numbers in two lists can be implemented with the following code in C/C++:

```
ListNode* Add(ListNode* pHead1, ListNode* pHead2)
{
   if(pHead1 == NULL || pHead2 == NULL)
      return NULL;

   pHead1 = Reverse(pHead1);
   pHead2 = Reverse(pHead2);

   ListNode* pResult = AddReversed(pHead1, pHead2);
   return Reverse(pResult);
}
```

Now let's implement the function AddReversed, to add nodes in two reversed lists. Digits are gotten in nodes along links between nodes. When we get two digits in two lists, we add them and create a new node to store the sum, and append the new node into the list for result. There are two issues worthy of attention: (1) The length of two lists might be different; (2) The sum of two digits may be greater than 10, so we have to take care of the carry when adding two digits. The function AddReversed can be implemented with the following C/C++ code:

```
ListNode* AddReversed(ListNode* pHead1, ListNode* pHead2)
{
   int carry = 0;
   ListNode* pPrev = NULL;
   ListNode* pHead = NULL;
   while(pHead1 != NULL || pHead2 != NULL)
   {
      ListNode* pNode = AddNode(pHead1, pHead2, &carry);
      AppendNode(&pHead, &pPrev, pNode);

      if(pHead1 != NULL)
         pHead1 = pHead1->m_pNext;
      if(pHead2 != NULL)
         pHead2 = pHead2->m_pNext;
   }
   if(carry > 0)
   {
      ListNode* pNode = CreateListNode(carry);
      AppendNode(&pHead, &pPrev, pNode);
   }

   return pHead;
}
```

The function AddNode adds digits in two nodes. The third parameter of this function takes the carry for addition calculation, as listed below:

```
ListNode* AddNode(ListNode* pNode1, ListNode* pNode2, int* carry)
{
   int num1 = 0;
   if(pNode1 != NULL)
      num1 = pNode1->m_nValue;
   int num2 = 0;
   if(pNode2 != NULL)
      num2 = pNode2->m_nValue;

   int sum = num1 + num2 + *carry;
   *carry = (sum >= 10) ? 1 : 0;

   int value = (sum >= 10) ? (sum - 10) : sum;
   return CreateListNode(value);
}
```

The function AppendNode is used append a node into the tail of lists. In order to avoid scanning the whole list to get the previous tail every time, the previous tails is stored in the parameter/variable pPrev, as listed in the following code:

```
void AppendNode(ListNode** pHead, ListNode** pPrev, ListNode* pNode)
{
   if(*pHead == NULL)
      *pHead = pNode;
   if(*pPrev == NULL)
      *pPrev = pNode;
   else
   {
      (*pPrev)->m_pNext = pNode;
      *pPrev = pNode;
   }
}
```

The function CreateListNode is to create a list node according to a value, which is omitted here because it's quite straightforward.

15. Reverse words in a sentence

Problem: Reverse the order of words in a sentence, but keep words themselves unchanged. Words in a sentence are divided by blanks. For instance, the reversed output should be "student. a am I" when the input is "I am a student".

Analysis: This is a very popular interview question of many companies. It can be solved with two steps: Firstly we reverse all characters in a sentence. If all characters in sentence "I am a student." are reversed, it becomes ".tneduts a ma I". Not only the order of words is reversed, but also the order of characters inside each word is reversed. Secondly, we reverse characters in every word. We can get "student. a am I" from the example input string with these two steps.

The key of our solution is to implement a function to reverse a string, which is shown as the Reverse function below:

```
void Reverse(char *pBegin, char *pEnd)
```

```
{
    if(pBegin == NULL || pEnd == NULL)
        return;

    while(pBegin < pEnd)
    {
        char temp = *pBegin;
        *pBegin = *pEnd;
        *pEnd = temp;

        pBegin ++, pEnd --;
    }
}
```

Now we can reverse the whole sentence and each word based on this Reverse function with the following code:

```
char* ReverseSentence(char *pData)
{
    if(pData == NULL)
        return NULL;

    char *pBegin = pData;

    char *pEnd = pData;
```

```
while(*pEnd != '\0')
    pEnd ++;
pEnd--;

// Reverse the whole sentence
Reverse(pBegin, pEnd);

// Reverse every word in the sentence
pBegin = pEnd = pData;
while(*pBegin != '\0')
{
    if(*pBegin == ' ')
    {
        pBegin ++;
        pEnd ++;
    }
    else if(*pEnd == ' ' || *pEnd == '\0')
    {
        Reverse(pBegin, --pEnd);
        pBegin = ++pEnd;
    }
    else
    {
        pEnd ++;
    }
    }

    return pData;
}
```

Since words are separated by blanks, we can get the beginning position and ending position of each word by scanning blanks. In the second phrase to reverse each word in the sample code above, the pointer pBegin points to the first character of a word, and pEnd points to the last character.

16. First Character Appearing Only Once

Problem: Implement a function to find the first character in a string which only appears once.

For example: It returns ‘b’ when the input is “abaccdeff”.

Analysis: Our native solution for this problem may be scanning the input string from its beginning to end. We compare the current scanned character with every one behind it. If there is no duplication after it, it is a character appearing once. Since it compares each character with O(n) ones behind it, the overall time complexity is O(n^2) if there are n characters in a string.

In order to get the numbers of occurrence times of each character in a string, a data container is needed. It is required to get and update the occurrence time of each character in a string, so the data container is used to project a character to a number. Hash tables fulfill this kind of requirement. We can implement a hash table, in which keys are characters and values are their occurrence times in a string.

It is necessary to scan strings twice: When a character is visited, we increase the corresponding occurrence time in the hash table during the first scanning. In second round of scanning, whenever a character is visited we also check its occurrence time in the hash table. The first character with occurrence time 1 is the required output.

Hash tables are complex, and they are not implemented in the C++ standard template library. Therefore, we have to implement one by ourselves.

Characters have 8 bits, so there only 256 variances. We can create an array with 255 numbers, in which indexes are ASCII values of all characters, and numbers are their occurrence times in a string. That is to say, we have a hash table whose size if 256, with ASCII values of characters as keys.

It is time for programming after we get a clear solution. The following are some sample code:

```
char FirstNotRepeatingChar(char* pString)
{
    if(pString == NULL)
        return '\0';

    const int tableSize = 256;
    unsigned int hashTable[tableSize];
    for(unsigned int i = 0; i<tableSize; ++ i)
        hashTable[i] = 0;

    char* pHashKey = pString;
    while(*(pHashKey) != '\0')
        hashTable[*(pHashKey++)] ++;

    pHashKey = pString;
    while(*pHashKey != '\0')
    {
        if(hashTable[*pHashKey] == 1)
            return *pHashKey;

        pHashKey++;
    }

    return '\0';
}
```

In the code above, it costs O(1) time to increase the occurrence time for each character. The time complexity for the first scanning is O(n) if the length of string is n. It takes O(1) time to get the occurrence time for each character, so it costs O(n) time for the second scanning. Therefore, the overall time it costs is O(n).

In the meantime, an array with 256 numbers is created, whose size is 1K. Since the size of array is constant, the space complexity of this algorithm is O(1).

17. Left Rotation of String

Problem: Left rotation of a string is to move some leading characters to its tail. Please implement a function to rotate a string.

For example, if the input string is “abcdefg” and a number 2, the rotated result is “cdefgab”.

Analysis: It looks difficult to get rules of left rotation on a string. Fortunately, the 7th problem in this series "Reverse Words in a Sentence" can give us some hints.

If we input a sentence with two words "hello world" for the problem "Reverse Words in a Sentence", the reversed result should be "world hello". It is noticeable that the result "world hello" can be viewed as a rotated result of "hello world". It becomes "world hello" when we move some leading characters of string "hello world" to its ending. Therefore, this problem is quite similar to problem "Reverse Words in a Sentence".

Let us take a string "abcdefg" as an example. We divide it into two parts: the first part contains the two leading characters "ab", and the second part contains all other characters "cdefg". We firstly reverse these two parts separately, and the whole string becomes "bagfedc". It becomes "cdefgab" if we reverse the whole string, which is the expected result of left rotation with 2.

According to the analysis above, we can see that left rotation of a string can be implemented calling a Reverse function three times to reverse a segment or whole string. The sample code is shown below:

```
char* LeftRotateString(char* pStr, int n)
{
    if(pStr != NULL)
    {
        int nLength = static_cast<int>(strlen(pStr));
        if(nLength > 0 && n > 0 && n < nLength)
        {
            char* pFirstStart = pStr;
            char* pFirstEnd = pStr + n - 1;
            char* pSecondStart = pStr + n;
            char* pSecondEnd = pStr + nLength - 1;
            // Reverse the n leading characters
            Reverse(pFirstStart, pFirstEnd);
            // Reverse other characters
            Reverse(pSecondStart, pSecondEnd);
            // Reverse the whole string
            Reverse(pFirstStart, pSecondEnd);
        }
    }

    return pStr;
}
```

The function Reverse is shown in “Reverse Words in a Sentence”, so we are not going to repeat it here.

18. Minimal Number of Palindromes on a String

Problem: A string can be partitioned into some substrings, such that each substring is a palindrome. For example, there are a few strategies to split the string “abbab” into palindrome substrings, such as: “abba”|”b”, “a”|”b”|”bab” and “a”|”bb”|”a”|”b”.

Given a string *str*, please get the minimal numbers of splits to partition it into palindromes. The minimal number of splits to partition the string “abbab” into a set of palindromes is 1.

Analysis: This is a typical problem which can be solved by dynamic programming. We have two strategies to analyze and solve this problem

Solution 1: Split at any space between two characters

Given a substring of *str*, starting from the index *i* and ending at the index *j* (denoted as *str*[*i*:*j*]), we define a function *f*(*i*, *j*) to denote the minimal number of splits to partition the substring *str*[*i*:*j*] into a set of palindromes. If the substring is a palindrome itself, we don't have to split so *f*(*i*, *j*) is 0. If the substring is not a palindrome, the substring is split between two characters *k* and *k*+1. *f*(*i*, *j*)= *f*(*i*,*k*)+ *f*(*k*+1, *j*)+1 under such conditions. Therefore, *f*(*i*, *j*) can be defined with the following equation:

$$f(i,j) = \begin{cases} 0 & \text{if } str[i:j] \text{is a palindr} \\ \min[f(i,k) + \ f(k+1,j) + 1 \ (i \le k < j)] & otherwise \end{cases}$$

The value of *f*(0, *n*-1) is the value of the minimal number of splits to partition *str* into palindromes, if *n* is the length of *str*.

If the equation is calculated recursively, its complexity grows exponentially with the length *n*. A better choice is to calculate in bottom-up order with a 2D matrix with size *n*×*n*. The following C++ code implements this solution:

```
int minSplit_1(const string& str)
{
    int length = str.size();
    int* split = new int[length * length];

    for(int i = 0; i < length; ++i)
        split[i * length + i] = 0;

    for(int i = 1; i < length; ++i)
```

```
{
    for(int j = length - i; j > 0; --j)
    {
        int row = length - i - j;
        int col = row + i;
        if(isPalindrome(str, row, col))
        {
            split[row * length + col] = 0;
        }
        else
        {
            int min = 0x7FFFFFFF;
            for(int k = row; k < col; ++k)
            {
                int temp1 = split[row * length + k];
                int temp2 = split[(k + 1) * length + col];
                if(min > temp1 + temp2 + 1)
                    min = temp1 + temp2 + 1;
            }
            split[row * length + col] = min;
        }
    }
}

int minSplit = split[length - 1];
```

```
    delete[] split;

    return minSplit;

}
```

Solution 2: Split only before a palindrome

We split the string *str* with another strategy. Given a substring ending at the index *i*, *str*[0, i], we do not have to split if the substring is a palindrome itself. Otherwise it is split between two characters at index *j* and *j*+1 only if the substring *str*[*j*+1,*i*] is a palindrome. Therefore, an equation*f*(*i*) can be defined as the following:

$$f(i) = \begin{cases} 0 & str[0,i] \text{ is a palindrome} \\ \min[f(j) + 1 \ (0 \le j < i)] & otherwise\ if\ str[j+1,i] \text{is a palindron} \end{cases}$$

The value of *f*(*n*-1) is the value of the minimal number of splits to partition *str* into palindromes, if*n* is the length of *str*.

We could utilize a 1D array to solve this equation in bottom-up order, as listed in the following code:

```
int minSplit_2(const string& str)

{

    int length = str.size();

    int* split = new int[length];

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

        split[i] = i;


    for(int i = 1; i < length; ++i)
    {
        if(isPalindrome(str, 0, i))
        {
            split[i] = 0;
            continue;
        }

        for(int j = 0; j < i; ++j)
        {
            if(isPalindrome(str, j + 1, i) && split[i] > split[j] + 1)
                split[i] = split[j] + 1;
        }
    }

    int minSplit = split[length - 1];
    delete[] split;
    return minSplit;
}
```

Optimization to verify palindromes:

Usually it costs O(*n*) time to check whether a string with length *n* is a palindrome, and the typical implementation looks like the following code:

```
bool isPalindrome(const string& str, int begin, int end)
```

```
{
   for(int i = begin; i < end - (i - begin); ++i)
   {
      if(str[i] != str[end - (i - begin)])
         return false;
   }

   return true;
}
```

Both solutions above cost O(n^3) time. The first solution contains three nesting for-loops. The function isPalindrome is inside two nesting for-loops.

If we could reduce the cost of isPalindrome to O(1), the time complexity of the second solution would be O(n^2).

Notice that the substring *str*[*i*,*j*] is a palindrome only if the characters at index *i* and *j*, and *str*[*i*+1,*j*-1] is also a palindrome. We could build a 2D table accordingly to store whether every substring of*str* is a palindrome or not during the preprocessing. With such a table, the function isPalindrome can verify the substring *str*[*i*,*j*] in O(1) time.

19. Translating Numbers to Strings

Question: Given a number, please translate it to a string, following the rules: 1 is translated to 'a', 2 to 'b', …, 12 to 'l', …, 26 to 'z'. For example, the number 12258 can be translated to "abbeh", "aveh", "abyh", "lbeh" and "lyh", so there are 5 different ways to translate 12258. How to write a function/method to count the different ways to translate a number?

Analysis: Let's take the number 12258 as an example to analyze the steps to translate from the beginning character to the ending one. There are two possible first characters in the translated string. One way is to split the number 12258 into 1 and 2258 two parts, and 1 is translated into 'a'. The other way is to split the number 12258 into 12 and 258 two parts, and 12 is translated into 'l'.

When the first one or two digits are translated into the first character, we can continue to translate the remaining digits. Obviously, we could write a recursive function/method to translate.

Let's define a function $f(i)$ as the count of different ways to translate a number starting from the i^{th} digit, $f(i)=g(i)*f(i+1)+h(i, i+1)*f(i+2)$. The function g(i) gets 1 when the i^{th} digit is in the range 1 to 9 which can be converted to a character, otherwise it gets 0. The function $h(i, i+1)$ gets 1 the i^{th} and $(i+1)^{th}$ digits are in the range 10 to 26 which can also be converted to a character. A single digit 0 can't be converted to a character, and two digits starting with a 0, such as 01 and 02, can't be converted either.

Even though the problem is analyzed with recursion, recursion is not the best approach because of overlapping sub-problems. For example, The problem to translate 12258 is split into two sub-problems: one is to translate 1 and 2258, and the other is to translate 12 and 258. In the next step during recursion, the problem to translate 2258 can also split into two sub-problems: one is to translate 2 and 258, and the other is to translate 22 and 58. Notice the sub-problem to translate 258 reoccurs.

Recursion solves problem in the top-down order. We could solve this problem in the bottom-up order, in order to eliminate overlap sub-problems. That's to say, we start to translate the number from the ending digits, and then move from right to left during translation.

The following is the C# code to solve this problem:

```
public static int GetTranslationCount(int number)
{
   if (number <= 0)
```

```
    {
        return 0;
    }

    string numberInString = number.ToString();
    return GetTranslationCount(numberInString);
}

private static int GetTranslationCount(string number)
{
    int length = number.Length;
    int[] counts = new int[length];

    for (int i = length - 1; i >= 0; --i)
    {
        int count = 0;
        if (number[i] >= '1' && number[i] <= '9')
        {
            if (i < length - 1)
            {
                count += counts[i + 1];
            }
            else
            {
                count += 1;
            }
        }

        if (i < length - 1)
        {
            int digit1 = number[i] - '0';
            int digit2 = number[i + 1] - '0';
            int converted = digit1 * 10 + digit2;
            if (converted >= 10 && converted <= 26)
            {
                if (i < length - 2)
                {
                    count += counts[i + 2];
                }
                else
                {
                    count += 1;
                }
            }
        }

        counts[i] = count;
    }

    return counts[0];
```

```
}
```

In order to simply the code implementation, we first convert the number into a string, and then translate.

