Cracking the Coding Interview – Solutions

Heyan Huang

Contents

Chapter 1

Arrays and Strings

1.1 Implement an algorithm to determine if a string has all unique characters. What if you can not use additional data structures?

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

bool isUnique1(string s) {
  bool set[256];
  memset(set, 0, sizeof(set));
  int len = s.length();
  for (int i = 0; i < len; i++) {
    int val = (int)s[i];
    if (set[val])
      return false;
    set[val] = true;
  }
  return true;
}

bool isUnique2(string s) {
  int A[8];
  memset(A, 0, sizeof(A));
  int len = s.length();
  for (int i = 0; i < len; i++) {
    int val = (int)s[i];
    int index = val/32;
    int shift = val % 32;
    if (A[index] & (1<<shift))
      return false;
    // A[index] ^ (1<<shift); // or A[index] | (1<<shift);  They are wrong!!!
    A[index] ^= (1<<shift); // or A[index] |= (1<<shift);
  }
  return true;
}

bool isUnique3(string s) {
    int check = 0;
    int len = s.length();
```

```
    for(int i=0; i < len; ++i) {
        int v = (int)(s[i]-'a');
        if(check & (1 << v)) return false;
        check |= (1 << v);
    }
    return true;
}

int main() {
    string s1 = "i am hawstein.";
    string s2 = "abcdefghijklmnopqrstuvwxyzABCD1234567890";
    cout << isUnique1(s1) << " " << isUnique1(s2) << endl;
    cout << isUnique2(s1) << " " << isUnique2(s2) << endl;
    return 0;
}
```

```
def has_unique_chars(string):
    # Strategy: Linearly check if the char of the string belongs to the dictionary, other
    # Time complexity is O(n) and space complexity is O(n)
    dictionary = {} #technically, a boolean array (for the char set) would do the job for
    for i in string:
        if i in dictionary:
            return False
        else:
            dictionary[i] = True
    return True

# What if you can't use an additional data structure?

def has_unique_chars2(string):
    # Stragety: Check every char of the string with every other char of the string
    # Time complexity is O(n^2) and space complexity is O(1)
    for i in xrange(len(string)):
        for j in xrange(i + 1, len(string)):
            if string[i] == string[j]:
                return False
    return True

# If destroying the input is allowed:

def has_unique_chars3(string):
    # Strategy: Sort the string inline and linearly check the string for identical neighb
    # Time complexity is O(n * log(n) + n => n * log(n)) and space complexity is O(1)
    string = ''.join(sorted(string)) #technically, this creates a new list and then joins
    for i in xrange(len(string) - 1):
        if string[i] == string[i+1]:
            return False
    return True

if __name__ == '__main__':
    case1 = 'abcdefghijklmnopqrstuvwxyz'
    case2 = 'abcdefghijklmabcdefghijklm'
    print has_unique_chars(case1)
    print has_unique_chars(case2)
    print has_unique_chars2(case1)
```

```
40      print has_unique_chars2(case2)
41      print has_unique_chars3(case1)
42      print has_unique_chars3(case2)
```

1.2 Write code to reverse a C-Style String. (C-String means that "abcd" is represented as five characters, including the null character.)

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  void swap(char &a, char &b){   // not understanding yet
6      a = a^b;
7      b = a^b;
8      a = a^b;
9  }
10
11 void reverse(char* s) {
12   int len = strlen(s);
13   char tmp;
14   for (int i = 0; i < len/2; ++i) {
15     tmp = s[len-1-i];
16     s[len-1-i] = s[i];
17     s[i] = tmp;
18   }
19 }
20
21 void reverseP(char* s) {
22   if(!s) return; // I forgot this line at beginning
23   char *f = s, *b = s;
24   char tmp;
25   while(*b) b++;
26   b--;
27   //while(f != b) {
28   while(f < b) {
29     tmp = *f;
30     *f = *b;
31     *b = tmp;
32     f++;
33     b--;
34   }
35 }
36
37 void reverseFinal(char* str) {
38   char * end = str;
39   char tmp;
40   if (str) {            // equvilent to: if(!str) return;
41     while(*end) ++end;
42     --end;
43     while(str < end) {
44       tmp = *str;
45       *str++ = *end;
46       *end-- = tmp;
47     }
48   }
```

```
49 }
50
51 void reverse2(char *s){
52     int n = strlen(s);
53     for(int i=0; i<n/2; ++i)
54         swap(s[i], s[n-i-1]);
55 }
56
57 void reverse1(char *s){
58     if(!s) return;
59     char *p = s, *q = s;
60     while(*q) ++q;
61     --q;
62     while(p < q)
63         swap(*p++, *q--);
64 }
65
66 int main(){
67   char s[] = "1234567890";
68   char a[] = "abcdefg";
69   char b[]= "hijklmn";
70   char c[]= "hijklmn";
71   char d[]= "hijklmn";
72   reverse(b);
73   reverseP(c);
74   reverseFinal(d);
75   reverse1(s);
76   reverse2(a);
77   cout << "reverse:" << b << endl;
78   cout << "reverseP:" << c << endl;
79   cout << "reverseFinal:" << d << endl;
80   cout << "reverse1:" << s << endl;
81   cout << "reverse2:" << a << endl;
82   return 0;
83 }
```

1.3 Design an algorithm and write code to remove the duplicate characters in a string without using any additional buffer. NOTE: One or two additional variables are fine. An extra copy of the array is not. FOLLOW UP Write the test cases for this method.

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

string removeDuplicate1(char s[]) {
    int len = strlen(s);
    if (len < 2)
        return s;
    string str = "";

    int a = 0;
    for (int i = 0; i < len; ++i) {
        int shift = (int)(s[i] - 'a');
        if ( (a & (1 << shift)) == 0 ) {
            str += s[i];
            a |= (1 << shift);
        }
    }
    return str;
}

string removeDuplicate2(char s[]) {
    int len = strlen(s);
    if (len < 2)
        return s;
    string str = "";

    for(int i = 0; i < len; ++i) {
        if (s[i] != '\0') {
            str += s[i];
            for (int j = i+1; j < len; ++j)
                if ( s[j] == s[i] )
                    s[j] = '\0';
        }
    }
    return str;
}

void removeDuplicate3(char s[]) {
    int len = strlen(s);
    if (len < 2) return;

    int p = 0;
    for(int i = 0; i < len; ++i) {
        if (s[i] != '\0') {
            s[p++] = s[i];
            for (int j = i+1; j < len; ++j)
                if ( s[j] == s[i] )
                    s[j] = '\0';
        }
    }
```

```
52      s[p] = '\0';
53  }
54
55  void removeDuplicate4(char s[]) {
56      int len = strlen(s);
57      if (len < 2) return;
58
59      bool flag[256];
60      memset(flag, 0, sizeof(flag));
61
62      int p = 0;
63      for (int i = 0; i < len; ++i) {
64          if (!flag[s[i]]) {
65              s[p++] = s[i];
66              flag[s[i]] = true;
67          }
68      }
69      s[p] = '\0';
70  }
71
72  void removeDuplicate5(char s[]) {
73      int len = strlen(s);
74      if (len < 2) return;
75
76      int a = 0, p = 0;
77      for (int i = 0; i < len; ++i) {
78          int shift = (int)(s[i] - 'a');
79          if ( !(a & (1 << shift)) ) {
80              s[p++] = s[i];
81              a |= 1 << shift;
82          }
83      }
84      s[p] = '\0';
85  }
86
87  int main() {
88      char a[] = "abcdefg";
89      char b[] = "aaaaabbbbbccccc";
90      char c[] = "";
91      char d[] = "abcdefgadchijkadefgabc";
92      char e[] = "aaaaaaaaa";
93      char a2[] = "abcdefg";
94      char b2[] = "aaaaabbbbbccccc";
95      char c2[] = "";
96      char d2[] = "abcdefgadchijkadefgabc";
97      char e2[] = "aaaaaaaaa";
98      cout << removeDuplicate1(a) << "␣" << removeDuplicate2(a2) << endl;
99      cout << removeDuplicate1(b) << "␣" << removeDuplicate2(b2) << endl;
100     cout << removeDuplicate1(c) << "␣" << removeDuplicate2(c2) << endl;
101     cout << removeDuplicate1(d) << "␣" << removeDuplicate2(d2) << endl;
```

1.4 Write a method to decide if two strings are anagrams or not.

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

bool isAnagram1 (string s, string t) {
    if (s == "" || t == "") return false;
    if (s.length() != t.length()) return false;

    sort(&s[0], &s[0]+s.length());
    sort(&t[0], &t[0]+t.length());
    if (s == t) return true;
    else return false;
}

bool isAnagram2 (string s, string t) {
    if (s == "" || t == "") return false;
    if (s.length() != t.length()) return false;

    int a[256];
    memset(a, 0, sizeof(a));

    int len = s.length();
    for (int i = 0; i < len; ++i) {
        ++a[s[i]];
        --a[t[i]];
    }

    for (int i = 0; i < 256; ++i)
        if (a[i])
            return false;
    return true;
}

int main() {
    string s1 = "aeiou";
    string s2 = "uoeai";
    cout << isAnagram1(s1, s2) << endl;
    cout << isAnagram2(s1, s2) << endl;
    return 0;
}
```

1.5 Write a method to replace all spaces in a string with '%20'.

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

char* replace1 (char* c) {
    if (c == NULL) return NULL; // didn't notice this one

    int len = strlen(c);    // didn't notice this one
    if (len == 0) return NULL;

    int cnt = 0;
    for (int i = 0; i < len; ++i) {
        if (c[i] == ' ')
            ++cnt;
    }

    char* cc = new char[len + 2*cnt];    // something I missed here!!!
    int p = 0;
    for (int i = 0; i < len; ++i) {
        if (c[i] == ' ') {
            cc[p++] = '%';
            cc[p++] = '2';
            cc[p++] = '0';
        } else
            cc[p++] = c[i];
    }
    cc[p] = '\0';
    return cc;
}

void replace2 (char* c) {
    if (c == NULL) return; // didn't notice this one

    int len = strlen(c);    // didn't notice this one
    if (len == 0) return;

    int cnt = 0;
    for (int i = 0; i < len; ++i) {
        if (c[i] == ' ')
            ++cnt;
    }

    int p = len + 2 * cnt;
    c[p--] = '\0';     // the space must be allocated first!
    for (int i = len-1; i >= 0; --i) {
        if (c[i] == ' ') {
            c[p] = '0';
            c[p - 1] = '2';
            c[p - 2] = '%';
            p -= 3;
        } else {
            c[p] = c[i];
```

```
54                 --p;
55             }
56         }
57     }
58
59     int main() {
60         const int len = 100;
61         char s[len] = "I am beautiful!";
62         cout << replace1(s) << endl;
63         //replace2(s);
64         //cout << s << endl;
65         return 0;
66     }
```

1.6 Given an image represented by an NxN matrix, where each pixel in the image is 4 bytes, write a method to rotate the image by 90 degrees. Can you do this in place?

```
1  #include <iostream>
2  using namespace std;
3
4  void swap(int &a, int &b) {
5      int tmp = a;
6      a = b;
7      b = tmp;
8  }
9
10 void transpose(int a[][4], int n) {
11     for (int i = 0; i < n; ++i )
12         for (int j = i+1; j < n; ++j )
13             swap(a[i][j], a[j][i]);
14     for (int i = 0; i < n/2; ++i )
15         for (int j = 0; j < n; ++j )
16             swap(a[i][j], a[n-1-i][j]);
17 }
18
19 int main() {
20     int a[4][4] = {
21         {1, 2, 3, 4},
22         {5, 6, 7, 8},
23         {9, 10, 11, 12},
24         {13, 14, 15, 16}
25     };
26     for (int i = 0; i < 4; ++i ) {
27         for (int j = 0; j < 4; ++j ) {
28             cout << a[i][j] << '\t';
29         }
30         cout << endl;
31     }
32     cout << endl;
33     transpose(a, 4);
34     for (int i = 0; i < 4; ++i ) {
35         for (int j = 0; j < 4; ++j ) {
36             cout << a[i][j] << '\t';
37         }
38         cout << endl;
```

```
39      }
40      return 0;
41  }
```

1.7 Write an algorithm such that if an element in an MxN matrix is 0, its entire row and column is set to 0.

```
1   #include <iostream>
2   #include <cstring>
3   #include <cstdio>
4   using namespace std;
5
6   void zero(int **a, int m, int n) {
7       bool row[m], col[n];
8       memset(row, 0, sizeof(row));    // seems to have problem LeetCode reference
9       memset(col, 0, sizeof(col));
10      for (int i = 0; i < m; ++i )
11          for (int j = 0; j < n; ++j )
12              if (a[i][j] == 0) {
13                  row[i] = true;
14                  col[j] = true;
15              }
16      for (int i = 0; i < m; ++i ) {
17          for (int j = 0; j < n; ++j ) {
18              if (row[i] || col[j])
19                  a[i][j] = 0;
20          }
21      }
22  }
23
24  int main() {
25      freopen("sev.txt", "r", stdin);
26
27      int m, n;
28      cin >> m >> n;
29      int **a;
30      for (int i = 0; i < m; ++i )
31          a[i] = new int[n];
32      for (int i = 0; i < m; ++i ){
33          for (int j = 0; j < n; ++j ) {
34              cin >> a[i][j];
35          }
36      }
37
38      for (int i = 0; i < m; ++i ) {
39          for (int j = 0; j < n; ++j ) {
40              cout << a[i][j] << '\t';
41          }
42          cout << endl;
43      }
44      cout << endl;
45
46      zero(a, m, n);
47      for (int i = 0; i < m; ++i ) {
48          for (int j = 0; j < n; ++j ) {
49              cout << a[i][j] << '\t';
```

```
50          }
51          cout << endl;
52      }
53
54      fclose(stdin);
55      return 0;
56  }
```

1.8 Assume you have a method isSubstring which checks if one word is a substring of another. Given two strings, s1 and s2, write code to check if s2 is a rotation of s1 using only one call to isSubstring (i.e., "waterbottle" is a rotation of "erbottlewat").

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  bool isSubstring(string s1, string s2) {
6      if ( s1.find(s2) != string::npos)  // the part that I am not that familiar!!!
7          return true;
8      else return false;
9  }
10
11 bool isRotation(string s1, string s2) {
12     if ( (s1.length() != s2.length()) || s1.length() <= 0 )
13         return false;
14     return isSubstring(s1+s1, s2);
15 }
16
17 int main() {
18     string s1 = "waterbottle";
19     string s2 = "erbottlewat";
20     cout << isRotation(s1, s2) << endl;
21     return 0;
22 }
```

Chapter 2

Linked Lists

2.1 Write code to remove duplicates from an unsorted linked list.

FOLLOW UP

How would you solve this problem if a temporary buffer is not allowed?

从一个未排序的链表中移除重复的项. 进一步地，如果不允许使用临时的缓存，你如何解决这个问题?

Solution:

如果可以使用额外的存储空间，我们就开一个数组来保存一个元素的出现情况。 对于这种情况，最好的解决方法当然是使用哈希表，但令人非常不爽的是C++标准里是没有 哈希表的(java里有)。网上有人用ext下的hash_map，但毕竟不是C++标准里的， 用起来怪怪的，搞不好换个环境就跑不起来了(像Linux和Windows下使用就不一样)。 所以，一般用一个数组模拟一下就好了。但，这里要注意一个问题， 就是元素的边界，比如链表里存储的是int型变量。那么，如果有负值，这个方法就不奏效 了。而如果元素里的最大值非常大，那么这个数组也要开得很大，而数组中大部分空间是用 不上的，会造成空间的大量浪费。

简言之，如果可以用哈希表，还是用哈希表靠谱。

如下代码遍历一遍链表即可，如果某个元素在数组里记录的是已经出现过， 那么将该元素删除。时间复杂度O(n):

```
1  void removedulicate(node *head){
2      if(head==NULL) return;
3      node *p=head, *q=head->next;
4      hash[head->data] = true;
5      while(q){
6          if(hash[q->data]){
7              node *t = q;
8              p->next = q->next;
9              q = p->next;
10             delete t;
11         }
12         else{
13             hash[q->data] = true;
14             p = q; q = q->next;
15         }
16     }
17 }
```

如果不允许使用临时的缓存(即不能使用额外的存储空间)，那需要两个指针， 当第一个指针指向某个元素时，第二个指针把该元素后面与它相同的元素删除， 时间复杂度O(n2)，代码如下:

```
1  void removedulicate1(node *head){
2      if(head==NULL) return;
3      node *p, *q, *c=head;
4      while(c){
5          p=c; q=c->next;
6          int d = c->data;
7          while(q){
```

```
8               if(q->data==d){
9                   node *t = q;
10                  p->next = q->next;
11                  q = p->next;
12                  delete t;
13              }
14              else{
15                  p = q; q = q->next;
16              }
17          }
18          c = c->next;
19      }
20  }
```

完整代码如下：

```
1   #include <iostream>
2   #include <cstring>
3   using namespace std;
4
5   typedef struct node {
6       int data;
7       node* next;
8   } node;
9
10  bool hash[100];
11
12  node* init(int a[], int n) {
13      node *head, *p;
14      for (int i = 0; i < n; ++i) {
15          node * nd = new node();
16          nd->data = a[i];
17          if (i == 0) {
18              head = p = nd; // set head
19              continue;
20          }
21          p->next = nd;
22          //p = nd;           // same as followed line
23          p = p->next;
24      }
25      return head;
26  }
27
28  void removeDuplicate1(node* head) {
29      if (head == NULL) return;
30
31      node* p = head, *q = head->next;
32      hash[head->data] = true;
33
34      while(q) {
35          if (hash[q->data]) {
36              node* tmp = q;
37              q = q->next;
38              p->next = q;
39              delete tmp;
40          } else {
41              hash[q->data] = true;
42              p = q;
43              q = q->next;
44          }
```

```
45      }
46  }
47
48  void removeDuplicate2(node* head) {
49      if (head == NULL) return;
50
51      node* p = head, *curr = head->next, *prev;
52      while(p) {
53          prev = p;
54          curr = p->next;
55          while(curr) {
56              if (curr->data == p->data) {
57                  node* tmp = curr;
58                  prev->next = curr->next;
59                  curr = curr->next;  // same as: curr = prev->next;
60                  delete tmp;
61              } else {
62                  prev = curr;
63                  curr = curr->next;
64              }
65          }
66          p = p->next;
67      }
68  }
69
70  void print(node* head) {
71      node * curr = head;
72      while (curr) {
73          cout << curr->data << ' ';
74          curr = curr->next;
75      }
76      cout << endl;
77  }
78
79  int main() {
80      int n = 10;
81      int a[] = {3, 2, 1, 3, 5, 6, 2, 6, 3, 1};
82      memset(hash, false, sizeof(hash));
83      node *head = init(a, 10);
84      print(head);
85      //removeDuplicate1(head);
86      removeDuplicate2(head);
87      print(head);
88      return 0;
89  }
```

2.2 Implement an algorithm to find the nth to last element of a singly linked list.

实现一个算法从一个单链表中返回倒数第n个元素。

Solution:

这道题的考点在于我们怎么在一个单链表中找到倒数第n个元素？ 由于是单链表，所以我们没办法从最后一个元素数起，然后数n个得到答案。 但这种最直观的思路显然是没错的，那我们有没有办法通过别的方式，从最后的元素数起数 n个来得到我们想要的答案呢。

这个次序颠倒的思路可以让我们联想到一种数据结构：栈。

我们如果遍历一遍单链表，将其中的元素压栈，然后再将元素一一出栈。那么， 第n个出栈的元素就是我们想要的。

那我们是不是需要显式地用栈这么一个结构来做这个问题呢？答案是否。看到栈，我们应当 要想到递归，它是一种天然使用栈的方式。所以，第一种解决方案用递归，让栈自己帮我 们从链表的最后一个元素数起。

思路很简单，如果指向链表的指针还未空，就不断递归。当指针为空时开始退递归，这个过 程n不断减1，直接等于1，即可把栈中当前的元素取出。代码如下：

```
node *pp;
int nn;
void findNthToLast1(node *head){
    if(head==NULL) return;
    findNthToLast1(head->next);
    if(nn==1) pp = head;
    --nn;
}
```

递归的特点就是理解直接，代码短小。所以，很多递归代码看起来都很漂亮(不包括我这个 哈，还用了两个全局变量，比较不美观)。除了递归，这道题目还有别的方法还做吗? 答案还是有。虽然我们没办法从单链表的最后一个元素往前数，但如果我们维护两个指针， 它们之间的距离为n。然后，我将这两个指针同步地在这个单链表上移动，保持它们的距离 为n不变。那么，当第二个指针指到空时，第一个指针即为所求。很tricky的方法， 将这个问题很漂亮地解决了。代码如下:

```
node* findNthToLast(node *head, int n){
    if(head==NULL || n < 1) return NULL;
    node *p=head, *q=head;
    while(n>0 && q){
        q = q->next;
        --n;
    }
    if(n > 0) return NULL;
    while(q){
        p = p->next;
        q = q->next;
    }
    return p;
}
```

没有额外的全局变量，代码看起来就要漂亮一些。:P
完整代码如下:

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

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

node* init(int a[], int n) {
    node *head, *p;
    for (int i = 0; i < n; ++i) {
        node * nd = new node();
        nd->data = a[i];
        if (i == 0) {
            head = p = nd; // set head
            continue;
        }
        p->next = nd;
        p = p->next;
    }
    return head;
}

// traverse the list twice, my method
int findNthToLast1(node* head, int n) {
    if (head == NULL) return 0;
    int m, cnt = 1;
    node *prev = head, *curr = prev->next;
```

```
29      while(curr) {
30          prev = curr;
31          curr = curr->next;
32          ++cnt;
33      }
34      m = cnt;
35      cnt = 0;
36      prev = head;
37      curr = prev->next;
38      if (curr==NULL && n == 1)
39          return head->data;
40
41      while(curr) {
42          if (cnt == m - n)
43              return prev->data;
44          else {
45              prev = curr;
46              curr = curr->next;
47              ++cnt;
48          }
49      }
50  }
51
52  node* pp;
53  int nn;
54
55  void findNthToLast2(node* head) {
56      if (head == NULL) return;
57      findNthToLast2(head->next);
58      if (nn==1)
59          pp = head;
60      --nn;
61  }
62
63  // two pointer with n nodes separated
64  node* findNthToLast3(node* head, int n) {
65      if ( (head == NULL) || n < 1) return NULL; // forgot to consider
66
67      node * prev = head, *curr = head;
68      while (n > 0 && curr) {
69          curr = curr->next;
70          --n;
71      }
72      if (n > 0) return NULL;  // forgot to consider
73
74      while(curr) {
75          prev = prev->next;
76          curr = curr->next;
77      }
78      return prev;
79  }
80
81  void print(node* head) {
82      node * curr = head;
83      while (curr) {
84          cout << curr->data << ' ';
85          curr = curr->next;
86      }
87      cout << endl;
```