20. Turning Number in an Array

Problem: Turning number is the maximum number in an array which increases and then decreases. This kind of array is also named unimodal array. Please write a function which gets the index of the turning number in such an array.

For example, the turning number in array {1, 2, 3, 4, 5, 10, 9, 8, 7, 6} is 10, so its index 5 is the expected output.

Analysis: As we know, the binary search algorithm is suitable to search a number in a sorted array. Since the input array for this problem is partially sorted, we may also have a try with binary search.

Let us try to get the middle number in an array. The middle number of array {1, 2, 3, 4, 5, 10, 9, 8, 7, 6} is 5 (the fourth number). It is greater than its previous number 4, and less than its next number 10, so it is in the increasing sub-array. Therefore, numbers before 5 can be discarded in the next round of search.

The remaining numbers for the next round of search are {5, 10, 9, 8, 7, 6}, and the number 9 is in the middle of them. Since 9 is less than is previous number 10 and greater than its next number 8, it is in the decreasing sub-array. Therefore, numbers after 9 can be discarded in the next round of search.

The remaining numbers for the next round of search are {5, 10, 9}, and the number 10 is in the middle. It is noticeable that number 10 is greater than its previous number 5 and greater than its next number 9, so it is the maximum number. That is to say, the number 10 is the turning number in the input array.

We can see the process above is actually a classic binary search. Therefore, we can implement the required function based on binary search algorithm, as listed below:

```
int TurningNumberIndex(int* numbers, int length)
{
    if(numbers == NULL || length <= 2)
        return -1;

    int left = 0;
    int right = length - 1;
    while(right > left + 1)
    {
        int middle = (left + right) / 2;
        if(middle == 0 || middle == length - 1)
            return -1;

        if(numbers[middle] > numbers[middle - 1] &&
            numbers[middle] > numbers[middle + 1])
            return middle;
        else if(numbers[middle] > numbers[middle - 1] &&
            numbers[middle] < numbers[middle + 1])
            left = middle;
        else
```

```
            right = middle;
        }

        return -1;
    }
```

21. Intersection of Sorted Arrays

Problem: Please implement a function which gets the intersection of two sorted arrays. Assuming numbers in each array are unique.

For example, if the two sorted arrays as input are {1, 4, 7, 10, 13} and {1, 3, 5, 7, 9}, it returns an intersection array with numbers {1, 7}.

Analysis: An intuitive solution for this problem is to check whether every number in the first array (denoted as array1) is in the second array (denoted as array2). If the length of array1 is *m*, and the length of array2 is *n*, its overall time complexity is O(*m***n*) based on linear search. We have two better solutions.

*Solution 1: With O(*m+n*) Time*

It is noticeable that the two input arrays are sorted. Supposing a number number1 in array1equals to a number number2 in array2, the numbers after number1 in array1 should be greater than the numbers before number2 in array2. Therefore, it is not necessary to compare the numbers after number1 in array1 with numbers before number2 in array2. It improves efficiency since many comparisons are eliminated.

The sample code for this solution is shown below:

```
void GetIntersection_solution1(const vector<int>& array1,
                    const vector<int>& array2,
                    vector<int>& intersection)
{
   vector<int>::const_iterator iter1 = array1.begin();
   vector<int>::const_iterator iter2 = array2.begin();

   intersection.clear();

   while(iter1 != array1.end() && iter2 != array2.end())
   {
      if(*iter1 == *iter2)
      {
         intersection.push_back(*iter1);
         ++ iter1;
         ++ iter2;
      }
      else if(*iter1 < *iter2)
         ++ iter1;
      else
         ++ iter2;
   }
}
```

Since it only requires to scan two arrays once, its time complexity is O(*m*+*n*).

Solution 2: With O(*nlogm*) Time

As we know, a binary search algorithm requires O(log*m*) time to find a number in an array with length *m*. Therefore, if we search each number of an array with length *n* from an array with length*m*, its overall time complexity is O(*n*log*m*). If *m* is much greater than *n*, O(*n*log*m*) is actually less than O(*m*+*n*). Therefore, we can implement a new and better solution based on binary search in such a situation.

For instance, the following same code is suitable when array1 is much longer than array2.

```
/* === Supposing array1 is much longer than array2 === */
void GetIntersection_solution2(const vector<int>& array1,
                 const vector<int>& array2,
                 vector<int>& intersection)
{
   intersection.clear();

   vector<int>::const_iterator iter1 = array1.begin();
   while(iter1 != array1.end())
   {
      if(binary_search(array2.begin(), array2.end(), *iter1))
         intersection.push_back(*iter1);
   }
}
```

22. String Path in Matrix

Question: How to implement a function to check whether there is a path for a string in a matrix of characters? It moves to left, right, up and down in a matrix, and a cell for a movement. The path can start from any entry in a matrix. If a cell is occupied by a character of a string on the path, it cannot be occupied by another character again.

For example, the matrix below with three rows and four columns has a path for the string "BCCED" (as highlighted in the matrix). It does not have a path for the string "ABCB", because the first "B" in the string occupies the "B" cell in the matrix, and the second "B" in the string cannot enter into the same cell again.

A	B	C	E
S	F	C	S
A	D	E	E

Analysis: It is a typical problem about backtracking, which can be solved by storing a path into a stack.

Firstly, it is necessary to define a structure for 2-D positions, as below:

```
struct Position
{
   int x;
   int y;
};
```

The movements of four directions can be defined accordingly:

```
Position up = {0, -1};
Position right = {1, 0};
Position down = {0, 1};
Position left = {-1, 0};
Position dir[] ={up, right, down, left};
```

Since paths can start from any entry in a matrix, we have to scan every cell to check whether the character in it is identical to the first character of the string. If it is identical, we begin to explore a path from such a cell.

A path is defined as a stack. When a cell on path is found, we push its position into the stack. Additionally, we also define a matrix of Boolean masks to void entering a cell twice, which is denoted as visited. Based on these considerations, the skeleton of solution can be implemented as the following:

```
bool hasPath(char* matrix, int rows, int cols, char* str)
{
  if(matrix == NULL || rows < 1 || cols < 1 || str == NULL)
    return false;

  bool *visited = new bool[rows * cols];
  memset(visited, 0, rows * cols);

  for(int row = 0; row < rows; ++row)
  {
        for(int column = 0; column < cols; ++column)
        {
            if(matrix[row * cols + column] != str[0])
                continue;

            std::stack<Position> path;
            Position position = {column, row};
            path.push(position);
            visited[row * cols + column] = true;

            if(hasPathCore(matrix, rows, cols, str, path, visited))
                return true;
        }
    }

    return false;
}
```

Now let us analyze how to explore a path in details. Supposing we have already found *k*characters on a path, and we are going to explore the next step. We stand at the cell corresponding to the *k*th character of the path, and check whether the character in its neighboring cell at up, right, down, and left side is identical to the (*k*+1)th character of the string.

If there is a neighboring cell whose value is identical to the (*k*+1)th character of the string, we continue exploring the next step.

If there is no such a neighboring cell whose value is identical to the (*k*+1)th character of the string, it means the cell corresponding to the *k*th character should not on a path. Therefore, we pop it off a path, and start to explore other directions on the (*k*-1)th character.

Based on the analysis above, the function hasPathCore can be defined as:

```
bool hasPathCore(char* matrix, int rows, int cols, char* str, std::stack<Position>& path, bool* visited)
{
    if(str[path.size()] == '\0')
        return true;

    if(getNext(matrix, rows, cols, str, path, visited, 0))
        return hasPathCore(matrix, rows, cols, str, path, visited);

    bool hasNext = popAndGetNext(matrix, rows, cols, str, path, visited);
    while(!hasNext && !path.empty())
        hasNext = popAndGetNext(matrix, rows, cols, str, path, visited);

    if(!path.empty())
        return hasPathCore(matrix, rows, cols, str, path, visited);

    return false;
}
```

The function getNext is defined to explore the (k+1)th character on a path. When it returnstrue, it means the (k+1)th character on a path has been found. Otherwise, we have to pop the$k$th character off. The function getNext is implemented as below:

```
bool getNext(char* matrix, int rows, int cols, char* str, std::stack<Position>& path, bool*
visited, int start)

{

   for(int i = start; i < sizeof(dir) / sizeof(Position); ++i)

   {

      Position next = {path.top().x + dir[i].x, path.top().y + dir[i].y};

      if(next.x >= 0 && next.x < cols

         && next.y >=0 && next.y < rows

         && matrix[next.y * cols + next.x] == str[path.size()]

         && !visited[next.y * cols + next.x])

      {

         path.push(next);

         visited[next.y * cols + next.x] = true;


         return true;

      }

   }


   return false;

}
```

When we found that the *k*th character should not be on a path, we call the functionpopAndGetNext to pop it off, and try on other directions from the (*k*-1)th character. This function is implemented as below:

```
bool popAndGetNext(char* matrix, int rows, int cols, char* str, std::stack<Position>& path, bool* visited)
{
    Position toBePoped = path.top();
    path.pop();
    visited[toBePoped.y * cols + toBePoped.x] = false;

    bool hasNext = false;
    if(path.size() >= 1)
    {
        Position previous = path.top();
        int deltaX = toBePoped.x - previous.x;
        int deltaY = toBePoped.y - previous.y;
        for(int i = 0; (i < sizeof(dir) / sizeof(Position) && !hasNext); ++i)
        {
            if(deltaX != dir[i].x || deltaY != dir[i].y)
                continue;

            hasNext = getNext(matrix, rows, cols, str, path, visited, i + 1);
        }
    }
    return hasNext;
}
```

23. Group of 1s in a Matrix

Problem: Given a matrix with 1s and 0s, please find the number of groups of 1s. A group is defined by horizontally or vertically adjacent 1s. For example, there are four groups of 1s in Figure 1 which are drawn with different colors.

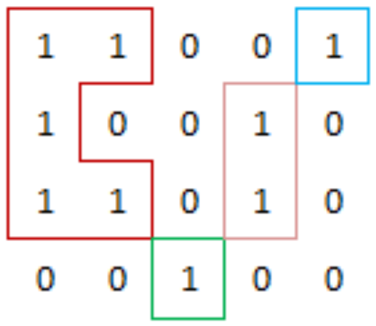


Figure 1: A matrix with four groups of 1s. Different groups are drawn with different colors.

Analysis: All numbers in the matrix are scanned one by one. When a 1 is met, a group of 1s has been found, and then all 1s in the group are flipped to 0s.

For example, the first entry in the matrix of Figure 1 contains 1, so the first group of 1s is found when we access to the first entry. After all 1s in the group have been flipped to 0s, the matrix becomes the one in Figure 2.

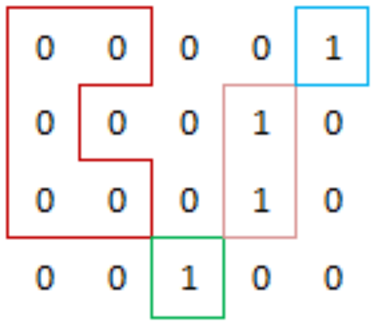


Figure 2: The new matrix after all 1s in the first group (in the red boundary) have been flipped

Therefore, the overall function to count the groups of 1s in a matrix can be implemented in the following C++ code:

```
int groupOf1(int* nums, int rows, int cols)
{
   if(nums == NULL || rows <= 0 || cols <= 0)
      return 0;

   int group = 0;
   for(int i = 0; i < rows * cols; ++i)
   {
      if(nums[i] == 1)
      {
         group++;
         flipAdjacent1(nums, rows, cols, i);
      }
   }

   return group;
}
```

Let's move on to flip 1s inside a group. Actually, it is an application of the seed filling algorithm. When we are going to flip a group starting from an entry, we take the entry as a seed and push it into a stack. At each step, we get an entry from the top of the stack, flip it, and then push its neighboring entries into the stack. The process continues until the stack is empty. The following code implements this algorithm:

```
void flipAdjacent1(int* nums, int rows, int cols, int index)
{
   stack<int> group;
   group.push(index);
   while(!group.empty())
   {
      int topIndex = group.top();
      group.pop();
      nums[topIndex] = 0;

      int row = topIndex / cols;
      int col = topIndex % cols;

      // up
```

```
        if(row > 0 && nums[(row - 1) * cols + col] == 1)
            group.push((row - 1) * cols + col);
        // right
        if(col < cols - 1 && nums[row * cols + col + 1] == 1)
            group.push(row * cols + col + 1);
        // down
        if(row < rows - 1 && nums[(row + 1) * cols + col] == 1)
            group.push((row + 1) * cols + col);
        // left
        if(col > 0 && nums[row * cols + col - 1] == 1)
            group.push(row * cols + col - 1);
    }
}
```

24. Group of 1s in a Matrix

Problem: Given a matrix with 1s and 0s, please find the number of groups of 1s. A group is defined by horizontally or vertically adjacent 1s. For example, there are four groups of 1s in Figure 1 which are drawn with different colors.

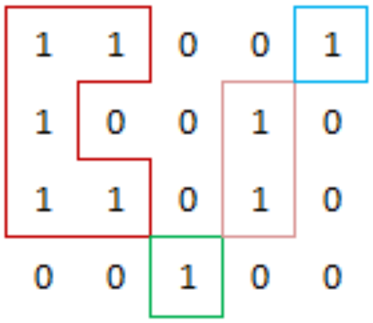


Figure 1: A matrix with four groups of 1s. Different groups are drawn with different colors.

Analysis: All numbers in the matrix are scanned one by one. When a 1 is met, a group of 1s has been found, and then all 1s in the group are flipped to 0s.

For example, the first entry in the matrix of Figure 1 contains 1, so the first group of 1s is found when we access to the first entry. After all 1s in the group have been flipped to 0s, the matrix becomes the one in Figure 2.

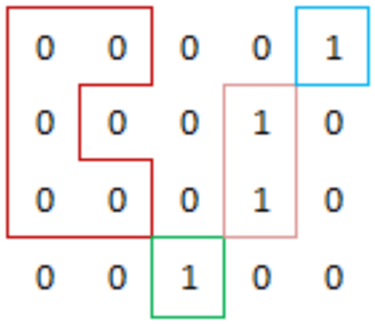


Figure 2: The new matrix after all 1s in the first group (in the red boundary) have been flipped

Therefore, the overall function to count the groups of 1s in a matrix can be implemented in the following C++ code:

```
int groupOf1(int* nums, int rows, int cols)
{
   if(nums == NULL || rows <= 0 || cols <= 0)
      return 0;

   int group = 0;
   for(int i = 0; i < rows * cols; ++i)
   {
      if(nums[i] == 1)
      {
         group++;
         flipAdjacent1(nums, rows, cols, i);
      }
   }

   return group;
}
```

Let's move on to flip 1s inside a group. Actually, it is an application of the seed filling algorithm. When we are going to flip a group starting from an entry, we take the entry as a seed and push it into a stack. At each step, we get an entry from the top of the stack, flip it, and then push its neighboring entries into the stack. The process continues until the stack is empty. The following code implements this algorithm:

```
void flipAdjacent1(int* nums, int rows, int cols, int index)
{
   stack<int> group;
```

```
    group.push(index);
    while(!group.empty())
    {
        int topIndex = group.top();
        group.pop();
        nums[topIndex] = 0;

        int row = topIndex / cols;
        int col = topIndex % cols;

        // up
        if(row > 0 && nums[(row - 1) * cols + col] == 1)
            group.push((row - 1) * cols + col);
        // right
        if(col < cols - 1 && nums[row * cols + col + 1] == 1)
            group.push(row * cols + col + 1);
        // down
        if(row < rows - 1 && nums[(row + 1) * cols + col] == 1)
            group.push((row + 1) * cols + col);
        // left
        if(col > 0 && nums[row * cols + col - 1] == 1)
            group.push(row * cols + col - 1);
    }
}
```

25. Maximal Value of Gifts

Question: A board has *n***m* cells, and there is a gift with some value (value is greater than 0) in every cell. You can get gifts starting from the top-left cell, and move right or down in each step, and finally reach the cell at the bottom-right cell. What's the maximal value of gifts you can get from the board?