```
88  }
89
90  int main() {
91      int n = 10;
92      int a[] = {3, 2, 1, 3, 5, 6, 2, 6, 3, 1};
93      node *head = init(a, 10);
94      print(head);
95
96      cout << findNthToLast1(head, 4) << endl;
97
98      node *p = findNthToLast3(head, 4);
99      if (p) cout << p->data << endl;
100     else cout << "the length of link is not long enough" << endl;
101
102     nn = 4;
103     findNthToLast2(head);
104     if (pp) cout << pp->data << endl;
105
106     return 0;
107 }
```

2.3 Implement an algorithm to delete a node in the middle of a single linked list, given only access to that node.

EXAMPLE

Input: the node 'c' from the linked list a->b->c->d->e

Result: nothing is returned, but the new linked list looks like a->b->d->e

实现一个算法来删除单链表中间的一个结点，只给出指向那个结点的指针。

例子:

输入：指向链表a->b->c->d->e中结点c的指针

结果：不需要返回什么，得到一个新链表：a->b->d->e

Solution:

这个问题的关键是你只有一个指向要删除结点的指针，如果直接删除它，这条链表就断了。 但你又没办法得到该结点之前结点的指针，是的，它连头结点也不提供。在这种情况下， 你只能另觅他径。重新审视一下这个问题，我们只能获得从c结点开始后的指针， 如果让你删除c结点后的某个结点，那肯定是没问题的。比如删除结点d，那只需要把c 的next指针指向e，然后delete d就ok了。好的，如果我们就删除结点d，我们将得到 a->b->c->e，和目标链表只差了一个结点。怎么办呢? 把d的数据给c！ 结点结构都是一样的，删除谁都一样，最关键的是结点中的数据，只要我们留下的是数据 a->b->d->e就OK了。

思路已经有了，直接写代码? 等等，写代码之前，让我们再简单分析一 下可能出现的各种情况(当然包括边界情况)。如果c指向链表的：1.头结点；2.中间结点。 3.尾结点。4.为空。情况1，2属于正常情况，直接将d的数据给c，c的next指针指向d 的next指向所指结点，删除d就OK。情况4为空，直接返回。情况3比较特殊，如果c 指向尾结点，一般会认为直接删除c就ok了，反正c后面也没有结点了，不用担心链表断开。 可是真的是这样吗? 代码告诉我们，直接删除c，指向c的那个结点(比如说b)的next指针 并不会为空。也就是说，如果你打印这个链表，它还是会打印出和原来链表长度一样的链表， 而且最后一个元素为0！

关于这一点，原书CTCI中是这么讲的，这正是面试官希望你指出来的。然后， 你可以做一些特殊处理，比如当c是尾结点时，将它的数据设置为一些特殊字符， 这样在打印时就可以不打印它。当然也可以直接不处理这种情况，原书里的代码就是这么做 的。这里，也直接不处理这种情况。代码如下:

```
1  bool remove(node *c){
2      if(c==NULL || c->next==NULL) return false;
3      node *q = c->next;
4      c->data = q->data;
5      c->next = q->next;
6      delete q;
7      return true;
8  }
```

```
1  #include <iostream>
2  using namespace std;
3
```

```
4  typedef struct node{
5      int data;
6      node *next;
7  }node;
8
9  node* init(int a[], int n){
10     node *head, *p;
11     for(int i=0; i<n; ++i){
12         node *nd = new node();
13         nd->data = a[i];
14         if(i==0){
15             head = p = nd;
16             continue;
17         }
18         p->next = nd;
19         p = p->next;
20     }
21     return head;
22 }
23
24 bool remove(node *c){
25     if (c==NULL || c->next == NULL) return false;
26     //if (c->next == NULL) { //c为最后一个元素时直接删除，不行，最后还是会打印出一个为0的结点，需要特殊处理
27     //     delete c;
28     //     return;
29     //}
30     node* tmp = c->next;
31     c->data = tmp->data;
32     c->next = tmp->next;
33     delete tmp;
34     return true;
35 }
36
37 void print(node *head){
38     while(head){
39         cout<<head->data<<" ";
40         head = head->next;
41     }
42     cout<<endl;
43 }
44
45 int main(){
46     int n = 10;
47     int a[] = {9, 2, 1, 3, 5, 6, 2, 6, 3, 1};
48     node *head = init(a, n);
49     int cc = 3;
50     node *c = head;
51     for(int i=1; i<cc; ++i) c = c->next;
52     print(head);
53     if(remove(c))
54         print(head);
55     else
56         cout<<"failure"<<endl;
57     return 0;
58 }
```

2.4 You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in reverse order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.

EXAMPLE

Input: (3 -> 1 -> 5) + (5 -> 9 -> 2)

Output: 8 -> 0 -> 8

你有两个由单链表表示的数。每个结点代表其中的一位数字。数字的存储是逆序的， 也就是说个位位于链表的表头。写一函数使这两个数相加并返回结果，结果也由链表表示。

例子：(3 -> 1 -> 5), (5 -> 9 -> 2)

输入：8 -> 0 -> 8

Solution: 这道题目并不难，需要注意的有：1.链表为空。2.有进位。3.链表长度不一样。 代码如下：

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

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

node* init(int a[], int n){
    node *head, *p;
    for(int i=0; i<n; ++i){
        node *nd = new node();
        nd->data = a[i];
        if(i==0){
            head = p = nd;
            continue;
        }
        p->next = nd;
        p = p->next;
    }
    return head;
}

// single list, return single list
node* sumAdd(node* head) {
    if (head == NULL || head->next == NULL) return NULL;

    node *prev = head, *curr = head->next;
    int a = prev->data, cnt = 0;
    while(curr->next) {
        prev = prev->next;
        curr = curr->next->next;
        ++cnt;
        a += prev->data * pow(10, cnt);
    }

    curr = prev->next;
    int b = curr->data;
    cnt = 0;
    while(curr->next) {
        curr = curr->next;
        ++cnt;
        b += curr->data * pow(10, cnt);
    }

    int result = a + b;
    if (result >= 0) {
        node* newhead = new node();
        newhead->data = result % 10;
```

```
51          newhead->next = NULL;
52
53          result = result / 10;
54          node * p = newhead;
55
56          while (result > 0) {
57              node* curr = new node();
58              curr->data = result % 10;
59              curr->next = NULL;
60              p->next = curr;
61              p = p->next;
62              result = result / 10;
63          }
64          return newhead;
65      } else
66          return NULL;
67  }
68
69  // double lists, return a list
70  node* addlink(node *p, node *q){
71      if (p == NULL) return q;
72      if (q == NULL) return p;
73
74      int result = 0;
75      node *curr = NULL, *newhead;
76      while (p && q) {
77          result += p->data + q->data;
78          node* tmp = new node();
79          tmp->data = result % 10;
80          tmp->next = NULL;
81          if (curr) {
82              curr->next = tmp;
83              curr = curr->next;
84          } else
85              newhead = curr = tmp;
86          p = p->next;
87          q = q->next;
88          result = result / 10;
89      }
90
91      node* r;
92      if (p) r = p;
93      else r = q;
94      while (r) {
95          result += r->data;
96          node* tmp = new node();
97          tmp->data = result % 10;
98          tmp->next = NULL;
99          curr->next = tmp;
100         curr = curr->next;
101         r = r->next;
102         result = result / 10;
103     }
104
105     if (result > 0) { //还有进位时
106         node* tmp = new node();
107         tmp->data = result;
108         tmp->next = NULL;
109         curr->next = tmp;
```

```
110         }
111         return newhead;
112     }
113
114     void print(node *head){
115         while(head){
116             cout<<head->data<<" ";
117             head = head->next;
118         }
119         cout<<endl;
120     }
121
122     int main(){
123         int n = 6;
124         int a[] = {3, 1, 4, 5, 9, 2};
125         node *head = init(a, n);
126         node* curr = sumAdd(head);
127         if (curr) print(curr);
128         else cout << "fail!!!" << endl;
129         cout << endl;
130
131         int n2 = 4;
132         int a2[] = {1, 2, 9, 3};
133         int m = 3;
134         int b2[] = {9, 9, 2};
135
136         node *p = init(a2, n2);
137         node *q = init(b2, m);
138         node *res = addlink(p, q);
139         if(p) print(p);
140         if(q) print(q);
141         if(res) print(res);
142
143         return 0;
144     }
```

2.5 Given a circular linked list, implement an algorithm which returns node at the beginning of the loop.

DEFINITION

Circular linked list: A (corrupt) linked list in which a node's next pointer points to an earlier node, so as to make a loop in the linked list.

EXAMPLE

input: A -> B -> C -> D -> E -> C [the same C as earlier]

output: C

给定一个循环链表，实现一个算法返回这个环的开始结点。

定义：

循环链表：链表中一个结点的指针指向先前已经出现的结点，导致链表中出现环。

例子：

输入：A -> B -> C -> D -> E -> C [结点C在之前已经出现过]

输出：结点C

Solution:

http://hawstein.com/posts/2.5.html

关于带环链表的题目，《编程之美》中也有讲过。方法很tricky，设置快慢指针，(快指针速度为2，慢指针速度为1)使它们沿着链表移动，如果这个链表中存在环，那么快指针最终会追上慢指针而指向同一个结点。接下来的问题是，快指针追上慢指针后，怎么找到这个环的开始结点？现在我们还没有答案，那让我们先来分析一下，快指针会在哪里追上慢指针。

设环的开始结点(图中的D)前有k个结点，环有n个结点(上图中n从D到K共8个结点)。快指针fast和慢指针slow一开始都指向头结点head，它们移动k步可到环的开始结点。假设慢指针走过m个结点后，快指针追上了它，这时快指针走过了2m个结点。快指针比慢指针多走过的结点都在环里转圈了，是环中结点数n的整数倍，即：

2m - m = pn -> m = pn, p为正整数

如果头结点是第一个结点的话，那么相遇结点就是第m+1=pn+1个结点(慢指针走了m步)。 当环开始结点为起点1时，相遇结点为第pn+1-k个结点(减去前k个结点)。这pn+1-k 有可能是绕环转了很多圈后的一个数，假设继续走过一些结点，它就绕环刚好q圈， 则从相遇结点需要再经过 qn-(pn+1-k)+1=(q-p)n+k个结点，才能回到环的开始结点(图中结点D)。 由于从相遇结点走回到环的开始结点(图中D)所需要步数一定是小于等于一圈的，因此有

(q-p)n+k <= n

由于q，p，n，k都是正整数，因此有q-p <= 0，否则上式左边大于右边。 因为q-p <= 0，可以得出上式左边是小于等于k的。即：

(q-p)n+k <= k

如果让快指针在相遇结点继续走，不过这次把速度变成了慢指针一样， 那么它要走(q-p)n+k步到达环开始结点，让慢指针从头结点head开始走， 它要走k步到达环开始结点。最后，它们将在环开始结点处相遇。

这个是怎么得出来的呢? 假设快指针走了(q-p)n+k个结点到达环的开始结点，这时， 慢指针也走了(q-p)n+k步，它离环的开始结点还有

k - [(q-p)n + k] = (p-q)n (步)

从上面我们知道q-p <= 0，因此p-q >= 0。 这说明慢指针离环开始结点的步数正好是环中结点数的整数倍， 所以当慢指针到达环的开始结点时， 快指针(此时它的速度也是1)刚好在环中转了(p-q)圈，然后和慢指针在环的开始结点处相遇。

代码如下：

```
node* loopstart(node *head){
    if(head==NULL) return NULL;
    node *fast = head, *slow = head;
    while(fast && fast->next){
        fast = fast->next->next;
        slow = slow->next;
        if(fast==slow) break;
    }
    if(!fast || !fast->next) return NULL;
    slow = head;
    while(fast!=slow){
        fast = fast->next;
        slow = slow->next;
    }
    return fast;
}
```

这个思路确实很巧很tricky。但，还有没有别的方法呢? 更直观更简单的方法。 既然这么问了，当然是有了。:p一个无环的链表，它每个结点的地址都是不一样的。 但如果有环，指针沿着链表移动，那这个指针最终会指向一个已经出现过的地址。 答案是不是已经呼之欲出了。嗯，没错，哈希表!

以地址为哈希表的键值，每出现一个地址，就将该键值对应的实值置为true。 那么当某个键值对应的实值已经为true时，说明这个地址之前已经出现过了， 直接返回它就OK了。

由于C++标准中没有哈希表的操作，我用map进行模拟。不过哈希表的插入和取值操作是O(1) 的时间。而map是由一个RB tree组织，为了维护这个RB tree，插入和取值都会花更多的时 间。

代码如下：

```
map<node*, bool> hash;
node* loopstart1(node *head){
    while(head){
        if(hash[head]) return head;
        else{
            hash[head] = true;
            head = head->next;
        }
    }
    return head;
}
```

完整代码如下页：

Chapter 3

Stacks and Queues

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

你如何只用一个数组实现三个栈?

Solution:

我们可以很容易地用一个数组来实现一个栈，压栈就往数组里插入值，栈顶指针加1； 出栈就直接将栈顶指针减1；取栈顶值就把栈顶指针指向的单元的值返回； 判断是否为空就直接看栈顶指针是否为-1。

如果要在一个数组里实现3个栈，可以将该数组分为3个部分。如果我们并不知道哪个栈将装 入更多的数据，就直接将这个数组平均分为3个部分，每个部分维护一个栈顶指针， 根据具体是对哪个栈进行操作，用栈顶指针去加上相应的偏移量即可。

代码如下:

```
1  class stack3{
2  private:
3      int *buf;
4      int ptop[3];
5      int size;
6  public:
7      stack3(int size = 300){
8          buf = new int[size*3];
9          ptop[0]=ptop[1]=ptop[2]=-1;
10         this->size = size;
11     }
12     ~stack3(){
13         delete[] buf;
14     }
15
16     void push(int stackNum, int val){
17         int idx = stackNum*size + ptop[stackNum] + 1;
18         buf[idx] = val;
19         ++ptop[stackNum];
20     }
21     void pop(int stackNum){
22         --ptop[stackNum];
23     }
24     int top(int stackNum){
25         int idx = stackNum*size + ptop[stackNum];
26         return buf[idx];
27     }
28     bool empty(int stackNum){
29         return ptop[stackNum]==-1;
30     }
31 };
```

当然，我们也可以有第二种方案。数组不分段，无论是哪个栈入栈，都依次地往这个数组里 存放。这样一来，我们除了维护每个栈的栈顶指针外，我们还需要维护每个栈中， 每个元素指向前一个元素的指针。这样一来，某个栈栈顶元素出栈后，它就能正确地找到下 一个栈顶元素。

所以，数组里存放的不再是基本数据类型的数据，我们需要先定义一个结点结构：

```
typedef struct node{
    int val;
    int preIdx;
} node;
```

数组中的每个元素将是这样一个结点，它保存当前位置的值，和指向上一个结点的索引。

具体代码如下：

```
class stack3_1{
private:
    node *buf;
    int ptop[3];
    int totalSize;
    int cur;
public:
    stack3_1(int totalSize = 900){
        buf = new node[totalSize];
        ptop[0]=ptop[1]=ptop[2]=-1;
        this->totalSize = totalSize;
        cur = 0;
    }
    ~stack3_1(){
        delete[] buf;
    }

    void push(int stackNum, int val){
        buf[cur].val = val;
        buf[cur].preIdx = ptop[stackNum];
        ptop[stackNum] = cur;
        ++cur;
    }
    void pop(int stackNum){
        ptop[stackNum] = buf[ptop[stackNum]].preIdx;
    }
    int top(int stackNum){
        return buf[ptop[stackNum]].val;
    }
    bool empty(int stackNum){
        return ptop[stackNum]==-1;
    }
};
```

这种实现有一个缺点，在频繁地入栈出栈后，会造成数组空间的大量浪费。 因为当前指针cur是一直递增的，而堆栈在出栈后相应位置的空间将不会再被利用到。 代码中pop函数，只是修改了栈顶指针。如果我们对一个栈先执行出栈，再入栈， 那么再次入栈的位置是在cur，而不是原来栈顶的位置。

有没有避免这种空间浪费的方法呢？当然是有的。不过这样一来，对cur的操作就变得复杂 了。第一，每次执行pop操作，检查该元素对应索引是否小于cur，如果是， 将cur更新到该元素索引；否则cur不变。第二，每次执行完push操作， cur要沿着数组依次向后查找，直到找到第一个空的空间，用它的索引更新cur。 这部分代码实现起来也是没什么难度的，这里不再贴出。

以下是完整代码：

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

class stack3 {
private:
```

```
6       int *buf;
7       int ptop[3];
8       int size;
9   public:
10      stack3(int size = 300) {
11          buf = new int[size*3];
12          ptop[0] = ptop[1] = ptop[2] = -1;
13          this->size = size;
14      }
15      ~stack3() {
16          delete[] buf;
17      }
18
19      void push(int stackNum, int val) {
20          int idx = stackNum*size + ptop[stackNum] + 1;
21          buf[idx] = val;
22          ++ptop[stackNum];
23      }
24      void pop(int stackNum) {
25          --ptop[stackNum];
26      }
27      int top(int stackNum) {
28          int idx = stackNum*size + ptop[stackNum];
29          return buf[idx];
30      }
31      bool empty(int stackNum) {
32          return ptop[stackNum] == -1;
33      }
34  };
35
36  typedef struct node {
37      int val;
38      int preIdx;
39  } node;
40
41  class stack3_1{
42  private:
43      node* buf;
44      int ptop[3];
45      int totalSize;
46      int curr;
47  public:
48      stack3_1(int totalSize = 900) {
49          buf = new node[totalSize];
50          ptop[0] = ptop[1] = ptop[2] = -1;
51          this->totalSize = totalSize;
52          curr = 0;
53      }
54      ~stack3_1() {
55          delete[] buf;
56      }
57
58      void push(int stackNum, int val) {
59          buf[curr].val = val;
60          buf[curr].preIdx = ptop[stackNum];
61          ptop[stackNum] = curr;
62          ++curr;
63      }
64      void pop(int stackNum) {
```

```
65          ptop[stackNum] = buf[ptop[stackNum]].preIdx;
66      }
67      int top(int stackNum) {
68          return buf[ptop[stackNum]].val;
69      }
70      bool empty(int stackNum) {
71          return ptop[stackNum] == -1;
72      }
73  };
74
75  int main(){
76
77      //stack3 mystack;
78      stack3_1 mystack;
79      for(int i=0; i<10; ++i)
80          mystack.push(0, i);
81      for(int i=10; i<20; ++i)
82          mystack.push(1, i);
83      for(int i=100; i<110; ++i)
84          mystack.push(2, i);
85      for(int i=0; i<3; ++i)
86          cout<<mystack.top(i)<<" ";
87      cout<<endl;
88
89      for(int i=0; i<3; ++i){
90          mystack.pop(i);
91          cout<<mystack.top(i)<<" ";
92      }
93      cout<<endl;
94
95      mystack.push(0, 111);
96      mystack.push(1, 222);
97      mystack.push(2, 333);
98      for(int i=0; i<3; ++i)
99          cout<<mystack.top(i)<<" ";
100     cout<<endl;
101
102     return 0;
103 }
```

3.2 How would you design a stack which, in addition to push and pop, also has a function min which returns the minimum element? Push, pop and min should all operate in O(1) time.

实现一个栈，除了push和pop操作，还要实现min函数以返回栈中的最小值。 push，pop和min函数的时间复杂度都为O(1)。

Solution:

看到这个题目最直接的反应是用一个变量来保存当前栈的最小值，让我们来看看这样可行否？ 如果栈一直push那是没有问题，入栈元素如果比当前最小值还小，那就更新当前最小值。 可是如果pop掉的栈顶元素就是最小值，那么我们如何更新最小值呢？显然不太好办。 既然只用一个变量没法解决这个问题，那我们就增加变量。如果说每个结点除了保存当前的 值，另外再保存一个从该结点到栈底的结点中的最小值。那么，不论哪个结点成为了栈顶结 点，我们都有办法取得剩下的这些元素的最小值。代价是付出的空间多了一倍。

代码如下：

```
1  const int MAX_INT = ~(1<<31);//2147483647
2
3  typedef struct node{
4      int val, min;
5  }node;
6
7  class StackWithMin{
```

```
public:
    StackWithMin(int size=1000){
        buf = new node[size];
        buf[0].min = MAX_INT;
        cur = 0;
    }
    ~StackWithMin(){
        delete[] buf;
    }
    void push(int val){
        buf[++cur].val = val;
        if(val<buf[cur-1].min) buf[cur].min = val;
        else buf[cur].min = buf[cur-1].min;
    }
    void pop(){
        --cur;
    }
    int top(){
        return buf[cur].val;
    }
    bool empty(){
        return cur==0;
    }
    int min(){
        return buf[cur].min;
    }

private:
    node *buf;
    int cur;
};
```

这种实现方式有一个明显的问题：数据冗余。比如说，栈里的数据从栈底到栈顶是1到10000， 那么，每个结点保存的最小值都是1，也就是保存了1的10000份拷贝，有这个必要吗？ 直觉告诉我们应该是没必要的，我们应该可以想办法只保留一份1的拷贝， 然后如果结点是这些1到10000，就正确地返回这个最小值。这个要怎么做呢？ 我们假设除了用一个栈s1来保存数据，还用另一个栈s2来保存这些非冗余最小值。那么， 当我们将数据压到要s1时，同时将它和s2的栈顶元素比较，如果不大于s2的栈顶元素， 那么将当前值也压入s2中。这样一来，s2中保存的就是一个阶段性最小值。 即s2中的每个值都是s1中栈底到达某个位置的最小值。那么，如果执行pop操作呢？ 执行pop操作除了将s1中的栈顶元素出栈，还要将它和s2中的栈顶元素比较，如果相等， 说明这个值是栈底到栈顶的最小值，而它出栈后，最小值就不再是它了。所以， s2也要将栈顶元素出栈，新的栈顶元素将对应s1剩下元素中新的最小值。

代码如下：

```
class stack{
public:
    stack(int size=1000){
        buf = new int[size];
        cur = -1;
    }
    ~stack(){
        delete[] buf;
    }
    void push(int val){
        buf[++cur] = val;
    }
    void pop(){
        --cur;
    }
    int top(){
        return buf[cur];
    }
```

```
19      bool empty(){
20          return cur==-1;
21      }
22
23  private:
24      int *buf;
25      int cur;
26  };
27
28  class StackWithMin1{
29  public:
30      StackWithMin1(){
31
32      }
33      ~StackWithMin1(){
34
35      }
36      void push(int val){
37          s1.push(val);
38          if(val<=min())
39              s2.push(val);
40      }
41      void pop(){
42          if(s1.top()==min())
43              s2.pop();
44          s1.pop();
45      }
46      int top(){
47          return s1.top();
48      }
49      bool empty(){
50          return s1.empty();
51      }
52      int min(){
53          if(s2.empty()) return MAX_INT;
54          else return s2.top();
55      }
56
57  private:
58      stack s1, s2;
59  };
```

完整代码如下：

```
1   #include <iostream>
2   using namespace std;
3
4   const int MAX_INT = ~(1<<31); // 2147483647
5
6   typedef struct node {
7       int val;
8       int min;
9   } node;
10
11  class StackWithMin{
12  private:
13      node* buf;
14      int curr;
15  public:
16      StackWithMin(int size = 1000){
```

```
17          buf = new node[size];
18          buf[0].min = MAX_INT;
19          curr = 0;
20      }
21      ~StackWithMin(){
22          delete[] buf;
23      }
24
25      void push(int val) {
26          buf[++curr].val = val;
27          if ( val < buf[curr-1].min )
28              buf[curr].min = val;
29          else
30              buf[curr].min = buf[curr-1].min;
31      }
32      void pop() {
33          --curr;
34      }
35      int top() {
36          return buf[curr].val;
37      }
38      bool empty() {
39          //return (buf[curr].min == MAX_INT);
40          return curr == 0;  // much more easier
41      }
42      int min() {
43          return buf[curr].min;
44      }
45  };
46
47
48  // maintain two stack to save space
49  class stack {
50   private:
51      int *buf;
52      int curr;
53   public:
54      stack(int size = 1000) {
55          buf = new int[size];
56          curr = -1;
57      }
58      ~stack() {
59          delete[] buf;
60      }
61
62      void push(int val) {
63          buf[++curr] = val;
64      }
65      void pop() {
66          --curr;
67      }
68      int top() {
69          return buf[curr];
70      }
71      bool empty() {
72          return curr == -1;
73      }
74  };
75
```