1	10	3	8
12	2	9	6
5	7	4	11
3	7	16	5

For example, the maximal value of gift from the board above is 53, and the path is highlighted in red.

Analysis: It is a typical problem about dynamic programming. Firstly let's analyze it with recursion. A function *f*(*i*, *j*) is defined for the maximal value of gifts when reaching the cell (*i*, *j*). There are two possible cells before the cell (*i*, *j*) is reached: One is (*i* - 1, *j*), and the other is the cell (*i*, *j*-1). Therefore, *f*(*i*, *j*)= *max*(*f*(*i*-1, *j*), *f*(*i*, *j*-1)) + gift[*i*, *j*].

Even though it's a recursive equation, it's not a good idea to write code in recursion, because there might be many over-lapping sub-problems. A better solution is to solve is with iteration. A 2-D matrix is utilized, and the value in each cell (*i*, *j*) is the maximal value of gift when reaching the cell (*i*, *j*) on the board.

The iterative solution can be implemented in the following Java code:

```
public static int getMaxValue(int[][] values) {
    int rows = values.length;
    int cols = values[0].length;
    int[][] maxValues = new int[rows][cols];

    for(int i = 0; i < rows; ++i) {
        for(int j = 0; j < cols; ++j) {
            int left = 0;
            int up = 0;
            if(i > 0) {
                up = maxValues[i - 1][j];
            }

            if(j > 0) {
                left = maxValues[i][j - 1];
            }

            maxValues[i][j] = Math.max(left, up) + values[i][j];
        }
    }

    return maxValues[rows - 1][cols - 1];
}
```

Optimization

The maximal value of gifts when reaching the cell (*i*, *j*) depends on the cells (*i*-1, *j*) and (*i*, *j*-1) only, so it is not necessary to save the value of the cells in the rows *i*-2 and above. Therefore, we can replace the 2-D matrix with an array, as the following code shows:

```
public static int getMaxValue(int[][] values) {

    int rows = values.length;

    int cols = values[0].length;


    int[] maxValues = new int[cols];

    for (int i = 0; i < rows; ++i) {

        for (int j = 0; j < cols; ++j) {

            int left = 0;

            int up = 0;


            if (i > 0) {
                up = maxValues[j];
            }

            if (j > 0) {
                left = maxValues[j - 1];
            }

            maxValues[j] = Math.max(left, up) + values[i][j];
        }
    }

    return maxValues[cols - 1];
}
```

26. Stack with Function min()

Problem: Define a stack, in which we can get its minimum number with a function min. In this stack, the time complexity of min(), push() and pop() are all O(1).

Analysis: Our intuition for this problem might be that we sort all of numbers in the stack when we push a new one, and keep the minimum number on the top of stack. In this way we can get the minimum number in O(1) time. However, we cannot assure that the last number pushed in to container will be the first one to be popped out, so it is no longer a stack.

We may add a new member variable in a stack to keep the minimum number. When we push a new number which is less than the minimum number, we will update it. It sounds good. However, how to get the next minimum number when the current minimum one is popped? Fortunately, we have two solutions for this problem.

Solution 1: With Auxiliary Stack

It is not enough to have only a member variable to keep the minimum number. When the minimum one is popped, we need to get the next minimum one. Therefore, we need to store the next minimum number before push the current minimum one.

How about to store each minimum number (the less value of current minimum number and the number to be pushed) into an auxiliary stack? We may analyze the process to push and pop numbers via some examples (Table 1).

Step	Operation	Data Stack	Auxiliary Stack	Minimum
1	Push 3	3	3	3
2	Push 4	3, 4	3, 3	3
3	Push 2	3, 4, 2	3, 3, 2	2
4	Push 1	3, 4, 2, 1	3, 3, 2, 1	1
5	Pop	3, 4, 2	3, 3, 2	2
6	Pop	3, 4	3, 3	3
7	Push 0	3, 4, 0	3, 3, 0	0

Table 1: The status of data stack, auxiliary stack, minimum value when we push 3, 4, 2, 1, pop twice, and then push 0

At first we push 3 into both data stack and auxiliary stack. Secondly we push 4 into the data stack. We push 3 again into the auxiliary stack because 4 is greater than 3. Thirdly, we continue pushing 2 into the data stack. We update the minimum number as 2 and push it into the auxiliary stack since 2 is less the previous minimum number 3. It is same in the fourth step when we push 1. We also need to update the minimum number and push 1 into the auxiliary stack. We can notice that the top of auxiliary stack is always the minimum number if we push the minimum number into auxiliary stack in each step.

Whenever we pop a number from data stack, we also pop a number from auxiliary stack. If the minimum number is popped, the next minimum number should be also on the top of auxiliary stack. In the fifth step we pop 1 from the data stack, and we also pop the number on the top of auxiliary (which is 1). We can see that the next minimum number 2 is now on the top of auxiliary stack. If we continue popping from both the data and auxiliary stacks, there are only two numbers 3 and 4 left in the data stack. The minimum number 3 is indeed on the top of the auxiliary stack. Therefore, it demonstrates that our solution is correct.

Now we can develop the required stack. The stack is declared as the following:

```
template <typename T> class StackWithMin
{
public:
   StackWithMin(void) {}
   virtual ~StackWithMin(void) {}

   T& top(void);

   void push(const T& value);
   void pop(void);

   const T& min(void) const;

private:
   std::stack<T>   m_data;     // data stack, to store numbers
   std::stack<T>   m_min;      // auxiliary stack, to store minimum numbers
};
```

The function push, pop and min and top can be implemented as:

```
template <typename T> void StackWithMin<T>::push(const T& value)
{
    // push the new number into data stack
    m_data.push(value);

    // push the new number into auxiliary stack
    // if it is less than the previous minimum number,
    // otherwise push a replication of the minimum number
    if(m_min.size() == 0 || value < m_min.top())
        m_min.push(value);
    else
        m_min.push(m_min.top());
}

template <typename T> void StackWithMin<T>::pop()
{
    assert(m_data.size() > 0 && m_min.size() > 0);

    m_data.pop();
    m_min.pop();
}
```

```
template <typename T> const T& StackWithMin<T>::min() const
{
    assert(m_data.size() > 0 && m_min.size() > 0);

    return m_min.top();
}

template <typename T> T& StackWithMin<T>::top()
{
    return m_data.top();
}
```

The length of auxiliary stack should be *n* if we push *n* numbers into data stack. Therefore, we need O(*n*) auxiliary memory for this solution.

Solution 2: Without Auxiliary Stack

The second solution is trickier without an auxiliary stack. We do not always push numbers into data stack directly, but we have some tricky calculation before pushing.

Supposing that we are going to push a number *value* into a stack with minimum number *min*. If*value* is greater than or equal to the *min*, it is pushed directly into data stack. If it is less than *min*, we push 2**value* -*min*, and update *min* as *value* since a new minimum number is pushed. How about to pop? We pop it directly if the top of data stack (it is denoted as *top*) is greater than or equal to *min*. Otherwise the number *top* is not the real pushed number. The real pushed number is stored is *min*. After the current minimum number is popped, we need to restore the previous minimum number, which is 2**min*-*top*.

Now let us demonstrate its correctness of this solution. Since *value* is greater than or equal to*min*, it is pushed into data stack direct without updating *min*. Therefore, when we find that the top of data stack is greater than or equal to *min*, we can pop directly without updating *min*. However, if we find *value* is less then *min*, we push 2**value*-*min*. We should notice that 2**value*-*min* should be less than *value*. Then we update current *min* as *value*. Therefore, the new top of data stack (*top*) is less than the current *min*. Therefore, when we find that the top of data stack is less then*min*, the real top (real pushed number *value*) is stored in *min*. After we pop the top of data stack, we have to restore the previous minimum number. Since *top* = 2**value*-previous *min* and *value* is current *min*, pervious *min* is 2*current *min* - *top*.

It sounds great. We feel confident to write code now with the correctness demonstration. The following is the sample code:

```
template <typename T> class StackWithMin
{
public:
   StackWithMin(void) {}
   virtual ~StackWithMin(void) {}

   T& top(void);

   void push(const T& value);
   void pop(void);

   const T& min(void) const;

private:
   std::stack<T>   m_data;     // data stack, to store numbers
```

```
    T               m_min;      // minimum number
};

template <typename T> void StackWithMin<T>::push(const T& value)
{
    if(m_data.size() == 0)
    {
        m_data.push(value);
        m_min = value;
    }
    else if(value >= m_min)
    {
        m_data.push(value);
    }
    else
    {
        m_data.push(2 * value - m_min);
        m_min = value;
    }
}

template <typename T> void StackWithMin<T>::pop()
{
    assert(m_data.size() > 0);
    if(m_data.top() < m_min)
        m_min = 2 * m_min - m_data.top();

    m_data.pop();
}

template <typename T> const T& StackWithMin<T>::min() const
{
    assert(m_data.size() > 0);

    return m_min;
}

template <typename T> T& StackWithMin<T>::top()
{
    T top = m_data.top();
    if(top < m_min)
        top = m_min;

    return top;
}
```

In this solution, we don’t need the O(*n*) auxiliary stack, so it is more efficient in the perspective of memory utilization than the first solution above.

27. Queue Implemented with Two Stacks

Problem: Implement a queue with two stacks. The class for queues is declared in C++ as below. Please implement two functions: appendTail to append an element into tail of a queue, anddeleteHead to delete an element from head of a queue.

```
template <typename T> class CQueue

{

public:

   CQueue(void);

   ~CQueue(void);


     void appendTail(const T& node);

   T deleteHead();


private:

   stack<T> stack1;

   stack<T> stack2;

};
```

Analysis: According to declaration above, a queue contains two stacks stack1 and stack2. Therefore, it is required to implement a queue which follows the rule “First In First Out” with two stacks which follow the rule of “First In Last Out”.

We analyze the process to add and delete some elements via some examples. Firstly en element a is inserted. Let us push it into stack1. There is an element {a} in stack1and stack2 is empty. We continue to add two more elements b and c (push them into stack1 too). There are three elements {a, b, c} in stack1 now, where c is on its top, and stack2 is still empty (as shown in Figure 1-a).

We then have a try to delete an element from a queue. According to the rule “First in First out”, the first element to be deleted is a since it is added before b and c. The element a is stored in tostack1, and it is not on the top of stack. Therefore, we cannot pop it directly. We can notice thatstack2 has not been used, so it is the time for us to utilize it. If we pop elements from stack1and push them into stack2 one by one, the order of elements in stack2 is reverse to the order in stack1. After three popping and pushing operations, stack1 becomes empty and there are three elements {c, b, a} in stack2. The element a can be popped out now since it is on the top ofstack2. Now there are two elements left {c, b} in stack2 and b is on its top (as shown in Figure 1-b).

How about to continue deleting more elements from the tail of queue? The element b is inserted into queue before c, so it should be deleted when there are two elements b and c left in queue. It can be popped out since it is on the top of stack2. After the popping operation, stack1 remains empty and there is only an element c in stack2 (as shown in Figure 1-c).

It is time to summarize the steps to delete an element from a queue: The top of stack2 can be popped out since it is the first element inserted into queue when stack2 is not empty. Whenstack2 is empty, we pop all elements from stack1 and push them into stack2 one by one. The first element in a queue is pushed into the bottom of stack1. It can be popped out directly after popping and pushing operations since it is on the top of stack2.

Let us insert another element d. How about to push it into stack1 (as shown in Figure1-d)? When we continue to delete the top of stack2, which is element c, can be popped because it is not empty (as shown in Figure 1-d). The element c is indeed inserted into queue before the element d, so it is a reasonable operation to delete c before d. The final status of the queue is shown as Figure 1-e.

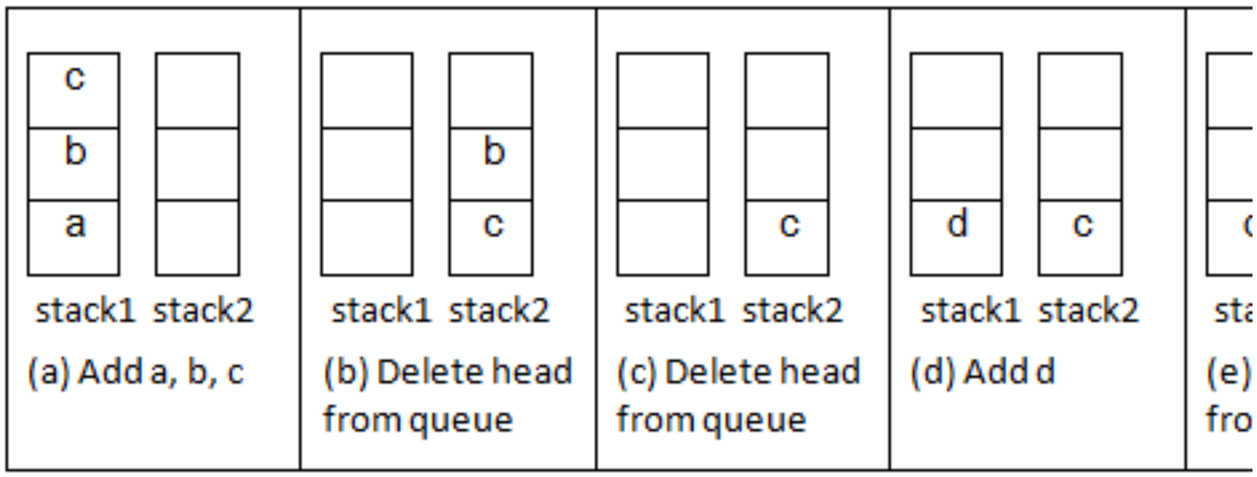


Figure 1: The process to simulate a queue with two stacks.

We can write code after we get clear ideas about the process to insert and delete elements. Some sample code is shown below:

```
template<typename T> void CQueue<T>::appendTail(const T& element)
{
    stack1.push(element);
}

template<typename T> T CQueue<T>::deleteHead()
{
    if(stack2.size()<= 0)
    {
        while(stack1.size()>0)
        {
            T& data = stack1.top();
            stack1.pop();

            stack2.push(data);

        }

    }


    if(stack2.size() == 0)

        throw new exception("queue is empty");


    T head = stack2.top();

    stack2.pop();


    return head;

}
```

28. Push and Pop Sequences of Stacks

Problem: Given two integer sequences, one of which is the push sequence of a stack, please check whether the other sequence is a corresponding pop sequence or not.

For example, if 1, 2, 3, 4, 5 is a push sequence, 4, 5, 3, 2, 1 is a corresponding pop sequence, but the sequence 4, 3, 5, 1, 2 is not.

Analysis: An intuitive thought on this problem is to create an auxiliary stack. We push the numbers in the first sequence one by one, and try to pop them out according to the order in the second sequence.

Take the sequence 4, 5, 3, 2, 1 as an example to analyze the process to push and pop. The first number to be popped is 4, so we need to push it into a stack. The pushing order is defined in the first sequence, where there are numbers 1, 2 and 3 prior to 4. Therefore, numbers 1, 2, and 3 are pushed into a stack before 4 is pushed. At this time, there are 4 numbers in a stack, which are 1, 2, 3 and 4, with 4 on top. When 4 is popped, numbers 1, 2 and 3 are left. The next number to be popped is 5, which is not on top of stack, so we have to push numbers in the first sequence into stack until 5 is pushed. When number 5 is on top of a stack, we can pop it. The next three numbers to be popped are 3, 2 and 1. Since these numbers are on top of a stack before pop operations, they can be popped directly. The whole process to push and pop is summarized in Table 1.

Step	Operation	Stack Status	Popped	Step	Operation	Stack Status	Popped
1	Push 1	1		**6**	Push 5	1, 2, 3, 5	
2	Push 2	1, 2		**7**	Pop	1, 2, 3	5
3	Push 3	1, 2, 3		**8**	Pop	1, 2	3
4	Push 4	1, 2, 3, 4		**9**	Pop	1	2
5	Pop	1, 2, 3	4	**10**	Pop		1

Table 1: The process to push and pop with a push sequence 1, 2, 3, 4, 5 and pop sequence 4, 5, 3, 2, 1

Let us continue to analyze another pop sequence 4, 3, 5, 1, 2. The process to pop the first number 4 is similar to the process above. After the number 4 is popped, 3 is on the top of stack and it can be popped. The next number to be popped is 5. Since it is not on top, we have to push numbers in the first sequence until the number 5 is pushed. The number 5 can be popped when it is pushed onto the top of a stack. After 5 is popped out, there are only two numbers 1 and 2 left in stack. The next number to be popped is 1, but it is not on the top of stack. We have to push numbers in the first sequence until 1 is pushed. However, all numbers in the first sequence have been pushed. Therefore, the sequence 4, 3, 5, 1, 2 is not a pop sequence of the stack with push sequence 1, 2, 3, 4, 5. The whole process to push and pop is summarized in Table 2.

Step	Operation	Stack Status	Popped	Step	Operation	Stack Status	Popped
1	Push 1	1		**6**	Pop	1, 2	3
2	Push 2	1, 2		**7**	Push 5	1, 2, 5	

3	Push 3	1, 2, 3		**8**	Pop	1, 2	5
4	Push 4	1, 2, 3, 4		The next number to be popped is 1, which is neither on the top of stack, nor in the remaining numbers of push sequence.			
5	Pop	1, 2, 3	4				

Table 1: The process to push and pop with a push sequence 1, 2, 3, 4, 5 and pop sequence 4, 3, 5, 1, 2

According to the analysis above, we get a solution to check whether a sequence is a pop sequence of a stack or not. If the number to be popped is currently on top of stack, just pop it. If it is not on the top of stack, we have to push remaining numbers into the auxiliary stack until we meet the number to be popped. If the next number to be popped is not remaining in the push sequence, it is not a pop sequence of a stack. The following is some sample code based on this solution:

```
bool IsPopOrder(const int* pPush, const int* pPop, int nLength)
{
   bool bPossible = false;

   if(pPush != NULL && pPop != NULL && nLength > 0)
   {
      const int* pNextPush = pPush;
      const int* pNextPop = pPop;

      std::stack<int> stackData;

      while(pNextPop - pPop < nLength)
      {
         // When the number to be popped is not on top of stack,
         // push some numbers in the push sequence into stack
         while(stackData.empty() || stackData.top() != *pNextPop)
         {
             // If all numbers have been pushed, break
             if(pNextPush - pPush == nLength)
                 break;

             stackData.push(*pNextPush);

             pNextPush ++;
         }

         if(stackData.top() != *pNextPop)
             break;

         stackData.pop();
         pNextPop ++;
      }

      if(stackData.empty() && pNextPop - pPop == nLength)
         bPossible = true;
   }

   return bPossible;
}
```

29. Maximums in Sliding Windows

Question: Given an array of numbers and a sliding window size, how to get the maximal numbers in all sliding windows?

For example, if the input array is {2, 3, 4, 2, 6, 2, 5, 1} and the size of sliding windows is 3, the output of maximums are {4, 4, 6, 6, 6, 5}, as illustrated in Table1.

Sliding Windows in an Array	**Maximums in Sliding Windows**
[2, 3, 4], 2, 6, 2, 5, 1	4
2, **[3, 4, 2]**, 6, 2, 5, 1	4
2, 3, **[4, 2, 6]**, 2, 5, 1	6
2, 3, 4, **[2, 6, 2]**, 5, 1	6
2, 3, 4, 2, **[6, 2, 5]**, 1	6
2, 3, 4, 2, 6, **[2, 5, 1]**	5

Table 1: Maximums of all sliding windows with size 3 in an array {2, 3, 4, 2, 6, 2, 5, 1}. A pair of brackets indicates a sliding window.

Analysis: It is not difficult to get a solution with brute force: Scan numbers in every sliding window to get its maximal value. The overall time complexity is O(*nk*) if the length of array is *n*and the size of sliding windows is *k*.

The naïve solution is not the best solution. Let us explore better solutions.

Solution 1: Maximal value in a queue

A window can be viewed as a queue. When it slides, a number is pushed into its back, and its front is popped off. Therefore, the problem is solved if we can get the maximal value of a queue.

There are no straightforward approaches to getting the maximal value of a queue. However, there are solutions to get the maximal value of a stack, which is similar to the solution introduced in the blog "Stack with Function min()". Additionally, a queue can also be implemented with two stacks (details are discussed in another blog "Queue implemented with Two Stacks").

If a new type of queue is implemented with two stacks, in which a function max() is defined to get the maximal value, the maximal value in a queue is the greater number of the two maximal numbers in two stacks.

This solution is workable. However, we may not have enough time to write all code to implement our own queue and stack data structures during interviews. Let us continue exploring a more concise solution.

Solution 2: Saving the maximal value into the front of a queue

Instead of pushing every numbers inside a sliding window into a queue, we try to push the candidates of maximum only into a queue. Let us take the array {2, 3, 4, 2, 6, 2, 5, 1} as an example to analyze the solution step by step.

The first number in the array is 2, we push it into a queue. The second number is 3, which is greater than the previous number 2. The number 2 should be popped off, because it is less than 3 and it has no chance to be the maximal value. There is only one number left in the queue when we pop 2 at the back and push 3 at the back. The operations are similar when we push the next number 4. There is only a number 4 remaining in the queue. Now the sliding window already has three elements, we can get the maximum value at the front of the queue.

We continue to push the fourth number. It is pushed at the back of queue, because it is less than the previous number 4 and it might be a maximal number in the future when the previous numbers are popped off. There are two numbers, 4 and 2, in the queue, and 4 is the maximum.

The next number to be pushed is 6. Since it is greater than the existing numbers, 4 and 2, these two numbers can be popped off because they have no chance to be the maximum. Now there is only one number in the queue, which is 6, after the current number is pushed. Of course, the maximum is 6.

The next number is 2, which is pushed into the back of the queue because it is less than the previous number 6. There are two numbers in the queue, 6 and 2, and the number 6 at the front of the queue is the maximal value.

It is time to push the number 5. Because it is greater than the number 2 at the back of the queue, 2 is popped off and then 5 is pushed. There are two numbers in the queue, 6 and 5, and the number 6 at the front of the queue is the maximal value.

Now let us push the last number 1. It can be pushed into the queue. It is noticeable that the number at the front is beyond the scope the current sliding window, and it should be popped off. How do we know whether the number at the front of the queue is out of sliding window? Rather than storing numbers in the queue directly, we can store indices instead. If the distance between the index at the front of queue and the index of the current number to be pushed is greater than or equal to the window size, the number corresponding to be the index at the font of queue is out of sliding window.

The analysis process above is summarized in Table 2.

Step	**Number to Be Pushed**	**Numbers in Sliding Window**	**Indices in queue**	**Maximum in Window**
1	2	2	0(2)	

2	3	2, 3	1(3)	
3	4	2, 3, 4	2(4)	4
4	2	3, 4, 2	2(4), 3(2)	4
5	6	4, 2, 6	4(6)	6
6	2	2, 6, 2	4(6), 5(2)	6
7	5	6, 2, 5	4(6), 6(5)	6
8	1	2, 5, 1	6(5), 7(1)	5

Table 2: The process to get the maximal number in all sliding windows with window size 3 in the array {2, 3, 4, 2, 6, 2, 5, 1}. In the column "Indices in queue", the number inside a pair of parentheses is the number indexed by the number before it in the array.

We can implement a solution based on the analysis above. Some sample code in C++ is shown below, which utilizes the type deque of STL.

```
vector<int> maxInWindows(const vector<int>& numbers, int windowSize)
{
    vector<int> maxInSlidingWindows;
    if(numbers.size() >= windowSize && windowSize > 1)
    {
        deque<int> indices;

        for(int i = 0; i < windowSize; ++i)
        {
            while(!indices.empty() && numbers[i] >= numbers[indices.back()])
                indices.pop_back();
        indices.push_back(i);
      }

      for(int i = windowSize; i < numbers.size(); ++i)
      {
        maxInSlidingWindows.push_back(numbers[indices.front()]);

        while(!indices.empty() && numbers[i] >= numbers[indices.back()])
          indices.pop_back();
        if(!indices.empty() && indices.front() <= i - windowSize)
          indices.pop_front();

        indices.push_back(i);
      }
      maxInSlidingWindows.push_back(numbers[indices.front()]);
    }

    return maxInSlidingWindows;
}
```

Extension: Another solution to get the maximum of a queue

As we mentioned before, a sliding window can be viewed as a queue. Therefore, we can implement a new solution to get the maximal value of a queue based on the second solution to get the maximums of sliding windows.

The following is the sample code:

```
template<typename T> class QueueWithMax
{
public:
    QueueWithMax(): currentIndex(0)
    {
    }

    void push_back(T number)
    {
        while(!maximums.empty() && number >= maximums.back().number)
            maximums.pop_back();

        InternalData internalData = {number, currentIndex};
        data.push_back(internalData);
        maximums.push_back(internalData);

        ++currentIndex;
    }

    void pop_front()
    {
        if(maximums.empty())
```

```
            throw new exception("queue is empty");

        if(maximums.front().index == data.front().index)
            maximums.pop_front();

        data.pop_front();
    }

    T max() const
    {
        if(maximums.empty())
            throw new exception("queue is empty");

        return maximums.front().number;
    }

private:
    struct InternalData
    {
        T number;
        int index;
    };

    deque<InternalData> data;
    deque<InternalData> maximums;
    int currentIndex;
};
```

30. Stacks Sharing an Array

Problem 1: How can you implement two stacks in a single array, where no stack overflows until no space left in the entire array space?

Analysis: An array has two ends, so each of the two stacks may grow from an end in the array. Figure 1 below shows the initial status of the array and two stacks (assuming the capacity of the array is 10).

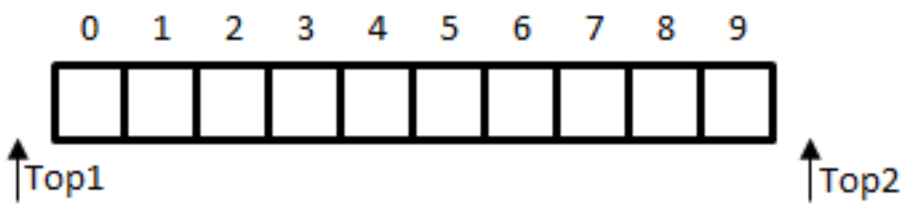


Figure 1: Initial status of the array and two stacks

Since two stacks are empty at first, so indexes of top items are initialized as -1 and 10 at first. When items are pushed into first stack, it grows from left to right. Similarly, the second stack grows from right to left when items are items are pushed into it. For example, Figure 2 shows the status when three items *a*, *b* and *c* are pushed into the first stack, and *d*, *e* are pushed into the second stack.

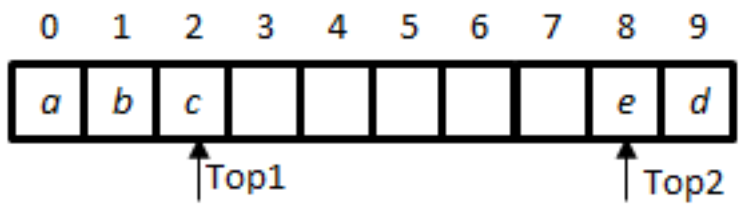


Figure 2: The status after three items are pushed into the first stack and two items are pushed into the second stack

No more items can be pushed into stacks when two top items are adjacent to each other, because all space in the array has been occupied.

Our solution can be implemented with the following C++ class:

```
template <typename T, int capacity> class TwoStacks
{
public:
    TwoStacks()
    {
        topFirst = -1;
        topSecond = capacity;
    }
    T top(int stackIndex)
    {
        validateIndex(stackIndex);
        if(empty(stackIndex))
            throw new exception("The stack is empty.");
        if(stackIndex == 0)
            return items[topFirst];
        return items[topSecond];
    }
    void push(int stackIndex, T item)
    {
        validateIndex(stackIndex);
        if(full())
            throw new exception("All space has been occupied.");
        if(stackIndex == 0)
```

```
            items[++topFirst] = item;
        else
            items[--topSecond] = item;
    }
    void pop(int stackIndex)
    {
        validateIndex(stackIndex);
        if(empty(stackIndex))
            throw new exception("The stack is empty.");
        if(stackIndex == 0)
            --topFirst;
        else
            ++topSecond;
    }
    bool empty(int stackIndex)
    {
        if(stackIndex == 0 && topFirst == -1)
            return true;
        if(stackIndex == 1 && topSecond == capacity)
            return true;
        return false;
    }
private:
    bool full()
    {
        return (topFirst >= topSecond - 1);
    }
    void validateIndex(int stackIndex)
    {
        if(stackIndex < 0 || stackIndex > 1)
            throw new exception("Invalid Stack Index.");
    }
private:
    T items[capacity];
    int topFirst;
    int topSecond;
};
```

Problem 2: How can you implement *n* (*n* > 2) stacks in a single array, where no stack overflows until no space left in the entire array space?

Analysis: An array only has two ends, so we need new strategies to implement more than two stacks in a single array.

We may implement the array like a list, where each item in the array has a link to another item. When an item has not been occupied by any stacks and it is available, it is linked to the next available item. When an item is pushed into a stack, it is linked to the previous item in the stack. That's to say, there are *n* + 1 list in total in the system if there are *n* stacks sharing an array, a list for the available space left in the array, and a list for each stack.

Let's take three stacks sharing an array with capacity 10 as an example. Figure 3 shows the initial status for the array and stacks. Each item in the array has two blocks, one for item data and the other for the index of another block.

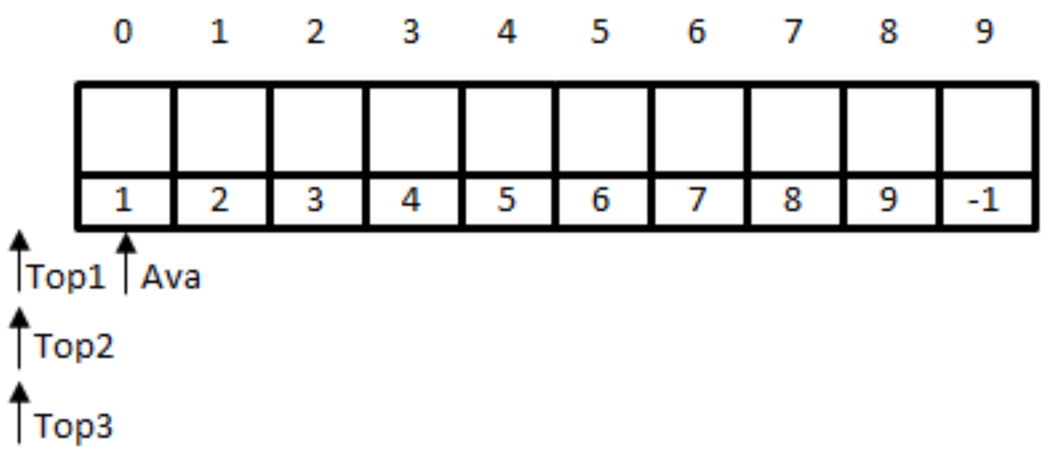


Figure 3: Initial status of three stacks sharing an array

As shown in Figure 3, each item is linked to the next item (the link block of the *i*th item is *i*+1). The head of list for available items in the array points to the item at position 0, which is the Ava pointer in the figure. Since three stacks are empty, their top (Top1, Top2 and Top3 in the figure) are initialized as -1.

Let's try to push an item *a* into the first stack. Currently the first available item is at the position 0, so we set its data block as *a*. The link block of the item is 1, which means the next available item is at the position 1, so we update the Ava pointer to the position 1. Additionally, the link block of the item at the position 1 should be updated to -1, the previous top index of the first stack. Lastly, we update to top index of the first stack to 0. The status of the array and stacks are shown in Figure 4.