```
76  class StackWithMin1{
77  private:
78      stack s1, s2;
79  public:
80      StackWithMin1(){
81      }
82      ~StackWithMin1(){
83      }
84
85      void push(int val) {
86          s1.push(val);
87          // if ( val <= s2.top() )
88          if ( val <= min() )  // more direct
89              s2.push(val);
90      }
91      void pop() {
92          if ( s1.top() == min() )
93              s2.pop();
94          int val = s1.top();
95          s1.pop();
96      }
97      int top() {
98          return s1.top();
99      }
100     bool empty() {
101         return s1.empty();
102     }
103     int min() {
104         if ( s2.empty() )
105             return MAX_INT;
106         else
107             return s2.top();
108     }
109 };
110
111
112 int main() {
113
114     StackWithMin1 mystack;
115
116     for(int i=0; i<20; ++i)
117         mystack.push(i);
118     cout << mystack.min() << " " << mystack.top() << endl;
119     mystack.push(-50);
120     mystack.push(-100);
121     cout << mystack.min() << " " << mystack.top() << endl;
122     mystack.pop();
123     cout << mystack.min() << " " << mystack.top() << endl;
124
125     return 0;
126 }
```

3.3 Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure SetOfStacks that mimics this. SetOfStacks should be composed of several stacks, and should create a new stack once the previous one exceeds capacity. SetOfStacks.push() and SetOfStacks.pop() should behave identically to a single stack (that is, pop() should return the same values as it would if there were just a single stack).

FOLLOW UP

Implement a function popAt(int index) which performs a pop operation on a specific sub-stack.

栈就像叠盘子，当盘子叠得太高时，就会倾斜倒下。因此，在真实的世界中，当一叠盘子 (栈)超过了一定的高度时，我们就会另起一堆，再从头叠起。实现数据结构SetOfStacks 来模拟这种情况。SetOfStacks由几个栈组成，当前一栈超出容量时，需要创建一个新的栈 来存放数据。SetOfStacks.push()和SetOfStacks.pop()的行为应当和只有一个栈时 表现的一样。

进一步地，

实现函数popAt(int index)在指定的子栈上进行pop操作。

Solution:

首先，我们如果不考虑popAt这个麻烦的函数，那么SetOfStacks的实现就简单很多。 SetOfStacks由栈的数组构成，我们需要一个指向当前栈的变量cur， 每当执行push操作时，我们需要检查一下当前栈是否已经达到其容量了， 如果是的话，就要将cur加1，指向下一个栈。而执行pop操作时， 需要先检查当前栈是否为空，如果是，则cur减1，移向上一个栈。top操作同理。 这时候，SetOfStacks可以想象成把一个本来可以叠得很高的栈，分成了好几个子栈。push和pop操作其实都只是在“最后”一个子栈上操作。

代码如下：

```
class SetOfStacks{//without popAt()
private:
    stack *st;
    int cur;
    int capacity;

public:
    SetOfStacks(int capa=STACK_NUM){
        st = new stack[capa];
        cur = 0;
        capacity = capa;
    }
    ~SetOfStacks(){
        delete[] st;
    }
    void push(int val){
        if(st[cur].full()) ++cur;
        st[cur].push(val);
    }
    void pop(){
        if(st[cur].empty()) --cur;
        st[cur].pop();
    }
    int top(){
        if(st[cur].empty()) --cur;
        return st[cur].top();
    }
    bool empty(){
        if(cur==0) return st[0].empty();
        else return false;
    }
    bool full(){
        if(cur==capacity-1) return st[cur].full();
        else return false;
    }
};
```

当加入popAt函数，情况就变得复杂了。因为这时候的数据分布可能出现中间的某些子栈使 用popAt把它们清空了，而后面的子栈却有数据。为了实现方便，我们不考虑因为popAt 带来的空间浪费。即如果我用popAt把中间某些子栈清空了，并不把后面子栈的数据往前移 动。这样一来，cur指向操作的“最后”一个栈，它后面的子栈一定都是空的， 而它本身及前面的子栈由于popAt函数的缘故都有可能是空的。如果没有popAt函数， cur前面的子栈一定都是满的(见上面的例子)。这样一来，push仍然只需要判断一次当前子 栈是否为满。但是，pop函数则要从cur向前一直寻找，直到找到一个非空的子栈， 才能进行pop操作。同理，popAt，top，empty也是一样的。

代码如下：

```
class SetOfStacks1{
```

```
private:
    stack *st;
    int cur;
    int capacity;

public:
    SetOfStacks1(int capa=STACK_NUM){
        st = new stack[capa];
        cur = 0;
        capacity = capa;
    }
    ~SetOfStacks1(){
        delete[] st;
    }
    void push(int val){
        if(st[cur].full()) ++cur;
        st[cur].push(val);
    }
    void pop(){
        while(st[cur].empty()) --cur;
        st[cur].pop();
    }
    void popAt(int idx){
        while(st[idx].empty()) --idx;
        st[idx].pop();
    }
    int top(){
        while(st[cur].empty()) --cur;
        return st[cur].top();
    }
    bool empty(){
        while(cur!=-1 && st[cur].empty()) --cur;
        if(cur==-1) return true;
        else return false;
    }
    bool full(){
        if(cur==capacity-1) return st[cur].full();
        else return false;
    }
};
```

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

const int STACK_SIZE = 100;
const int STACK_NUM = 10;

class stack {
 private:
    int *buf;
    int curr;
    int capacity;
 public:
    stack(int capa = STACK_SIZE) {
        buf = new int[capa];
        curr = -1;
        capacity = capa;
    }
    ~stack() {
```

```
19          delete[] buf;
20      }
21
22      void push(int val) {
23          buf[++curr] = val;
24      }
25      void pop() {
26          --curr;
27      }
28      int top() {
29          return buf[curr];
30      }
31      bool empty() {
32          return curr == -1;
33      }
34      bool full() {
35          return curr == capacity-1;
36      }
37  };
38
39  class SetOfStacks {  // without popAt() yet
40  private:
41      stack *st;
42      int curr;
43      int capacity;
44  public:
45      SetOfStacks(int capa = STACK_NUM) {
46          st = new stack[capa];
47          curr = 0;
48          capacity = capa;
49      }
50      ~SetOfStacks() {
51          delete[] st;
52      }
53
54      void push(int val) {
55          if ( st[curr].full() )
56              ++curr;
57          st[curr].push(val);
58      }
59      void pop() {
60          if ( st[curr].empty() )
61              --curr;
62          st[curr].pop();
63      }
64      int top() {
65          if ( st[curr].empty() )
66              --curr;
67          return st[curr].top();
68      }
69      bool empty() {
70          if ( curr == 0 )
71              return  st[curr].empty() ;
72          else
73              return false;
74      }
75      bool full() {
76          if ( curr == capacity-1 )
77              return st[curr].full();
```

```
78          else
79              return false;
80      }
81  };
82
83
84  // with popAt() function
85  class SetOfStacks1 {
86  private:
87      stack *st;
88      int curr;
89      int capacity;
90  public:
91      SetOfStacks1(int capa = STACK_NUM) {
92          st = new stack[capa];
93          curr = 0;
94          capacity = capa;
95      }
96      ~SetOfStacks1() {
97          delete[] st;
98      }
99
100     void push(int val) {
101         if ( st[curr].full() )
102             ++curr;
103         st[curr].push(val);
104     }
105     void pop() {
106         while ( st[curr].empty() && curr > 0 ) // consider first stack empty condition
107             --curr;
108         st[curr].pop();
109     }
110     void popAt(int val) {
111         if ( st[val].empty() && val > 0 )
112             --val;
113         st[val].pop();
114         if ( curr == val && st[val].empty() ) // I think I need to consider this condition
115             --curr;
116     }
117     int top() {
118         while ( st[curr].empty() && curr > 0 )
119             --curr;
120         return st[curr].top();
121     }
122     bool empty() {
123         //while ( st[curr].empty() && curr > 0 ) --curr;
124         //if ( curr == 0 ) return  st[curr].empty() ;
125
126         while ( curr != -1 && st[curr].empty() )
127             --curr;
128         if (curr == -1) return true;
129         else
130             return false;
131     }
132     bool full() {
133         if ( curr == capacity-1 )
134             return st[curr].full();
135         else
136             return false;
```

```
137         }
138     };
139
140
141     int main() {
142         SetOfStacks1 s1;
143         for (int i = 0; i < 3*STACK_SIZE+1; ++i )
144             s1.push(i);
145         for (int i = 0; i < STACK_SIZE; ++i ) {
146             s1.popAt(0);
147             s1.popAt(2);
148         }
149         s1.popAt(3);
150         while ( !s1.empty() ) {
151             cout << s1.top() << endl;
152             s1.pop();
153         }
154         return 0;
155     }
```

3.4 In the classic problem of the Towers of Hanoi, you have 3 rods and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (e.g., each disk sits on top of an even larger one). You have the following constraints:

(A) Only one disk can be moved at a time.

(B) A disk is slid off the top of one rod onto the next rod.

(C) A disk can only be placed on top of a larger disk.

Write a program to move the disks from the first rod to the last using Stacks.

编程解决汉诺塔问题，使用数据结构栈(偷个懒，如果不知道汉诺塔是什么，请自行Google)

Solution:

汉诺塔是个非常经典的问题，讲递归时应该都会讲到它。如果我们没有递归的先验知识， 直接去解答这道题，常常会觉得不知道如何下手。用递归却可以非常优美地解决这个问题。

使用递归的一个关键就是，我们先定义一个函数，不用急着去实现它， 但要明确它的功能。

对于汉诺塔问题，我们定义如下函数原型：

void hanoi(int n, char src, char bri, char dst);

我们先不去关心它是怎么实现的，而是明确它的功能是：

- 将n个圆盘从柱子src移动到柱子dst，其中可以借助柱子bri(bridge)。
- 注：n个圆盘从上到下依次的标号依次为1到n，表示圆盘从小到大。
- 移动的过程中，不允许大圆盘放在小圆盘的上面。

OK，既然要用到递归，当然是在这个函数中还是用到这个函数本身， 也就是说，我们完成这个任务中的步骤还会用到hanoi这个操作，只是参数可能不一样了。 我们定义一组元组来表示三根柱子的状态：(src上的圆盘，bri上的圆盘，dst上的圆盘) 初始状态是：(1 n, 0, 0)表示src上有1到n共n个圆盘，另外两个柱子上没有圆盘。 目标状态是：(0, 0, 1 n)表示dst上有1到n共n个圆盘，另外两个柱子上没有圆盘。 由于最大的圆盘n最后是放在dst的最下面，且大圆盘是不能放在小圆盘上面的， 所以，一定存在这样一个中间状态：(n, 1 n-1, 0)，这样才能把最大的圆盘n 移动到dst的最下面。这时候，有人就会问，你怎么就想到这个中间状态而不是其它呢？ 好问题。因为，我现在手头上的工具(可用的函数)只有hanoi， 那我自然要想办法创造可以使用这个函数的情景，而不是其它情景。

- 初始状态是：(1 n, 0, 0)
- 中间状态是：(n, 1 n-1, 0)

从初始状态到中间状态使用操作hanoi(n-1, src, dst, bri)就可以到达了。即把n-1 个圆盘从src移动到bri，中间可以借助柱子dst。

接下来就是将圆盘n从src移动到dst了，这个可以直接输出：

cout«"Move disk "«n«" from "«src«" to "«dst«endl;

这个操作后得到的状态是：

(0, 1 n-1, n)

然后再利用hanoi函数，将n-1个圆盘从bri移动到dst，中间可借助柱子src， hanoi(n-1, bri, src, dst)，操作后得到最终状态：

(0, 0, 1 n)