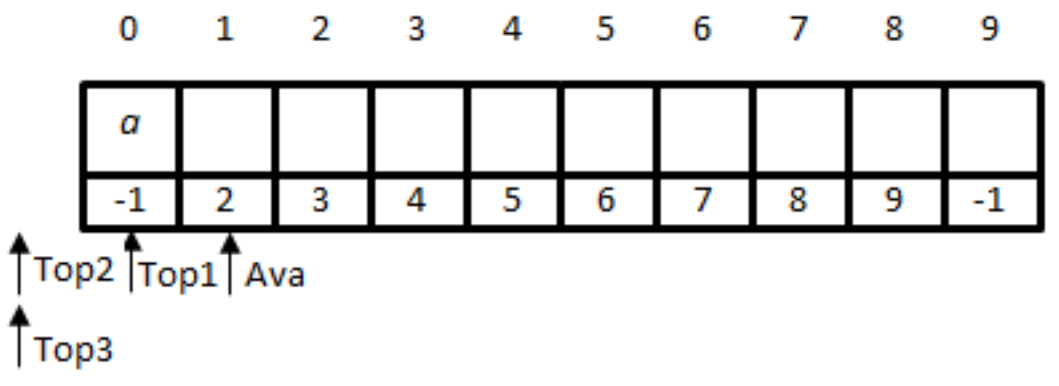


Figure 4: The status after pushing a into the first stack

Let's push two more items *b* and *c* into the first stack. The operations are similar as before, and the status is shown in Figure 5.

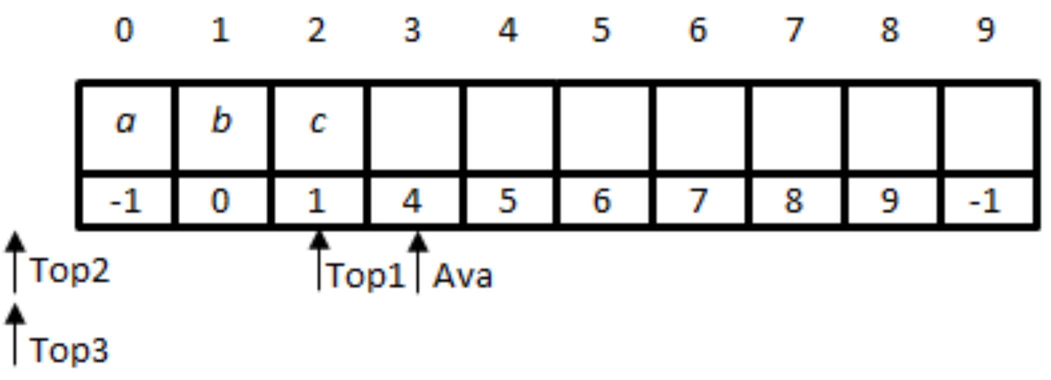


Figure 5: The status after adding two more items, b and c, into the first stack

In the next step we are going to push another *d* into the second stack. Most operations are similar as before. The link block in the item at the position 3 is updated to -1, since the second stack is empty before and its top index is -1 previously. And then the top index of the second stack is updated to 3. The status after adding *d* into the second stack is shown in Figure 6.

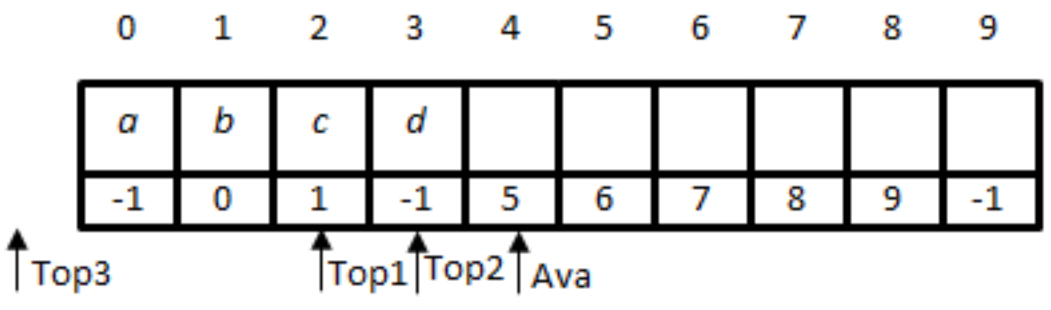


Figure 6: The status after pushing d into the second stack

If we continue to push three more item *e*, *f*, and *g* into the second stack, the status is shown in Figure 7.

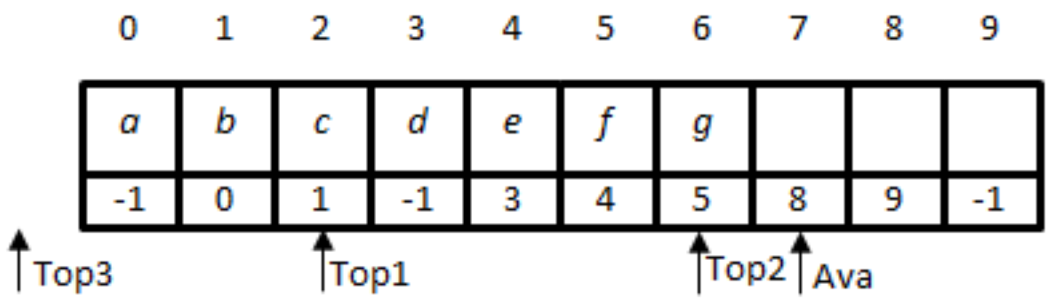


Figure 7: The status after pushing three more items, e, f, and g, into the second stack

At this time we are going to pop an item from the first stack. Since the top index of the first stack is 2, the item to be popped off is at the position 2. The link value in that item is 1, which means the previous top in the first stack is at the position 1 in the array, so we update the top index of the first stack as 1. Now the item at the position 2 becomes available, it should be linked to the list for available items. We move the Ava pointer to 2, and then update the link value of the item at the position 2 as 7, which is the previous head position of the list for available items. The status is shown in Figure 8.

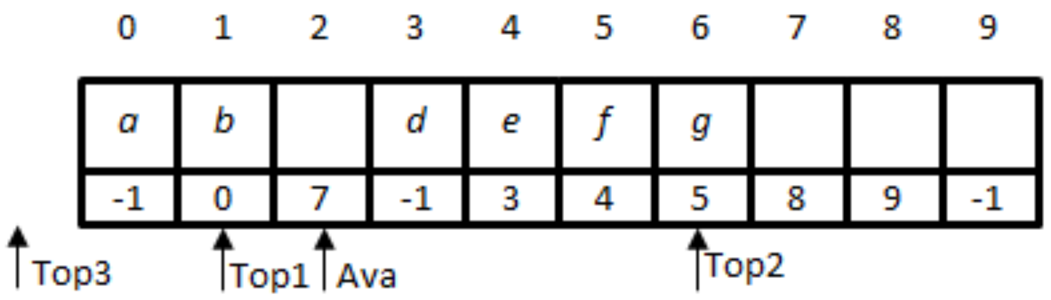


Figure 8: The status after popping an item from the first stack

If we pop an item off the second stack, the item at the position 6 will be linked to the list for available items too. Figure 9 depicts the status after the related operations.

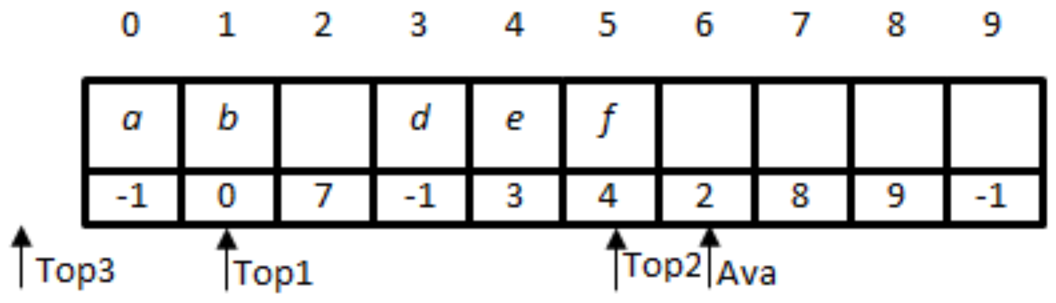


Figure 9: The status after popping an item off the second stack

If we push *h* into the third stack, it will be placed into the item at the position 6 because Ava points to that item. The next available item is at the position 2 (the link value in the item at the position 6). Therefore, the head of the list for available items points to location 2, as shown in Figure 10.

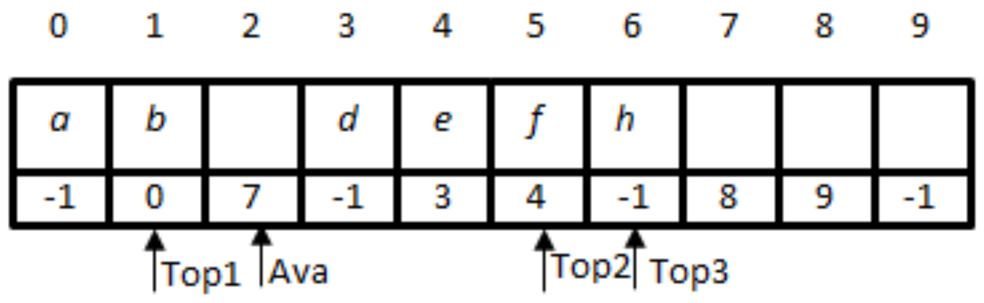


Figure 10: The status after pushing h into the third stack

Let's continue to push four more items into the third stack, *i*, *j*, *k*, and *l*. The status after these four items are pushed is shown in Figure 11. At this time, there are two items in the first stack (*a* and*b*), three items in the second stack (*d*, *e* and *f*), and five items in the third stack (*h*, *i*, *j*, *k*, and *l*). Please note that items inside a stack are not necessarily adjacent.

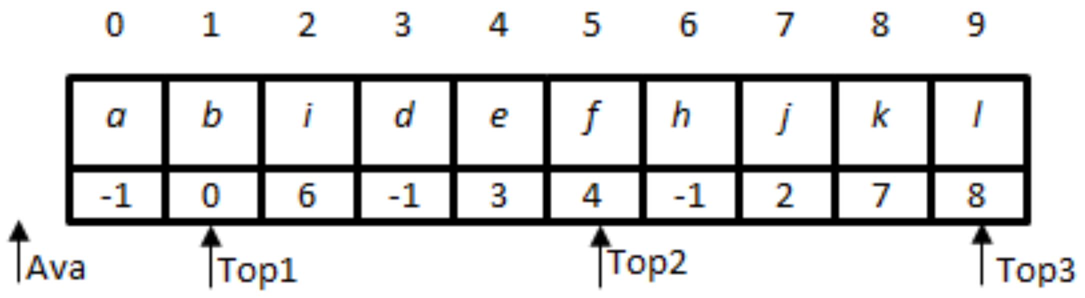


Figure 11: The status after pushing i, j, k, l into the third stack

After *l* is pushed into the item at the position 9, the Ava pointer is update to -1 (the previous link value in the item at the position 9), which means all items in the array have been occupied by stacks. We can't push more items until some items are popped off.

The source code in C++ to implement stacks sharing an array is listed below:

```
template <typename T, int capacity, int count> class Stacks

{

public:

    Stacks()

    {

        int i;

        for(i = 0; i < capacity - 1; ++i)

            items[i].link = i + 1;

        items[i].link = -1;

        emptyHead = 0;


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

            stackHead[i] = -1;

    }


    T top(int stackIndex)

    {

        validateIndex(stackIndex);

        if(empty(stackIndex))

            throw new exception("The stack is empty.");
        return items[stackHead[stackIndex]].data;
    }

    void push(int stackIndex, const T& item)
    {
        validateIndex(stackIndex);
        if(full())
            throw new exception("All space has been occupied.");

        Item<T>& block = items[emptyHead];
        int nextEmpty = block.link;

        block.data = item;
        block.link = stackHead[stackIndex];
        stackHead[stackIndex] = emptyHead;

        emptyHead = nextEmpty;
    }

    void pop(int stackIndex)
    {
        validateIndex(stackIndex);
        if(empty(stackIndex))
            throw new exception("The stack is empty.");
        Item<T>& block = items[stackHead[stackIndex]];

        int nextItem = block.link;


        block.link = emptyHead;
        emptyHead = stackHead[stackIndex];

        stackHead[stackIndex] = nextItem;

    }


    bool empty(int stackIndex)

    {

        return (stackHead[stackIndex] < 0);

    }


private:

    void validateIndex(int stackIndex)

    {

        if(stackIndex < 0 || stackIndex >= count)

            throw new exception("Invalid index of stack.");

    }


    bool full()

    {

        return (emptyHead < 0);

    }
```

```
private:
    template <typename T> struct Item
    {
        T data;
        int link;
    };

private:
    Item<T> items[capacity];
    int emptyHead;
    int stackHead[count];
};
```

31. Copy List with Random Pointer

Problem:

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a deep copy of the list.

1. Some Thoughts

We can solve this problem by doing the following steps:

1. copy every node, i.e., duplicate every node, and insert it to the list
2. copy random pointers for all newly created nodes
3. break the list to two

```
public RandomListNode copyRandomList(RandomListNode head) {
            if (head == null)                   return null;
            RandomListNode p = head;

// copy every node and insert to list
while (p != null) {
            RandomListNode copy = new RandomListNode(p.label);
            copy.next = p.next;
      p.next = copy;
      p = copy.next;
}

// copy random pointer for each new node
      p = head;
      while (p != null) {
            if (p.random != null)
            p.next.random = p.random.next;
      p = p.next.next;
   }

// break list to two
            p = head;
  RandomListNode newHead = head.next;
  while (p != null) {
   RandomListNode temp = p.next;
   p.next = temp.next;
    if (temp.next != null)
                        temp.next = temp.next.next;
            p = p.next;
}


   return newHead; }
```

The break list part above move pointer 2 steps each time, you can also move one at a time which is simpler, like the following:

```
while(p != null && p.next != null){
  RandomListNode temp = p.next;
  p.next = temp.next;
  p = temp;
 }
```

Solution Using HashMap

```
public RandomListNode copyRandomList(RandomListNode head) {
if (head == null)                  return null;

HashMap<RandomListNode, RandomListNode> map = new HashMap<RandomListNode,
RandomListNode>();
RandomListNode newHead = new RandomListNode(head.label);
            RandomListNode p = head;
RandomListNode q = newHead;
map.put(head, newHead);
p = p.next;

while (p != null) {
            RandomListNode temp = new RandomListNode(p.label);
            map.put(p, temp);
      q.next = temp;
      q = temp;
      p = p.next;
}
p = head;
q = newHead;

            while (p != null) {
            if (p.random != null)
            q.random = map.get(p.random);
      else
            q.random = null;
            p = p.next;
            q = q.next;            }

return newHead;
 }
```

32. **Given a sorted linked list, delete all duplicates such that each element appear only once.**

For example,

Given 1->1->2, return 1->2. Given 1->1->2->3->3, return 1->2->3.

Thoughts

The key of this problem is using the right loop condition. And change what is necessary in each loop. You can use different iteration conditions like the following 2 solutions.

Solution 1

```
/**  * Definition for singly-linked list.  *
 public class ListNode {
 *     int val;
*     ListNode next;
  *     ListNode(int x)
{  *         val = x;
*         next = null;  *     }  * }
 */
public class Solution {
   public ListNode deleteDuplicates(ListNode head) {
     if(head == null || head.next == null)
  return head;

   ListNode prev = head;
   ListNode p = head.next;

   while(p != null){
  if(p.val == prev.val){
   prev.next = p.next;
   p = p.next;

  //no change prev              }
else{
   prev = p;
   p = p.next;
    }
 }
   return head;     } }
```

Solution 2

```
public class Solution {
 public ListNode deleteDuplicates(ListNode head) {
    if(head == null || head.next == null)
  return head;

 ListNode p = head;
  while( p!= null && p.next != null){
       if(p.val == p.next.val){
               p.next = p.next.next;
    }else{
        p = p.next;              }
   }
 return head;     }
}
```

33. ***Merge two sorted linked lists and return it as a new list.***

Problem:

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

Key to solve this problem

The key to solve the problem is defining a fake head. Then compare the first elements from each list. Add the smaller one to the merged list. Finally, when one of them is empty, simply append it to the merged list, since it is already sorted.

Java Solution

```
/**
  * Definition for singly-linked list.
  * public class ListNode
 {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;   *
        next = null;
 *     }
 * }
 */
public class Solution {
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
     ListNode p1 = l1;
     ListNode p2 = l2;
     ListNode fakeHead = new ListNode(0);
     ListNode p = fakeHead;
     while(p1 != null && p2 != null){
     if(p1.val <= p2.val){
          p.next = p1;
          p1 = p1.next;
  }else{
        p.next = p2;
        p2 = p2.next;
      }
       p = p.next;
    }

   if(p1 != null)
       p.next = p1;
   if(p2 != null)
     p.next = p2;

return fakeHead.next;
   }
}
```

34. ***Given an array of integers, find two numbers such that they add up to a specific target number.***

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

For example:

```
Input: numbers={2, 7, 11, 15}, target=9 Output: index1=1, index2=2
```

Naive Approach

This problem is pretty straightforward. We can simply examine every possible pair of numbers in this integer array.

Time complexity in worst case: O(n^2).

```
public static int[] twoSum(int[] numbers, int target) {
int[] ret = new int[2];
for (int i = 0; i < numbers.length; i++) {
            for (int j = i + 1; j < numbers.length; j++) {
            if (numbers[i] + numbers[j] == target) {
                        ret[0] = i + 1;
                        ret[1] = j + 1;
}
}
}

return ret; }
```

Can we do better?

Better Solution

Use HashMap to store the target value.

```
public class Solution {
 public int[] twoSum(int[] numbers, int target) {
               HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
               int[] result = new int[2];
         for (int i = 0; i < numbers.length; i++) {
                if (map.containsKey(numbers[i])) {
               int index = map.get(numbers[i]);
               result[0] = index+1 ;
                result[1] = i+1;
               break;
  } else {
               map.put(target - numbers[i], i);
}
}
return result;
 }
 }
```

Time complexity depends on the put and get operations of HashMap which is normally O(1).

Time complexity of this solution: O(n).

35. Regular Expression Matching

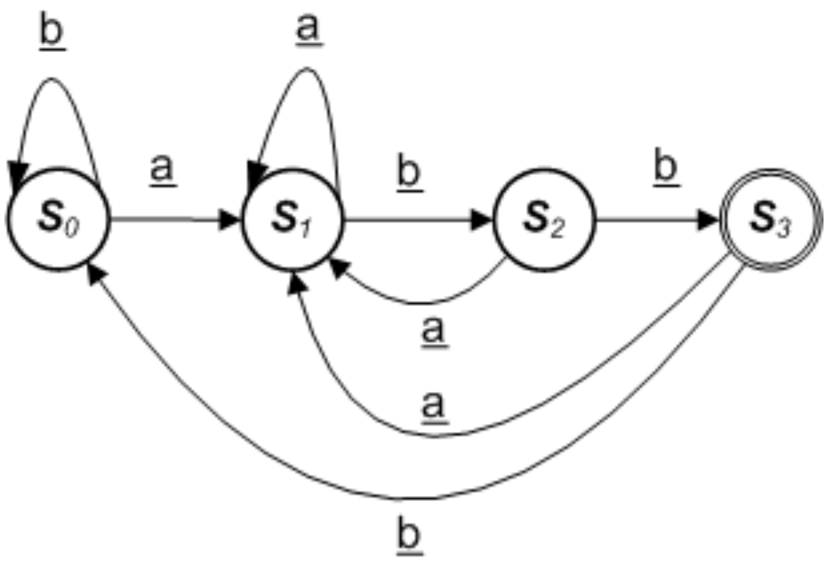
Implement regular expression matching with support for '.' and '*'.

'.' Matches any single character.
'*' Matches zero or more of the preceding element.

The matching should cover the **entire** input string (not partial).
The function prototype should be:
bool isMatch(const char *s, const char *p)

Some examples:
isMatch(“aa”,”a”) → false
isMatch(“aa”,”aa”) → true
isMatch(“aaa”,”aa”) → false
isMatch(“aa”, “a*”) → true
isMatch(“aa”, “.*”) → true
isMatch(“ab”, “.*”) → true
isMatch(“aab”, “c*a*b”) → true

Edit:
It seems that some readers are confused about why the regex pattern “.*” matches the string “ab”. “.*” means repeat the *preceding* element 0 or more times. Here, the *“preceding”* element is the **dot** character in the *pattern*, which can match any characters. Therefore, the regex pattern “.*” allows the dot to be repeated any number of times, which matches any string (even an empty string).
Hints:
Think carefully how you would do matching of ‘*’. Please note that ‘*’ in regular expression is *different* from wildcard matching, as we match the previous character 0 or more times. But, how many times? If you are stuck, recursion is your friend.
This problem is a tricky one. Due to the huge number of edge cases, many people would write lengthy code and have numerous bugs on their first try. Try your best getting your code correct first, then refactor mercilessly to as clean and concise as possible!



A sample diagram of a deterministic finite state automata (DFA). DFAs are useful for doing lexical analysis and pattern matching. An example is UNIX’s grep tool. Please note that this post does not attempt to descibe a solution using DFA.

Solution:
This looks just like a straight forward string matching, isn't it? Couldn't we just match the pattern and the input string character by character? The question is, how to match a '*'?
A natural way is to use a greedy approach; that is, we attempt to match the previous character as many as we can. Does this work? Let us look at some examples.

s = "abbbc", *p* = "ab*c"
Assume we have matched the first 'a' on both *s* and *p*. When we see "b*" in *p*, we skip all b's in *s*. Since the last 'c' matches on both side, they both match.
s = "ac", *p* = "ab*c"
After the first 'a', we see that there is no b's to skip for "b*". We match the last 'c' on both side and conclude that they both match.
It seems that being greedy is good. But how about this case?

s = "abbc", *p* = "ab*bbc"
When we see "b*" in *p*, we would have skip all b's in s. They both should match, but we have no more b's to match. Therefore, the greedy approach fails in the above case.
One might be tempted to think of a quick workaround. How about counting the number of consecutive b's in *s*? If it is smaller or equal to the number of consecutive b's after "b*" in *p*, we conclude they both match and continue from there. For the opposite, we conclude there is not a match.
This seem to solve the above problem, but how about this case:
s = "abcbcd", *p* = "a.*c.*d"
Here, ".*" in *p* means repeat '.' 0 or more times. Since '.' can match any character, it is not clear how many times '.' should be repeated. Should the 'c' in *p* matches the first or second 'c' in *s*? Unfortunately, there is no way to tell without using some kind of exhaustive search.
We need some kind of backtracking mechanism such that when a matching fails, we return to the last successful matching state and attempt to match more characters in *s* with '*'. This approach leads naturally to recursion.
The recursion mainly breaks down elegantly to the following two cases:

1. If the next character of *p* is **NOT** '*', then it must match the current character of *s*. Continue pattern matching with the next character of both *s* and *p*.
2. If the next character of *p* is '*', then we do a brute force exhaustive matching of 0, 1, or more repeats of current character of *p*… Until we could not match any more characters.

You would need to consider the base case carefully too. That would be left as an exercise to the reader.

Below is the extremely concise code (Excluding comments and asserts, it's about 10 lines of code).

```
bool isMatch(const char *s, const char *p) {
  assert(s && p);
  if (*p == '\0') return *s == '\0';

  // next char is not '*': must match current character
  if (*(p+1) != '*') {
    assert(*p != '*');
    return ((*p == *s) || (*p == '.' && *s != '\0')) && isMatch(s+1, p+1);
  }
  // next char is '*'
  while ((*p == *s) || (*p == '.' && *s != '\0')) {
    if (isMatch(s, p+2)) return true;
    s++;
  }
  return isMatch(s, p+2);
}
```

Further Thoughts:

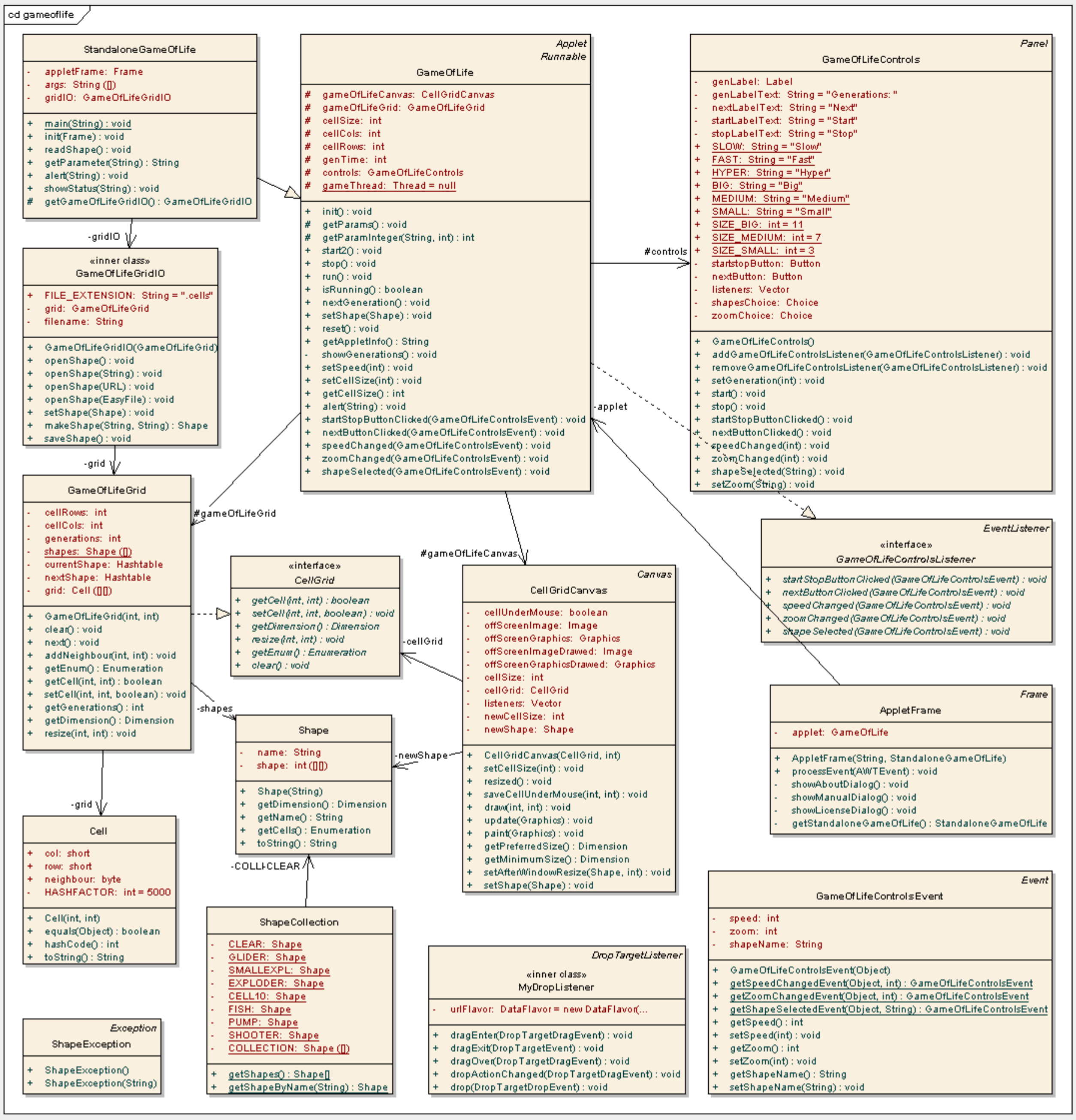
Some extra exercises to this problem:

1. If you think carefully, you can exploit some cases that the above code runs in exponential complexity. Could you think of some examples? How would you make the above code more efficient?
2. Try to implement partial matching instead of full matching. In addition, add '^' and '$' to the rule. '^' matches the starting position within the string, while '$' matches the ending position of the string.
3. Try to implement wildcard matching where '*' means any sequence of zero or more characters.

36. **Design "Game of life"**

The Java Source Code

On this page you can find the source code of the Game of Life.


cd gameoflife
StandaloneGameOfLife
- appletFrame: Frame
- args: String ([])
- gridIO: GameOfLifeGridIO
+ main(String) : void
+ init(Frame) : void
+ readShape() : void
+ getParameter(String) : String
+ alert(String) : void
+ showStatus(String) : void
# getGameOfLifeGridIO() : GameOfLifeGridIO
-gridIO
«inner class» GameOfLifeGridIO
+ FILE_EXTENSION: String = ".cells"
- grid: GameOfLifeGrid
- filename: String
+ GameOfLifeGridIO(GameOfLifeGrid)
+ openShape() : void
+ openShape(String) : void
+ openShape(URL) : void
+ openShape(EasyFile) : void
+ setShape(Shape) : void
+ makeShape(String, String) : Shape
+ saveShape() : void
-grid
GameOfLifeGrid
- cellRows: int
- cellCols: int
- generations: int
- shapes: Shape ([])
- currentShape: Hashtable
- nextShape: Hashtable
- grid: Cell ([][])
+ GameOfLifeGrid(int, int)
+ clear() : void
+ next() : void
+ addNeighbour(int, int) : void
+ getEnum() : Enumeration
+ getCell(int, int) : boolean
+ setCell(int, int, boolean) : void
+ getGenerations() : int
+ getDimension() : Dimension
+ resize(int, int) : void
-grid
Cell
+ col: short
+ row: short
+ neighbour: byte
- HASHFACTOR: int = 5000
+ Cell(int, int)
+ equals(Object) : boolean
+ hashCode() : int
+ toString() : String
Exception
ShapeException
+ ShapeException()
+ ShapeException(String)
Applet Runnable
GameOfLife
# gameOfLifeCanvas: CellGridCanvas
# gameOfLifeGrid: GameOfLifeGrid
# cellSize: int
# cellCols: int
# cellRows: int
# genTime: int
# controls: GameOfLifeControls
# gameThread: Thread = null
+ init() : void
# getParams() : void
# getParamInteger(String, int) : int
+ start2() : void
+ stop() : void
+ run() : void
+ isRunning() : boolean
+ nextGeneration() : void
+ setShape(Shape) : void
+ reset() : void
+ getAppletInfo() : String
- showGenerations() : void
+ setSpeed(int) : void
+ setCellSize(int) : void
+ getCellSize() : int
+ alert(String) : void
+ startStopButtonClicked(GameOfLifeControlsEvent) : void
+ nextButtonClicked(GameOfLifeControlsEvent) : void
+ speedChanged(GameOfLifeControlsEvent) : void
+ zoomChanged(GameOfLifeControlsEvent) : void
+ shapeSelected(GameOfLifeControlsEvent) : void
#gameOfLifeGrid
#gameOfLifeCanvas
#controls
-applet
«interface» CellGrid
+ getCell(int, int) : boolean
+ setCell(int, int, boolean) : void
+ getDimension() : Dimension
+ resize(int, int) : void
+ getEnum() : Enumeration
+ clear() : void
-cellGrid
-shapes
Shape
- name: String
- shape: int ([][])
+ Shape(String)
+ getDimension() : Dimension
+ getName() : String
+ getCells() : Enumeration
+ toString() : String
-newShape
-COLLECLEAR
ShapeCollection
- CLEAR: Shape
- GLIDER: Shape
- SMALLEXPL: Shape
- EXPLODER: Shape
- CELL10: Shape
- FISH: Shape
- PUMP: Shape
- SHOOTER: Shape
- COLLECTION: Shape ([])
+ getShapes() : Shape[]
+ getShapeByName(String) : Shape
Canvas
CellGridCanvas
- cellUnderMouse: boolean
- offScreenImage: Image
- offScreenGraphics: Graphics
- offScreenImageDrawed: Image
- offScreenGraphicsDrawed: Graphics
- cellSize: int
- cellGrid: CellGrid
- listeners: Vector
- newCellSize: int
- newShape: Shape
+ CellGridCanvas(CellGrid, int)
+ setCellSize(int) : void
+ resized() : void
+ saveCellUnderMouse(int, int) : void
+ draw(int, int) : void
+ update(Graphics) : void
+ paint(Graphics) : void
+ getPreferredSize() : Dimension
+ getMinimumSize() : Dimension
+ setAfterWindowResize(Shape, int) : void
+ setShape(Shape) : void
DropTargetListener
«inner class» MyDropListener
- urlFlavor: DataFlavor = new DataFlavor(...
+ dragEnter(DropTargetDragEvent) : void
+ dragExit(DropTargetEvent) : void
+ dragOver(DropTargetDragEvent) : void
+ dropActionChanged(DropTargetDragEvent) : void
+ drop(DropTargetDropEvent) : void
Panel
GameOfLifeControls
- genLabel: Label
- genLabelText: String = "Generations: "
- nextLabelText: String = "Next"
- startLabelText: String = "Start"
- stopLabelText: String = "Stop"
+ SLOW: String = "Slow"
+ FAST: String = "Fast"
+ HYPER: String = "Hyper"
+ BIG: String = "Big"
+ MEDIUM: String = "Medium"
+ SMALL: String = "Small"
+ SIZE_BIG: int = 11
+ SIZE_MEDIUM: int = 7
+ SIZE_SMALL: int = 3
- startstopButton: Button
- nextButton: Button
- listeners: Vector
- shapesChoice: Choice
- zoomChoice: Choice
+ GameOfLifeControls()
+ addGameOfLifeControlsListener(GameOfLifeControlsListener) : void
+ removeGameOfLifeControlsListener(GameOfLifeControlsListener) : void
+ setGeneration(int) : void
+ start() : void
+ stop() : void
+ startStopButtonClicked() : void
+ nextButtonClicked() : void
+ speedChanged(int) : void
+ zoomChanged(int) : void
+ shapeSelected(String) : void
+ setZoom(String) : void
EventListener
«interface» GameOfLifeControlsListener
+ startStopButtonClicked(GameOfLifeControlsEvent) : void
+ nextButtonClicked(GameOfLifeControlsEvent) : void
+ speedChanged(GameOfLifeControlsEvent) : void
+ zoomChanged(GameOfLifeControlsEvent) : void
+ shapeSelected(GameOfLifeControlsEvent) : void
Frame
AppletFrame
- applet: GameOfLife
+ AppletFrame(String, StandaloneGameOfLife)
+ processEvent(AWTEvent) : void
- showAboutDialog() : void
- showManualDialog() : void
- showLicenseDialog() : void
+ getStandaloneGameOfLife() : StandaloneGameOfLife
Event
GameOfLifeControlsEvent
- speed: int
- zoom: int
- shapeName: String
+ GameOfLifeControlsEvent(Object)
+ getSpeedChangedEvent(Object, int) : GameOfLifeControlsEvent
+ getZoomChangedEvent(Object, int) : GameOfLifeControlsEvent
+ getShapeSelectedEvent(Object, String) : GameOfLifeControlsEvent
+ getSpeed() : int
+ setSpeed(int) : void
+ getZoom() : int
+ setZoom(int) : void
+ getShapeName() : String
+ setShapeName(String) : void