这些操作合起来就三行代码：

- hanoi(n-1, src, dst, bri);
- cout«"Move disk "«n«" from "«src«" to "«dst«endl;
- hanoi(n-1, bri, src, dst);

最后，我们还需要递归停止条件。什么时候递归结束呢？当n等于1时，既只有一个圆盘时， 直接把它从src移动到dst即可：

if(n==1)

cout«"Move disk "«n«" from "«src«" to "«dst«endl;

所以，完整的汉诺塔问题的递归解法如下：

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

void hanoi(int n, char src, char bri, char dst){
    if(n==1){
        cout<<"Move disk "<<n<<" from "<<src<<" to "<<dst<<endl;
    }
    else{
        hanoi(n-1, src, dst, bri);
        cout<<"Move disk "<<n<<" from "<<src<<" to "<<dst<<endl;
        hanoi(n-1, bri, src, dst);
    }
}

int main(){
    int n = 3;
    hanoi(n, 'A', 'B', 'C');
    return 0;
}
```

汉诺塔的递归解法讲完了，可是这并不是题目要求的。题目要求用栈来解决这个问题。 递归解法其实也是用到了栈的，在每次递归调用自己的时候， 将中间的状态参数压入栈中。不过这些操作都是系统隐式进行的， 所以你不用去关心它具体是怎么压栈出栈的。如果我们要用栈自己来实现这个过程， 就不得不考虑这其中的细节了。

接下来，我们就显式地用栈来实现递归过程中，这些状态参数的压栈出栈过程。首先， 我们需要定义一个数据结构来保存操作过程中的参数。

```
struct op{
    int begin, end;
    char src, bri, dst;
    op(){
    }
    op(int pbegin, int pend, int psrc, int pbri, int pdst):begin(pbegin), end(pend), src(psrc),
    }
};
```

其中的5个参数表示，在柱子src上有一叠圆盘，标号从begin到end， 要将它们从src移动到dst，中间可借助柱子bri。end其实相当于递归解法中的n， src，bri，dst与递归解法中的对应。那为什么还要定义begin这个变量呢？为了判断柱子上是否只剩下一个盘子。如果begin等于end， 说明柱子上只剩下“最后”一个圆盘，可以进行移动。当然了， 用另外一个布尔变量来表示是否只剩下一个圆盘也是可以的，效果一样。 讲递归方法的时候，说到从初始状态到最终状态一共要经过以下几个状态：

- (1 n, 0, 0)
- (n, 1 n-1, 0)
- (0, 1 n-1, n)
- (0, 0, 1 n)

这些过程我们现在需要自己压栈出栈处理。压栈的时候不做处理，出栈时进行处理。因此， 压栈的时候需要与实际要操作的步骤相反。一开始，我们将最终想要完成的任务压栈。 听起来怪怪的，其实就是往栈中压入一组参数:

stack<op> st;

st.push(op(1, n, src, bri, dst));

这组参数表示，柱子src上有1 n个圆盘，要把它移动到dst上，可以借助柱子bri。 当栈st不为空时，不断地出栈，当begin和end不相等时，进行三个push操作 (对应上面四个状态，相邻状态对应一个push操作，使状态变化)，push与实际操作顺序相反(因为出栈时才进行处理，出栈时顺序就正确了)， 如果，begin与end相等，则剩下当前问题规模下的“最后”一个圆盘，直接打印移动方案， hanoi代码如下:

```
void hanoi(int n, char src, char bri, char dst){
    stack<op> st;
    op tmp;
    st.push(op(1, n, src, bri, dst));
    while(!st.empty()){
        tmp = st.top();
        st.pop();
        if(tmp.begin != tmp.end){
            st.push(op(tmp.begin, tmp.end-1, tmp.bri, tmp.src, tmp.dst));
            st.push(op(tmp.end, tmp.end, tmp.src, tmp.bri, tmp.dst));
            st.push(op(tmp.begin, tmp.end-1, tmp.src, tmp.dst, tmp.bri));
        }
        else{
            cout<<"Move disk "<<tmp.begin<<" from "<<tmp.src<<" to "<<tmp.dst<<endl;
        }

    }
}
```

完整代码如下:

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

void hanoi(int n, string src, string bri, string dst) {
    if (n == 1)
        cout << "Move disk " << n << " from " << src << " to " << dst << endl;
    else {
        hanoi(n-1, src, dst, bri);
        cout << "Move disk " << n << " from " << src << " to " << dst << endl;
        hanoi(n-1, bri, src, dst);
    }
}

int main() {
    int n = 3;
    hanoi(n, "src", "bri", "dst");
    return 0;
}
```

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

struct op {
    int begin, end;
    char src, bri, dst;
    op() {
    }
    op(int pbegin,int pend,int psrc,int pbri,int pdst):begin(pbegin),end(pend),src(psrc)
    }
```

```
};

void hanoi(int n, char src, char bri, char dst) {
    stack<op> st;
    op tmp;
    st.push( op(1, n, src, bri, dst) );
    while ( !st.empty() ) {
        tmp = st.top();
        st.pop();
        if (tmp.begin != tmp.end) {
            st.push( op(tmp.begin, tmp.end-1, tmp.bri, tmp.src, tmp.dst) );
            st.push( op(tmp.end, tmp.end, tmp.src, tmp.bri, tmp.dst) );
            st.push( op(tmp.begin, tmp.end-1, tmp.src, tmp.dst, tmp.bri) );
        } else {
            cout << "Move disk " << tmp.begin << " from " << tmp.src << " to " << tmp.dst << enc
        }
    }
}

int main() {
    int n = 3;
    hanoi(n, 'A', 'B', 'C');
    return 0;
}
```

3.5 Implement a MyQueue class which implements a queue using two stacks.

使用两个栈实现一个队列MyQueue。

Solution:

队列是先进先出的数据结构(FIFO)，栈是先进后出的数据结构(FILO)， 用两个栈来实现队列的最简单方式是：进入队列则往第一个栈压栈， 出队列则将第一个栈的数据依次压入第二个栈，然后出栈。每次有数据进入队列， 都先将第二个栈的数据压回第一个栈，然后再压入新增的那个数据； 每次有数据出队列，都将第一个栈的数据压入第二个栈，然后第二个栈出栈。 代码很简单：

```
template <typename T>
class MyQueue{
public:
    MyQueue(){

    }
    ~MyQueue(){

    }
    void push(T val){
        move(sout, sin);
        sin.push(val);
    }
    void pop(){
        move(sin, sout);
        sout.pop();
    }
    T front(){
        move(sin, sout);
        return sout.top();
    }
    T back(){
        move(sout, sin);
        return sin.top();
    }
    int size(){
        return sin.size()+sout.size();
```

```
28      }
29      bool empty(){
30          return sin.empty()&&sout.empty();
31      }
32      void move(stack<T> &src, stack<T> &dst){
33          while(!src.empty()){
34              dst.push(src.top());
35              src.pop();
36          }
37      }
38
39  private:
40      stack<T> sin, sout;
41  };
```

对于上面的实现，我们可以稍作改进来提高效率。上面的实现方法， 数据在两个栈之间移动得太频繁了，必然会导致效率下降。事实上有些移动是没有必要的。 有数据进入队列时，我们不去管第二个栈是否有数据，只管往第一个栈压栈即可。 当数据出队列时，如果第二个栈有数据，那就出栈。 因为此时第二个栈的栈顶元素即为队列的队首元素；如果第二个栈没有数据， 这才将第一个栈的数据出栈移动到第二个栈，然后第二个栈再出栈。这样一来， 逻辑上相当于将一个队列从中间切开，第一个栈从栈顶到栈底对应队列的队尾到切开处， 第二个栈从栈顶到栈底对应队列的队首到切开处。这样简单的修改， 可以减少许多不必要的数据移动，提高效率。

代码如下：

```
1   template <typename T>
2   class MyQueue1{
3   public:
4       MyQueue1(){
5
6       }
7       ~MyQueue1(){
8
9       }
10      void push(T val){
11          sin.push(val);
12      }
13      void pop(){
14          move(sin, sout);
15          sout.pop();
16      }
17      T front(){
18          move(sin, sout);
19          return sout.top();
20      }
21      T back(){
22          move(sout, sin);
23          return sin.top();
24      }
25      int size(){
26          return sin.size()+sout.size();
27      }
28      bool empty(){
29          return sin.empty()&&sout.empty();
30      }
31      void move(stack<T> &src, stack<T> &dst){
32          if(dst.empty()){
33              while(!src.empty()){
34                  dst.push(src.top());
35                  src.pop();
36              }
37          }
```

```
38      }
39
40  private:
41      stack<T> sin, sout;
42  };
```

完整代码如下：

```
1   #include <iostream>
2   #include <stack>
3   using namespace std;
4
5   template <typename T>
6   class MyQueue {
7   private:
8       stack<T> sin, sout;
9   public:
10      MyQueue() {
11      }
12      ~MyQueue() {
13      }
14
15      void push(T val) {
16          move(sout, sin);
17          sin.push(val);
18      }
19      void pop() {
20          move(sin, sout);
21          sout.pop();
22      }
23      T front() {
24          move(sin, sout);
25          sout.top();
26      }
27      T back() {
28          move(sout, sin);
29          sin.top();
30      }
31      int size() {
32          return sin.size() + sout.size();
33      }
34      bool empty() {
35          return sin.empty() && sout.empty();
36      }
37      void move(stack<T> &src, stack<T> &dst) {
38          while ( !src.empty() ) {
39              dst.push( src.top() );
40              src.pop();
41          }
42      }
43  };
44
45
46  // control move under different member functions
47  template <typename T>
48  class MyQueue1 {
49  private:
50      stack<T> sin, sout;
51  public:
52      MyQueue1() {
```

```
53      }
54      ~MyQueue1() {
55      }
56
57      void push(T val) {
58          sin.push(val);
59      }
60      void pop() {
61          if ( sout.empty() )
62              move(sin, sout);
63          sout.pop();
64      }
65      T front() {
66          if ( sout.empty() )
67              move(sin, sout);
68          sout.top();
69      }
70      T back() {
71          if ( sin.empty() )
72              move(sout, sin);
73          sin.top();
74      }
75      int size() {
76          return sin.size() + sout.size();
77      }
78      bool empty() {
79          return sin.empty() && sout.empty();
80      }
81      void move(stack<T> &src, stack<T> &dst) {
82          while ( !src.empty() ) {
83              dst.push( src.top() );
84              src.pop();
85          }
86      }
87  };
88
89
90  // control under move function only
91  template <typename T>
92  class MyQueue2 {
93  private:
94      stack<T> sin, sout;
95  public:
96      MyQueue2() {
97      }
98      ~MyQueue2() {
99      }
100
101     void push(T val) {
102         sin.push(val);
103     }
104     void pop() {
105         move(sin, sout);
106         sout.pop();
107     }
108     T front() {
109         move(sin, sout);
110         sout.top();
111     }
```

```
112     T back() {
113         move(sout, sin);
114         sin.top();
115     }
116     int size() {
117         return sin.size() + sout.size();
118     }
119     bool empty() {
120         return sin.empty() && sout.empty();
121     }
122     void move(stack<T> &src, stack<T> &dst) {
123         if ( dst.empty() ) {
124             while ( !src.empty() ) {
125                 dst.push( src.top() );
126                 src.pop();
127             }
128         }
129     }
130 };
131
132 int main(){
133     MyQueue<int> q;
134     MyQueue1<int> q1;
135
136     for(int i=0; i<10; ++i){
137         q.push(i);
138         q1.push(i);
139     }
140
141     cout<<q.front()<<" "<<q.back()<<endl;
142     cout<<q1.front()<<" "<<q1.back()<<endl;
143     cout<<endl;
144     q.pop();
145     q1.pop();
146     q.push(10);
147     q1.push(10);
148     cout<<q.front()<<" "<<q.back()<<endl;
149     cout<<q1.front()<<" "<<q1.back()<<endl;
150     cout<<endl;
151     cout<<q.size()<<" "<<q.empty()<<endl;
152     cout<<q1.size()<<" "<<q1.empty()<<endl;
153     return 0;
154 }
```

3.6 Write a program to sort a stack in ascending order. You should not make any assumptions about how the stack is implemented. The following are the only functions that should be used to write this program: push | pop | peek | isEmpty.

写程序将一个栈按升序排序。对这个栈是如何实现的，你不应该做任何特殊的假设。 程序中能用到的栈操作有：push | pop | peek | isEmpty。

Solution:

方案1：

使用一个附加的栈来模拟插入排序。将原栈中的数据依次出栈与附加栈中的栈顶元素比较， 如果附加栈为空，则直接将数据压栈。否则， 如果附加栈的栈顶元素大于从原栈中弹出的元素，则将附加栈的栈顶元素压入原栈。 一直这样查找直到附加栈为空或栈顶元素已经不大于该元素， 则将该元素压入附加栈。

代码如下：

```
1 stack<int> Ssort(stack<int> s){
2     stack<int> t;
3     while(!s.empty()){
```

```
        int data = s.top();
        s.pop();
        while(!t.empty() && t.top()>data){
            s.push(t.top());
            t.pop();
        }
        t.push(data);
    }
    return t;
}
```

方案2:
使用一个优先队列来为出栈的元素排序，原栈中的元素不断出栈然后插入优先队列， 直到原栈为空。然后再将优先队列中的元素不断压回原栈，这样操作后， 栈中的元素便有序化了。
代码如下:

```
void Qsort(stack<int> &s){
    priority_queue< int,vector<int>,greater<int> > q;
    while(!s.empty()){
        q.push(s.top());
        s.pop();
    }
    while(!q.empty()){
        s.push(q.top());
        q.pop();
    }
}
```

完整代码如下:

```
#include <iostream>
#include <stack>
#include <queue>
#include <cstdlib>
using namespace std;

stack<int> Ssort(stack<int> s) {
    stack<int> t;
    while ( !s.empty() ) {
        int data = s.top();
        s.pop();
        while ( !t.empty() && t.top() > data ) {
            s.push( t.top() );
            t.pop();
        }
        t.push(data);
    }
    return t;
}

void Qsort(stack<int> &s) {
    priority_queue< int,vector<int>,greater<int> > q;
    while ( !s.empty() ) {
        q.push( s.top() );
        s.pop();
    }
    while ( !q.empty() ) {
        s.push( q.top() );
        q.pop();
    }
};
```

```
32
33  int main() {
34      srand( (unsigned)time(0) );
35      stack<int> s;
36      for (int i = 0; i < 10; ++i)
37          s.push( rand()%100 );
38      Qsort(s);
39      while ( !s.empty() ) {
40          cout << s.top() << endl;
41          s.pop();
42      }
43      return 0;
44  }
```

Chapter 4

Trees and Graphs

4.1 Implement a function to check if a tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that no two leaf nodes differ in distance from the root by more than one.

实现一个函数检查一棵树是否平衡。对于这个问题而言， 平衡指的是这棵树任意两个叶子结点到根结点的距离之差不大于1。

Solution:

对于这道题，要审清题意。它并不是让你判断一棵树是否为平衡二叉树。 平衡二叉树的定义为：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1， 并且左右两个子树都是一棵平衡二叉树。 而本题的平衡指的是这棵树任意两个叶子结点到根结点的距离之差不大于1。 这两个概念是不一样的。例如下图，它是一棵平衡二叉树，但不满足本题的平衡条件。 (叶子结点f和l到根结点的距离之差等于2，不满足题目条件)

对于本题，只需要求出离根结点最近和最远的叶子结点， 然后看它们到根结点的距离之差是否大于1即可。

假设只考虑二叉树，我们可以通过遍历一遍二叉树求出每个叶子结点到根结点的距离。 使用中序遍历，依次求出从左到右的叶子结点到根结点的距离，递归实现。

```
int d = 0, num = 0, dep[maxn];
void getDepth(Node *head){
    if(head == NULL) return;
    ++d;
    getDepth(head->lchild);
    if(head->lchild == NULL && head->rchild == NULL)
        dep[num++] = d;
    getDepth(head->rchild);
    --d;
}
```

求出所有叶子结点到根结点的距离后，再求出其中的最大值和最小值， 然后作差与1比较即可。此外，空树认为是平衡的。代码如下：

```
bool isBalance(Node *head){
    if(head == NULL) return true;
    getDepth(head);
    int max = dep[0], min = dep[0];
    for(int i=0; i<num; ++i){
        if(dep[i]>max) max = dep[i];
        if(dep[i]<min) min = dep[i];
    }
    if(max-min > 1) return false;
    else return true;
}
```

完整代码如下：

```
#include <iostream>
#include <cstring>
```

```
3  using namespace std;
4
5  const int maxn = 100;
6
7  struct Node {
8      int key;
9      Node *left, *right, *parent;
10 };
11 Node *head, *p, node[maxn];
12 int cnt;
13
14 void init() {
15     head = p = NULL;
16     memset(node, '\0', sizeof(node));
17     cnt = 0;
18 }
19
20 void insert(Node* &head, int x) {
21     if (head == NULL) {
22         node[cnt].key = x;
23         node[cnt].parent = p;
24         head = &node[cnt++];
25         return;
26     }
27     p = head;
28     if (x < head->key)
29         insert(head->left, x);
30     else
31         insert(head->right, x);
32 }
33
34 int d = 0, num = 0, depth[maxn];
35 void getDepth(Node * head) {
36     if (head == NULL) return;
37     ++d;
38     getDepth(head->left);
39     if ( !head->left && !head->right ) // leaf node
40         depth[num++] = d;              // leaf node depth
41     getDepth(head->right);
42     --d;
43 }
44
45 bool isBalanced(Node* head) {
46     if (head == NULL) return true;
47     num = 0;
48     getDepth(head);
49
50     int min = depth[0];
51     int max = depth[0];
52     for (int i = 1; i < num; ++i) {
53         if ( depth[i] < min ) min = depth[i];
54         if (depth[i] > max) max = depth[i];
55     }
56     if (max - min > 1)
57         return false;
58     else
59         return true;
60 }
61
```

```
int main() {
    init();
    int a[] = {5, 3, 8, 1, 4, 7, 10, 2, 6, 9, 11, 12};
    for (int i = 0; i < 12; ++i )
        insert(head, a[i]);
    cout << isBalanced(head) << endl;
    return 0;
}
```

4.2 Given a directed graph, design an algorithm to find out whether there is a route between two nodes.
给定一个有向图，设计算法判断两结点间是否存在路径。

Solution:
根据题意，给定一个有向图和起点终点，判断从起点开始，是否存在一条路径可以到达终点。 考查的就是图的遍历，从起点开始遍历该图，如果能访问到终点， 则说明起点与终点间存在路径。稍微修改一下遍历算法即可。
代码如下(在BFS基础上稍微修改)：

```
bool route(int src, int dst){
    q.push(src);
    visited[src] = true;
    while(!q.empty()){
        int t = q.front();
        q.pop();
        if(t == dst) return true;
        for(int i=0; i<n; ++i)
            if(g[t][i] && !visited[i]){
                q.push(i);
                visited[i] = true;
            }
    }
    return false;
}
```

完整代码如下：

```
#include <iostream>
#include <cstring>
#include <queue>
#include <fstream>
using namespace std;

const int maxn = 100;
bool g[maxn][maxn], visited[maxn];
int n;
queue<int> q;

void init() {
    memset(g, false, sizeof(g));
    memset(visited, false, sizeof(visited));
}

bool route(int src, int dst) {
    q.push(src);
    visited[src] = true;
    while ( !q.empty() ) {
        int t = q.front();
        q.pop();
        if (t == dst) return true;

        for (int i = 0; i < n; ++i )
            if ( g[t][i] && !visited[i] ) {
```

```
27                  q.push(i);
28                  visited[i] = true;
29              }
30      }
31      return false;
32  }
33
34  int main() {
35      freopen("4.2.in", "r", stdin);
36      init();
37      int m, u, v;
38      cin >> n >> m;
39      for (int i = 0; i < m; ++i) {
40          cin >> u >> v;
41          g[u][v] = true;
42      }
43      cout << route(0, 6) << endl;
44
45      fclose(stdin);
46
47      return 0;
48  }
```

4.3 Given a sorted (increasing order) array, write an algorithm to create a binary tree with minimal height.

给定一个有序数组(递增)，写程序构建一棵具有最小高度的二叉树。

Solution:

想要使构建出来的二叉树高度最小，那么对于任意结点， 它的左子树和右子树的结点数量应该相当。比如，当我们将一个数放在根结点， 那么理想情况是，我们把数组中剩下的数对半分，一半放在根结点的左子树， 另一半放在根结点的右子树。我们可以定义不同的规则来决定这些数怎样对半分， 不过最简单的方法就是取得有序数组中中间那个数，然后把小于它的放在它的左子树， 大于它的放在它的右子树。不断地递归操作即可构造这样一棵最小高度二叉树。

```
1  void create_minimal_tree(Node* &head, Node *parent, int a[], int start, int end){
2      if(start <= end){
3          int mid = (start + end)>>1;
4          node[cnt].key = a[mid];
5          node[cnt].parent = parent;
6          head = &node[cnt++];
7          create_minimal_tree(head->lchild, head, a, start, mid-1);
8          create_minimal_tree(head->rchild, head, a, mid+1, end);
9      }
10 }
```

完整代码如下：

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  const int maxn = 100;
6
7  struct Node {
8      int key;
9      Node *left, *right;
10     Node *parent;
11 };
12 Node *p, node[maxn];
13 int cnt;
14
15 void init() {
```

```
16      p = NULL;
17      memset(node, '\0', sizeof(node));
18      cnt = 0;
19  }
20
21  void create_minimal_tree(Node* &head, Node *parent, int a[], int start, int end) {
22      if (start <= end) {
23          int mid = (start + end) / 2;
24          node[cnt].key = a[mid];
25          node[cnt].parent = parent;
26          head = &node[cnt++];
27          create_minimal_tree(head->left, head, a, start, mid-1);
28          create_minimal_tree(head->right, head, a, mid+1, end);
29      }
30  }
31
32  int height(Node* head) {
33      if (head == NULL) return 0;
34      return max( height(head->left), height(head->right) ) + 1;
35  }
36
37  int main() {
38      init();
39      int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
40      Node* head = NULL;
41      create_minimal_tree(head, NULL, a, 0, 8);
42      cout << height(head) << endl;
43      return 0;
44  }
```

4.4 Given a binary search tree, design an algorithm which creates a linked list of all the nodes at each depth (i.e., if you have a tree with depth D, you'll have D linked lists).