org.bitstorm.gameoflife

    - GameOfLife.java - The code for the applet
    - StandaloneGameOfLife.java - The code for the standalone program, extends GameOfLife.java.
    - Cell.java - Represents a cell.
    - CellGrid.java - Represents the visible grid of cells.
    - CellGridCanvas.java - The Canvas showing the grid.
    - GameOfLifeControls.java - The controls (buttons, pull down menu's...).
    - GameOfLifeControlsEvent.java - Event generated by controls.
    - GameOfLifeControlsListener.java - Listener for controls.
    - GameOfLifeGrid.java - Contains the Game of Life algoritm and the current shape.
    - Shape.java - Represents a shape.
    - ShapeCollection.java - Collection of predefined shapes.
    - ShapeException.java - Exceptions of shape operations.
- **org.bitstorm.util**
    - AboutDialog.java - About Dialog box class.
    - AlertBox.java - Alert box utility class.
    - EasyFile.java - File reading and writing utility class.
    - ImageComponent.java - Image component utility class.
    - LineEnumerator.java - Enumerates over lines in a text.
    - TextFileDialog.java - Shows text in a dialog.

See also the GameOfLife JavaDoc documentation.

37. **Maximum Sum of All Sub-arrays**

Question: A sub-array has one number of some continuous numbers. Given an integer array with positive numbers and negative numbers, get the maximum sum of all sub-arrays. Time complexity should be O(n).

For example, in the array {1, -2, 3, 10, -4, 7, 2, -5}, its sub-array {3, 10, -4, 7, 2} has the maximum sum 18.

Analysis: During interviews, many candidates can solve it by enumerating all of sub-arrays and calculate their sum. An array with n elements has n(n+1)/2 sub-arrays. It costs O(n^2) time at least to calculate their sum. Usually the intuitive and forceful solution is not the most optimized one. Interviewers will tell us that there are better solutions.

Solution 1: Analyze arrays number by number

We try to accumulate every number in the example array from first to end. We initialize *sum* as 0. At our first step, add the first number 1, and *sum* is 1. And then if we add the second number -2, *sum* becomes -1. At the third step, we add the third number 3. We can notice that *sum* is less than

0, so the new *sum* will be 2 and it is less than the third number 3 itself if we add it with a negative *sum*. Therefore, the accumulated *sum* can be discarded.

We continue accumulating from the third number with *sum* 3. When we add it with the fourth number 10, *sum* becomes 13, and it decreases to 9 when we add it with the fifth number -4. We can notice that the *sum* with -4 is less than the previous *sum* since -4 is a negative number. Therefore, we should store the previous *sum* 13, since it might be the max sum of sub-arrays. At the sixth step, we add the sixth number 7 and *sum* is 16. Now *sum* is greater than the previous max sum of sub-arrays, we need to update it to 16. It is same when we add the seventh number 2. The max sum of sub-arrays is updated to 18. Lastly we add -5 and sum becomes 13. Since it is less than the previous max sum of sub-arrays, it final max sum of sub-array is 18, and the sub-array is {3, 10, -4, 7, 2} accordingly. We can summarize the whole process with the Table 1:

Step	Operation	Accumulated sum of sub-arrays	Maximum sum
1	Add 1	1	1
2	Add -2	-1	1
3	Discard previous sum -1, add 3	3	3
4	Add 10	13	13
5	Add -4	9	13
6	Add 7	16	16
7	Add 2	18	18
8	Add -5	13	18

Table 1: The process to calculate the maximum sum of all sub-arrays in the array {1, -2, 3, 10, -4, 7, 2, -5}

After we get clear ideas of this solution, we are ready to develop some code. The following is the sample code:

```
bool g_InvalidInput = false;

int FindGreatestSumOfSubArray(int *pData, int nLength)
{
   if((pData == NULL) || (nLength <= 0))
   {
      g_InvalidInput = true;
      return 0;
   }

   g_InvalidInput = false;

   int nCurSum = 0;
   int nGreatestSum = 0x80000000;
   for(int i = 0; i < nLength; ++i)
   {
      if(nCurSum <= 0)
         nCurSum = pData[i];
```

```
        else
            nCurSum += pData[i];

        if(nCurSum > nGreatestSum)
            nGreatestSum = nCurSum;
    }

    return nGreatestSum;
}
```

We should keep invalid inputs during interview, such as the pointer parameter of array is NULL or its length is 0. What should be return for the invalid inputs? If it is 0, how can we distinguish the two situations: one is for the actual maximum sum of sub-arrays is 0 and the other is for invalid inputs? Therefore, we define a global variable to make whether inputs are invalid or not.

Solution 2: Dynamic programming

If we have a deep understanding of algorithm, we may solve this problem with dynamic programming. If function *f(i)* stands for the maximum sum of a sum-array ended with the ith number, what it is to get is *max[f(i)]*. We can calculate *f(i)* with the following recursive equation:

$$f(i) = \begin{cases} number[i] & i = 0 \text{ or } f(i-1) \le 0 \\ f(i-1) + number[i] & i \ne 0 \text{ and } f(i-1) > 0 \end{cases}$$

In the equation above, if the sum of sub-array ended with the (i-1)th number is negative, the sum of sub-array ended with the ith number should be the ith number itself (it is the third step in the Table 1). Otherwise, we can get the sum of sub-array ended with the ith number by adding the ith number and the sum of sub-array ended with the previous number.

Even though we analyze the problem recursively, we will write code based on iteration eventually. The iterative code according to the equation above should be save to the code of the first solution. nCursum is the *f(i)* in the equation, and nGreatestSum is *max[f(i)]*. Therefore, these two solutions are essentially identical to each other.

38. Dynamic Programming on Stolen Values

Problem: There are *n* houses built in a line, each of which contains some value in it. A thief is going to steal the maximal value in these houses, but he cannot steal in two adjacent houses because the owner of a stolen house will tell his two neighbors on the left and right side. What is the maximal stolen value?

For example, if there are four houses with values {6, 1, 2, 7}, the maximal stolen value is 13 when the first and fourth houses are stolen.

Analysis: A function *f*(*i*) is defined to denote the maximal stolen value from the first house to the

i^{th} house, and the value contained in the i^{th} house is denoted as v_i. When the thief reaches the i^{th} house, he has two choices: to steal or not. Therefore, $f(i)$ can be defined with the following equation:

$$f(i) = \max[v_i + f(i-2) \text{ when the } i^{th} \text{ house is stolen}, f(i-1) \text{ otherwise}]$$

It would be much more efficient to calculate in bottom-up order than to calculate recursively. It looks like a 1D array with size *n* is needed, but actually it is only necessary to cache two values for $f(i\text{-}1)$ and $f(i\text{-}2)$ to calculate $f(i)$.

This algorithm can be implemented with the following C++ code:

```
int maxStolenValue(const vector<int>& values)
{
   int length = values.size();
   if(length == 0)
      return 0;

   int value1 = values[0];
   if(length == 1)
      return value1;

   int value2 = max<int>(values[0], values[1]);
   if(length == 2)
      return value2;

   int value;
   for(int i = 2; i < length; ++i)
   {
      value = max<int>(value2, value1 + values[i]);
      value1 = value2;
      value2 = value;
   }

   return value;

}
```

39.**Design question**

1. 入门级的news feed
http://www.quora.com/What-are-best-practices-for-building-somet
http://www.infoq.com/presentations/Scale-at-Facebook
http://www.infoq.com/presentations/Facebook-Software-Stack
一般的followup question是估算需要多少server
这篇文章稍微提到要怎么approach这种题，可以稍微看看

http://book.douban.com/reading/23757677/

2. facebook chat,这个也算是挺常问的
http://www.erlang-factory.com/upload/presentations/31/EugeneLet
https://www.facebook.com/note.php?note_id=14218138919
http://www.cnblogs.com/piaoger/archive/2012/08/19/2646530.html
http://essay.utwente.nl/59204/1/scriptie_J_Schipers.pdf

3. typeahead search/search suggestion，这个也常见
https://www.facebook.com/video/video.php?v=432864835468
问题在这个帖子里被讨论到，基本上每个问题，在视频里都有回答

4. Facebook Messaging System(有提到inbox search, which has been asked before）
messaging system就是一个把所有chat/sms/email之类的都结合起来的一个系统
http://www.infoq.com/presentations/HBase-at-Facebook
http://sites.computer.org/debull/A12june/facebook.pdf
http://www.slideshare.net/brizzzdotcom/facebook-messages-hbase/
https://www.youtube.com/watch?v=UaGINWPK068

5. 任给一个手机的位置信号(经纬度)，需要返回附近5mile 的POI

6. Implement second/minute/hour/day counters
这题真不觉得是system design，但万一问道，还是要有准备，貌似在总部面试会被问道....

7. facebook photo storage，这个不太会被问起，但是知道也不错
https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Beaver.pdf
https://www.facebook.com/note.php?note_id=76191543919

8. facebook timeline,这个也不太是个考题，看看就行了
https://www.facebook.com/note.php?note_id=10150468255628920
http://highscalability.com/blog/2012/1/23/facebook-timeline-bro

除了这些，准备一下这些题目
implement memcache
http://www.adayinthelifeof.nl/2011/02/06/memcache-internals/

implement tinyurl（以及distribute across multiple servers)
http://stackoverflow.com/questions/742013/how-to-code-a-url-sho

determine trending topics(twitter)
http://www.americanscientist.org/issues/pub/the-britney-spears-
http://www.michael-noll.com/blog/2013/01/18/implementing-real-t

copy one file to multiple servers
http://vimeo.com/11280885

稍微知道一下dynamo key value store，以及google的gfs和big table

另外推荐一些网站
http://highscalability.com/blog/category/facebook
这个high scalability上有很多讲system design的东西，不光是facebook的，没空的，就光看你要面试的那家就好了..
facebook engineering blog
http://www.quora.com/Facebook-Engineering/What-is-Facebooks-arc
http://stackoverflow.com/questions/3533948/facebook-architectur

其他家的
http://www.quora.com/What-are-the-top-startup-engineering-blogs

==
在说说怎么准备这样的面试
首先如果你连availability/scalability/consistency/partition之类的都不是太有概念的话，我建议先去wikipedia或者找一个某个大学讲这门课的网站稍微看一下，别一点都不知道这个链接也不错
http://www.aosabook.org/en/distsys.html

如果你这些基本的东西都还知道，那么我觉得你就和大部分毫无实际经验的人差不多一个水平...能做的就是一点一点去准备，如果你还有充足的时间的话，建议从你面试的那家公司的engineering blog看起，把人家用的technology stack/product都搞清楚，然后在把能找到的面试题都做一遍呗....我们做coding题说白了不也是题海战术...而且你如果坚持看下去，真的会看出心得，你会发现很多地方都有相同之处，看多了就也能照葫芦画瓢了...

再有就是面试的时候应该怎么去approach这种题，我说说我的做法
1. product spec/usage scenario 和面试者confirm这个东西到底是做什么的可以先列出来几个major functionality，然后有时间的话，再补充一些不重要的把你想的都写下来

2. define some major components
就是画几个圈圈框框的，每个发表一番您的高见....然后讲他们之间怎么interact

以上是question specific的东西，

这个讲完了，我们可以讲一些每道题都是用的，比如说怎么scale/怎么partition/怎么实现consistency，这些东西，可以套用到任何题上

当然了，我们遇到的题和解题的方法可能都有些出入，不见得每道题有一个路数下来，最重要的是，讲题的时候要有条理，画图要清楚，保持和面试官的交流，随时问一下人家的意见。

40. Number of 1 in a Binary

Problem: Please implement a function to get the number of 1s in an integer. For example, the integer 9 is 1001 in binary, so it returns 2 since there are two bits of 1s.

Analysis: It looks like a simple question about binary numbers, and we have many solutions for it. Unfortunately, the most intuitive solution for many candidates is incorrect. We should be careful.

Solution 1: Check the most right bit, possibly with endless loop

When candidates are interviewed with this problem, many of them find a solution in short time: check whether the most right bit is 0 or 1, and then right shit the integer one bit and check the most right bit again. It continues in a loop until the integer becomes 0. How to check whether the most right bit of an integer is 0 or 1? It is simple since we have AND operation. There is only one bit of 1, which is the most right bit, in the binary format of integer 1. When we have AND operation on an integer and 1, we can check whether the most right bit is 0 or 1. When the result of AND operation is 1, it indicates the right most bit is 1; otherwise it is 0. We can implement a function base on this solution quickly:

```
int NumberOf1(int n)
{
    int count = 0;
    while(n)
    {
        if(n & 1)
            count ++;

        n = n >> 1;
    }

    return count;
}
```

Interviewers may ask a question when they are told this solution: What is the result when the input integer is a negative number such as 0x80000000? When we right shift the negative number 0x80000000 a bit, it become 0xC0000000 rather than 0x40000000, which is the result to move the first bit of 1 to the second bit. The integer 0x8000000 is negative before shift, so we should guarantee it is also negative after shift. Therefore, when a negative integer is right shifted, the first bit is set as 1 after the right shift operation. If we continue to shift to right side, a negative integer will be 0xFFFFFFFF eventually and it is trapped in an endless loop.

Solution 2: Check the most right bit, with left shift operation on 1

We should not right shift the input integer to avoid endless loop. Instead of shifting the input integer n to right, we may shift the number 1 to left. We may check firstly the least important bit of n, and then shift the number 1 to left, and continue to check the second least important bit of n. Now we can rewrite our code based on this solution:

```
int NumberOf1(int n)
{
   int count = 0;
   unsigned int flag = 1;
   while(flag)
   {
      if(n & flag)
         count ++;

      flag = flag << 1;
   }

   return count;
}
```

In the code above, it loops 32 times on 32-bit numbers.

Solution 3: Creative solution

Let us analyze that what happens when a number minus 1. There is at least one bit 1 in a non-zero number. We firstly assume the right most bit is 1. It becomes 0 if the number minus 1 and other bits keep unchanged. This result is identical to the not operation on the most right bit of a number, which also turns the right most bit from 1 into 0.

Secondly we assume the right most bit is 0. Since there is at least one bit 1 in a non-zero number, we suppose the mth bit is the most right bit of 1. When it minus 1, the mth bit becomes 0, and all 0 bits behind the mth bit become 1. For instance, the second bit of binary number 1100 is the most right bit of 1. When it minus 1, the second bit becomes 0, and the third and fourth bits become 1, so the result is 1011.

In both situations above, the most right of bit 1 becomes 0 when it minus 1. When there are some

0 bits at right side, all of them become 1. The result of bit and operation on the original number and minus result is identical to result to modify the most right 1 into 0. Take the binary number 1100 as an example again. Its result is 1011 when it minus 1. The result of and operation on 1100 and 1011 is 1000. If we change the most right of 1 bit in number 1100, it also becomes 1000.

The analysis can be summarized as: If we first minus a number with 1, and have and operation on the original number and the minus result, the most right of 1 bit becomes 0. We continue these operations until the number becomes 0. We can develop the following code accordingly:

```
int NumberOf1(int n)
{
    int count = 0;

    while (n)
    {
        ++ count;
        n = (n - 1) & n;
    }

    return count;
}
```

The number of times in the while loops equals to the number of 1 in the binary format of input n.

41. Numbers Appearing Once

Problem: In an array, all numbers appear three times except one which only appears only once. Please find the unique number.

Analysis: It is simpler if we modify the problem a little bit: Please find a unique number from an array where other numbers appear twice. We could solve this simplified problem with the XOR bitwise operation. If all numbers in the array are XORed, the result is the number appearing only once, since pairs of numbers get 0 when they are XORed.

The strategy with XOR does not work since all numbers except one appear three times, since the XOR result of a triple of numbers is the number itself.

Even though we could not solve the problem with XOR, we may still stick on the bitwise operations. A number appears three times, each bit (either 0 or 1) also appears three times. If every bit of numbers appearing three time is added, the sum of every bit should be multiple of 3.

Supposing every bit of numbers (including the unique number) in the input array is added. If the sum of a bit is multiple of 3, the corresponding bit in the unique number is 0. Otherwise, it is 1.

The solution can be implemented in Java as the code listed below:

```
public static int FindNumberAppearingOnce(int[] numbers) {
    int[] bitSum = new int[32];
    for(int i = 0; i < 32; ++i) {
        bitSum[i] = 0;
    }

    for(int i = 0; i < numbers.length; ++i) {
        int bitMask = 1;
        for(int j = 31; j >= 0; --j) {
            int bit = numbers[i] & bitMask;
            if(bit != 0) {
                bitSum[j] += 1;
            }

            bitMask = bitMask << 1;
        }
    }

    int result = 0;
    for(int i = 0; i < 32; ++i) {
        result = result << 1;
        result += bitSum[i] % 3;
    }

    return result;
}
```

The time efficiency of this solution is O(*n*), and space efficiency is O(1) because an array with 32 elements is created. It's more efficient than two straightforward solutions: (1) It's easy to find the unique number from a sorted array, but it costs O(*n*log*n*) time to sort an array with *n* elements. (2) We may utilize a hash table to store the number of occurrences of each element in the array, but the cost for the hash table is O(*n*).

42. Next Number with Same Set of Digits

Problem: Reorder the digits of a number, in order to get the next number which is the least one that is greater than the input number. For example, the number 34724126 is the next number of 34722641 when reordering digits.

}

<u>**Analysis:**</u> When a digit in a number is swapped with a greater digit on its right side, the number becomes greater. For example, if the digit 3 in the number 34722641 is swapped with digit 7, the result is 74322641 which is greater than the original number. The remaining issue how to get least number which is greater than the original one.

Since we are going to get the least number after reordering digits, let's find digits to be swapped on the right side. The first three digits on the right side of 34722641 are 641 which are decreasingly sorted. The two digits among them are swapped, the whole number will become less.

Let's move on to the left digits. The next digit on the right side is 2, which is less than 6 and 4 on its right side. If the digit 2 is swapped with 6 or 4, the whole number will be become greater. Since we are going to keep the swapped number as less as possible, the digit 2 is swapped with 4, which is less one between 4 and 6. The number becomes 34724621.

Now 34724621 is greater than the original number 34722641, but it's not the least one which is greater than 34722641. The three digits on the right side, 6, 2 and 1, should be increasingly sorted, in order to form the least number 34724126 among numbers which are greater than of 34722641.

The solution can be implemented with the following JAVA code:

```java
public static String getLeastGreaterNumber(String number) {
   List<Character> decreasingChars = new ArrayList();
   int firstDecreasing = getDecreasingChars(number, decreasingChars);

   if(isGreatestNumber(firstDecreasing)) {
      return "";
   }

   String prefix = "";
   if(firstDecreasing > 1) {
      prefix = number.substring(0, firstDecreasing - 1);
   }

   StringBuilder resultBuilder = new StringBuilder(prefix);
   char target = number.charAt(firstDecreasing - 1);
   char leastGreater = swapLeastGreater(decreasingChars, target);
   resultBuilder.append(leastGreater);

   Collections.sort(decreasingChars);
   appendList(resultBuilder, decreasingChars);

   return resultBuilder.toString();
}
```

When all digits are already increasingly sorted in the input number, the number itself is the greatest number with given digits. We should discuss with our interviewers what to return for this case during interviewers. Here we just return an empty string for this case.

When firstDecreasing is 0, it means all digits are increasingly sorted, and the input number is the greatest number with given digits, as listed in the following method isGreatestNumber.

```
private static Boolean isGreatestNumber(int firstDecreasing) {
    return firstDecreasing == 0;
}
```

The following method getDecreasingChars gets the longest sequence of decreasing digits on the right side of a number:

```
private static int getDecreasingChars(String number, List<Character> decreasing) {
    int firstDecreasing = number.length() - 1;
    for(; firstDecreasing > 0; --firstDecreasing) {
        char curChar = number.charAt(firstDecreasing);
        char preChar = number.charAt(firstDecreasing - 1);
        decreasing.add(curChar);

        if(curChar > preChar) {
            break;
        }
    }

    return firstDecreasing;
}
```

The following method swapLeastGreater swaps the digit before the decreasing digits on the right side (target) and the least digit which is greater than target:

```
private static char swapLeastGreater(List<Character> chars, char target) {
    Iterator it=chars.iterator();
    char finding = '9';
    while(it.hasNext()) {
        char value = ((Character)it.next()).charValue();
        if(value > target && value < finding) {
            finding = value;
        }
    }

    chars.remove(new Character(finding));
    chars.add(new Character(target));
    return finding;
}
```

The following method appendList appends characters from a list into a string builder:

```
private static void appendList(StringBuilder str, List<Character> chars) {
    Iterator it=chars.iterator();
    while(it.hasNext()) {
        char value = ((Character)it.next()).charValue();
        str.append(value);
    }
}
```

Code with unit tests is shared at http://ideone.com/czs13W.

Extended Problem 1: Given a set of digits, please output all numbers permutated by the digits in increasing order. For example, if the input are five digits 1, 2, 3, 4, 5, the output are numbers from 12345, 12354, ..., to 54321 in increasing order.

Extended Problem 2: Given a number *n*, please out put all numbers with *n* bits of 1 in increasing order. For example, if the input is 3, the output are numbers 7, 11, 13, …

43. Median in Stream

Question: How to get the median from a stream of numbers at any time? The median is middle value of numbers. If the count of numbers is even, the median is defined as the average value of the two numbers in middle.

Analysis: Since numbers come from a stream, the count of numbers is dynamic, and increases over time. If a data container is defined for the numbers from a stream, new numbers will be inserted into the container when they are deserialized. Let us find an appropriate data structure for such a data container.

An array is the simplest choice. The array should be sorted, because we are going to get its median. Even though it only costs O(lg*n*) time to find the position to be inserted with binary search algorithm, it costs O(*n*) time to insert a number into a sorted array, because O(*n*) numbers will be moved if there are *n* numbers in the array. It is very efficient to get the median, since it only takes O(1) time to access to a number in an array with an index.

A sorted list is another choice. It takes O(*n*) time to find the appropriate position to insert a new number. Additionally, the time to get the median can be optimized to O(1) if we define two pointers which points to the central one or two elements.

A better choice available is a binary search tree, because it only costs O(lg*n*) on average to insert a new node. However, the time complexity is O(*n*) for the worst cases, when numbers are inserted in sorted (increasingly or decreasingly) order. To get the median number from a binary search tree, auxiliary data to record the number of nodes of its sub-tree is necessary for each node. It also requires O(lg*n*) time to get the median node on overage, but O(*n*) time for the worst cases.

We may utilize a balanced binary search tree, AVL, to avoid the worst cases. Usually the balance factor of a node in AVL trees is the height difference between its right sub-tree and left sub-tree. We may modify a little bit here: Define the balance factor as the difference of number of nodes between its right sub-tree and left sub-tree. It costs O(lg*n*) time to insert a new node into an AVL, and O(1) time to get the median for all cases.

An AVL is efficient, but it is not implemented unfortunately in libraries of the most common programming languages. It is also very difficult for candidates to implement the left/right rotation of AVL trees in dozens of minutes during interview. Let us looks for better solutions.

As shown in Figure 1, if all numbers are sorted, the numbers which are related to the median are indexed by P1 and P2. If the count of numbers is odd, P1 and P2 point to the same central number. If the count is even, P1 and P2 point to two numbers in middle.

Median can be get or calculated with the numbers pointed by P1 are P2. It is noticeable that all numbers are divided into two parts. The numbers in the first half are less than the numbers in the second half. Moreover, the number indexed by P1 is the greatest number in the first half, and the number indexed by P2 is the least one in the second half.

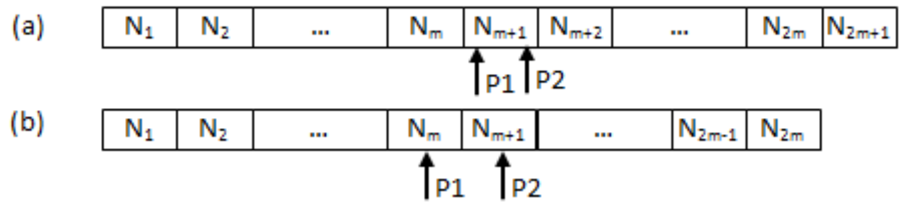

(a)
N1 N2 ... Nm Nm+1 Nm+2 ... N2m N2m+1
P1 P2
(b)
N1 N2 ... Nm Nm+1 ... N2m-1 N2m
P1 P2

Figure 1: Numbers are divided in two parts by one or two numbers in its center.

If numbers are divided into two parts, and all numbers in the first half is less than the numbers in the second half, we can get the median with the greatest number of the first part and the least number of the second part. How to get the greatest number efficiently? Utilizing a max heap. It is also efficient to get the least number with a min heap.

Therefore, numbers in the first half are inserted into a max heap, and numbers in the second half are inserted into a min heap. It costs O(lg*n*) time to insert a number into a heap. Since the median can be get or calculated with the root of a min heap and a max heap, it only takes O(1) time.

Table 1 compares the solutions above with a sorted array, a sorted list, a binary search tree, an AVL tree, as well as a min heap and a max heap.

Type for Data Container	Time to Insert	Time to Get Median
Sorted Array	O(*n*)	O(1)
Sorted List	O(*n*)	O(1)

Binary Search Tree	O(lg$n$) on average, O(n) for the worst cases	O(lg$n$) on average, O(n) for the worst cases
AVL	O(lg$n$)	O(1)
Max Heap and Min Heap	O(lg$n$)	O(1)

Table 1: Summary of solutions with a sorted array, a sorted list, a binary search tree, an AVL tree, as well as a min heap and a max heap.

Let us consider the implementation details. All numbers should be evenly divided into two parts, so the count of number in min heap and max heap should diff 1 at most. To achieve such a division, a new number is inserted into the min heap if the count of existing numbers is even; otherwise it is inserted into the max heap.

We also should make sure that the numbers in the max heap are less than the numbers in the min heap. Supposing the count of existing numbers is even, a new number will be inserted into the min heap. If the new number is less than some numbers in the max heap, it violates our rule that all numbers in the min heap should be greater than numbers in the min heap.

In such a case, we can insert the new number into the max heap first, and then pop the greatest number from the max heap, and push it into the min heap. Since the number pushed into the min heap is the former greatest number in the max heap, all numbers in the min heap are greater than numbers in the max heap with the newly inserted number.

The situation is similar when the count of existing numbers is odd and the new number to be inserted is greater than some numbers in the min heap. Please analyze the insertion process carefully by yourself.

The following is sample code in C++. Even though there are no types for heaps in STL, we can build heaps with vectors utilizing function push_heap and pop_heap. Comparing functor less and greater are employed for max heaps and min heaps correspondingly.

```
template<typename T> class DynamicArray
{
public:
   void Insert(T num)
   {
      if(((minHeap.size() + maxHeap.size()) & 1) == 0)
      {
         if(maxHeap.size() > 0 && num < maxHeap[0])
         {
            maxHeap.push_back(num);
            push_heap(maxHeap.begin(), maxHeap.end(), less<T>());

            num = maxHeap[0];

            pop_heap(maxHeap.begin(), maxHeap.end(), less<T>());
            maxHeap.pop_back();
            }

            minHeap.push_back(num);
            push_heap(minHeap.begin(), minHeap.end(), greater<T>());
        }
        else
        {
            if(minHeap.size() > 0 && minHeap[0] < num)
            {
                minHeap.push_back(num);
                push_heap(minHeap.begin(), minHeap.end(), greater<T>());

                num = minHeap[0];

                pop_heap(minHeap.begin(), minHeap.end(), greater<T>());
                minHeap.pop_back();
            }

            maxHeap.push_back(num);
            push_heap(maxHeap.begin(), maxHeap.end(), less<T>());
        }
    }

    int GetMedian()
    {
        int size = minHeap.size() + maxHeap.size();
        if(size == 0)
            throw exception("No numbers are available");

        T median = 0;
        if(size & 1 == 1)
            median = minHeap[0];
        else
            median = (minHeap[0] + maxHeap[0]) / 2;

        return median;
    }

private:
    vector<T> minHeap;
    vector<T> maxHeap;
};
```

In the code above, function Insert is used to insert a new number deserialized from a stream, and GetMedian is used to get the median of the existing numbers dynamically.

44. Permutation

Questions: Please print all permutations of a given string. For example, print “abc”, “acb”, “bac”, “bca”, “cab”, and “cba” when given the input string “abc”.

Analysis: For many candidates, it is not a simple problem to get all permutations of a set of characters. In order to solve such a problem, candidates might try to divide it into simple subproblems. An input string is partitioned into two parts. The first part only contains the first characters, and the second part contains others. As shown in Figure 1, a string is divided into two parts with different background colors.

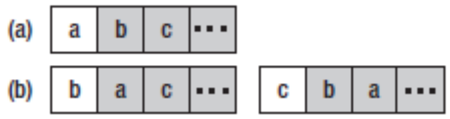


Figure 1: The process to get permutations of a string. (a) A string is divided into two parts of which the first part only contains the first character, and the second part contains others (with gray background). (b)
All characters in the second part are swapped with the first character one by one.

This solution gets permutations of a given string with two steps. The first step is to swap the first character with the following characters one by one. The second step is to get permutations of the string excluding the first character. Take the sample string “abc” as an example. It gets permutations of “bc” when the first character is ‘a’. It then swaps the first character with ‘b’, gets permutations of “ac”, and finally gets permutation of “ba” after swapping the first character with ‘c’.

The process to get permutations of a string excluding the first character is similar to the process to
get permutations of a whole string. Therefore, it can be solved recursively, as shown below:

```
void Permutation(char* pStr)
{
   if(pStr == NULL)
      return;
   PermutationCore(pStr, pStr);
}

void PermutationCore(char* pStr, char* pBegin)
{
   char *pCh = NULL;
   char temp;
   if(*pBegin == '\0')
   {
      printf("%s\n", pStr);
   }
```

```
    else
    {
        for(pCh = pBegin; *pCh != '\0'; ++ pCh)
        {
            temp = *pCh;
            *pCh = *pBegin;
            *pBegin = temp;

            PermutationCore(pStr, pBegin + 1);

            temp = *pCh;
            *pCh = *pBegin;
            *pBegin = temp;
        }
    }
}
```