给定一棵二叉查找树，设计算法，将每一层的所有结点构建为一个链表(也就是说， 如果树有D层，那么你将构建出D个链表)

Solution:

这道题目本质上是个BFS，也就是说，如果已经构建了第i层结点的链表， 那么将此链表中每个结点的左右孩子结点取出，即可构建第i+1层结点的链表。 设结点类型为Node，那么指向每一层链表头结点的类型为list<Node*>， 将每一层头结点指针放到vector中。如果当前层的链表不为空， 那么将该链表的结点依次取出，然后将这些结点的不为空的孩子放入新的链表中。

代码如下：

```
1   vector<list<Node*> > find_level_linklists(Node *head){
2       vector<list<Node*> > res;
3       int level = 0;
4       list<Node*> li;
5       li.push_back(head);
6       res.push_back(li);
7       while(!res[level].empty()){
8           list<Node*> l;
9           list<Node*>::iterator it;
10          for(it=res[level].begin(); it!=res[level].end(); ++it){
11              Node *n = *it;
12              if(n->lchild) l.push_back(n->lchild);
13              if(n->rchild) l.push_back(n->rchild);
14          }
15          ++level;
16          res.push_back(l);
17      }
18      return res;
19  }
```

完整代码如下：

```
#include <iostream>
#include <cstring>
#include <vector>
#include <list>
using namespace std;

const int maxn = 100;

struct Node {
    int key;
    Node *left, *right, *parent;
};
Node *p, node[maxn];
int cnt;

void init() {
    p = NULL;
    memset(node, '\0', sizeof(node));
    cnt = 0;
}

void createMinimalTree (Node* &head, Node *parent, int a[], int start, int end) {
    if (start <= end) {
        //int mid = (start + end) / 2;
        int mid = (start + end) >> 1; // a cheaper implementation
        node[cnt].key = a[mid];
        node[cnt].parent = head;
        head = &node[cnt++];  // needs special attention
        createMinimalTree(head->left, head, a, start, mid-1);
        createMinimalTree(head->right, head, a, mid+1, end);
    }
}

vector<list<Node*> > find_level_linklists( Node *head ) {
    vector<list<Node*> > res;
    int level = 0;
    list<Node*> li;
    li.push_back(head);
    res.push_back(li);

    while ( !res[level].empty() ) {
        list<Node*> l;
        list<Node*>::iterator it;
        for (it = res[level].begin(); it != res[level].end(); ++it) {
            Node *n = *it;
            if (n->left) l.push_back(n->left);
            if (n->right) l.push_back(n->right);
        }
        ++level;
        res.push_back(l); // mistake here before, no index
    }
    return res;
}

void print(vector<list<Node*> > res) {
    vector< list<Node*> >::iterator vit;
    for (vit = res.begin(); vit != res.end(); ++vit) {
        list<Node*> li = *vit;
```

```
59          list<Node*>::iterator lit;
60          for (lit = li.begin(); lit != li.end(); ++lit) {
61              Node *n = *lit;
62              cout << n->key << '\t';
63          }
64          cout << endl;
65      }
66  }
67
68  int main() {
69      init();
70      int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
71      Node *head = NULL;
72      createMinimalTree(head, NULL, a, 0, 8);
73      vector<list<Node*> > res;
74      res = find_level_linklists(head);
75      print(res);
76      return 0;
77  }
```

4.5 Write an algorithm to find the 'next' node (i.e., in-order successor) of a given node in a binary search tree where each node has a link to its parent.

给定二叉查找树的一个结点， 写一个算法查找它的"下一个"结点(即中序遍历后它的后继结点)， 其中每个结点都有指向其父亲的链接。

Solution:

我们知道，在二叉查找树中， 每个结点的值都大于等于它左子树所有结点的值且小于右子树所有结点的值(或是大于它左子树所有结点的值且小于等于右子树所有结点的值，等号放哪边示情况而定。) 二叉查找树中序遍历后，元素将按递增排序， 某一结点的后继结点即为比该结点大的结点中最小的一个。如果该结点有右儿子， 则后继结点为右儿子的最左子孙。否则需要不断地向上查找该结点的祖先， 直到找到第一个比它大的结点为止。

代码如下：

```
1  Node* minimal(Node* no){
2      if(no == NULL) return NULL;
3      while(no->lchild)
4          no = no->lchild;
5      return no;
6  }
7  Node* successor(Node* no){
8      if(no == NULL) return NULL;
9      if(no->rchild) return minimal(no->rchild);
10     Node *y = no->parent;
11     while(y && y->rchild==no){
12         no = y;
13         y = y->parent;
14     }
15     return y;
16 }
```

完整代码如下：

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  const int maxn = 100;
6  struct Node {
7      int key;
8      Node *left, *right, *parent;
9  };
10 Node* p, node[maxn];
```

```
11 int cnt;
12
13 void init() {
14     p = NULL;
15     memset(node, '\0', sizeof(node));
16     cnt = 0;
17 }
18
19 void create_minimal_tree(Node* &head, Node *parent, int a[], int start, int end) {
20     if (start <= end) {
21         int mid = (start + end) >> 1;
22         node[cnt].key = a[mid];
23         node[cnt].parent = head;
24         head = &node[cnt++];
25         create_minimal_tree(head->left, head, a, start, mid-1);
26         create_minimal_tree(head->right, head, a, mid+1, end);
27     }
28 }
29
30 Node* minimal(Node* head) {
31     if (head == NULL) return NULL;
32     while (head->left)
33         head = head->left;
34     return head;
35 }
36
37 Node* successor (Node* ptr) {
38     if (ptr == NULL) return NULL;
39     if (ptr->right)
40         return minimal(ptr->right);
41
42     Node* p = ptr->parent;
43     while (p && p->right == ptr) {
44         ptr = p;
45         p = p->parent;
46     }
47     return p;
48 }
49
50 int main() {
51     int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
52     init();
53     Node *head = NULL;
54     create_minimal_tree(head, NULL, a, 0, 8);
55     cout << "the head is " << head->key << endl;
56     cout << "the successor of head is " << (successor(head))->key << endl;
57     return 0;
58 }
```

4.6 Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.

写程序在一棵二叉树中找到两个结点的第一个共同祖先。不允许存储额外的结点。注意： 这里不特指二叉查找树。

Solution:

本题的关键应当是在Avoid storing additional nodes in a data structure 这句话上。我的理解是，不允许开额外的空间(比如说一个数组)来存储作为中间变量的结点。 虽然我也怀疑它是不是说不允许在结点数据结构Node中加入额外的东西， 比如说父结点的指针。Anyway，我们先从最简单的入手，再一步步加入限制条件。

如果没有任何限制条件，那我觉得最直观的思路就是把其中一个点的所有祖先(包含它自身) 都放入一个哈希表，然后再一步步查找另一个点的祖先结点， 第一个在哈希表中出现的祖先结点即为题目所求。

代码如下，用map模拟(当然，效率比不上哈希表)：

```
Node* first_ancestor(Node* n1, Node* n2){
    if(n1 == NULL || n2 == NULL) return NULL;
    map<Node*, bool> m;
    while(n1){
        m[n1] = true;
        n1 = n1->parent;
    }
    while(n2 && !m[n2]){
        n2 = n2->parent;
    }
    return n2;
}
```

这里用了一个map来存储中间变量，如果题目不允许开额外的辅助空间，那该如何做呢？ 那就老老实实地一个个地试。不断地取出其中一个结点的父结点， 然后判断这个结点是否也为另一个结点的父结点。代码如下：

```
bool father(Node* n1, Node* n2){
    if(n1 == NULL) return false;
    else if(n1 == n2) return true;
    else return father(n1->lchild, n2) || father(n1->rchild, n2);
}
Node* first_ancestor1(Node* n1, Node* n2){
    if(n1 == NULL || n2 == NULL) return NULL;
    while(n1){
        if(father(n1, n2)) return n1;
        n1 = n1->parent;
    }
    return NULL;
}
```

让我们把条件再限制地严苛一些，如果数据结构Node中不允许有指向父亲结点的指针， 那么我们又该如何处理？其实也很简单，首先根结点一定为任意两个结点的共同祖先， 从根结点不断往下找，直到找到最后一个这两结点的共同祖先，即为题目所求。代码如下：

```
void first_ancestor2(Node* head, Node* n1, Node* n2, Node* &ans){
    if(head==NULL || n1==NULL || n2==NULL) return;
    if(head && father(head, n1) && father(head, n2)){
        ans = head;
        first_ancestor2(head->lchild, n1, n2, ans);
        first_ancestor2(head->rchild, n1, n2, ans);
    }
}
```

这里用到了递归，ans最终保存的是这两个结点从根结点算起最后找到的那个祖先。 因为从根结点开始，每次找到满足要求的结点，ans都会被更新。

完整代码如下：

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

const int maxn = 100;
struct Node {
    int key;
    Node *left, *right, *parent;
};
Node *p, node[maxn];
int cnt;

void init() {
    p = NULL;
```

```
16      memset(node, '\0', sizeof(node));
17      cnt = 0;
18  }
19
20  void create_minimal_tree(Node* &head, Node *parent, int a[], int start, int end){
21      if(start <= end){
22          int mid = (start + end)>>1;
23          node[cnt].key = a[mid];
24          node[cnt].parent = parent;
25          head = &node[cnt++];
26          create_minimal_tree(head->left, head, a, start, mid-1);
27          create_minimal_tree(head->right, head, a, mid+1, end);
28      }
29  }
30
31  Node* first_ancestor(Node *n1, Node *n2) {
32      if (n1 == NULL || n2 == NULL) return NULL;
33      map<Node*, bool> m;
34      while (n1) {
35          m[n1] = true;
36          n1 = n1->parent;
37      }
38      while (n2 && !m[n2])
39              n2 = n2->parent;
40      return n2;
41  }
42
43  // mine, confusing, should be wrong
44  /*
45  Node* first_ancestor1(Node *n1, Node *n2) {
46      if (n1 == NULL || n2 == NULL) return NULL;
47      Node *p = n2;
48
49      while (n1) {
50          while (n2 && n2 != n1)
51              n2 = n2->parent;
52          n2 = p;
53          n1 = n1->parent;
54      }
55      return n2;
56  }  */
57
58
59  bool father(Node *n1, Node *n2) {
60      if (n1 == NULL) return false;
61      else if (n1 == n2) return true;
62      else return father(n1->left, n2) || father(n1->right, n2);
63  }
64
65  Node* first_ancestor2(Node *n1, Node *n2) {
66      if (n1 == NULL || n2 == NULL) return NULL;
67
68      while (n1) {
69          if (father(n1, n2)) return n1;
70          n1 = n1->parent;
71      }
72      return NULL;
73  }
74
```

```
75  // no parent pointer
76  void first_ancestor3(Node *head, Node *n1, Node *n2, Node* &ans) {
77      if (head == NULL || n1 == NULL || n2 == NULL) return;
78      if ( head && father(head, n1) && father(head, n2) ) {
79          ans = head;
80          first_ancestor3(head->left, n1, n2, ans);
81          first_ancestor3(head->right, n1, n2, ans);
82      }
83  }
84
85  Node* search(Node* head, int x) {
86      if (head == NULL) return NULL;
87      else if (x == head->key) return head;
88      else if (x <= head->key) return search(head->left, x);
89      else return search(head->right, x);
90  }
91
92  int main() {
93      init();
94      int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
95      Node *head = NULL;
96      create_minimal_tree(head, NULL, a, 0, 8);
97      Node *n1 = search(head, 3);
98      Node *n2 = search(head, 7);
99      cout << "keys: " << n1->key << " " << n2->key << endl;
100
101     Node* ans = first_ancestor2(n1, n2);
102     cout << "ans->key: " << ans->key << endl;
103
104     Node *ans1 = NULL;
105     first_ancestor3(head, n1, n2, ans1);
106     cout << "ans->key: " << ans->key << endl;
107
108     return 0;
109 }
```

4.7 You have two very large binary trees: T1, with millions of nodes, and T2, with hundreds of nodes. Create an algorithm to decide if T2 is a subtree of T1

有两棵很大的二叉树：T1有上百万个结点，T2有上百个结点。写程序判断T2是否为T1的子树。

Solution:

我觉得这道题目意欲考察如何在二叉树中结点数量非常巨大的情况下进行快速的查找与匹配， 不过除了常规的暴力法，我暂时想不到什么高效的方法。暴力法就很简单了， 先在T1中找到T2的根结点，然后依次去匹配它们的左右子树即可。

这里要注意的一点是，T1中的结点可能包含多个与T2根结点的值相同的结点。因此， 在T1中查找T2的根结点时，如果找到与T2匹配的子树，则返回真值；否则，还要继续查找， 直到在T1中找到一棵匹配的子树或是T1中的结点都查找完毕。

代码如下：

```
1  bool match(Node* r1, Node* r2){
2      if(r1 == NULL && r2 == NULL) return true;
3      else if(r1 == NULL || r2 == NULL) return false;
4      else if(r1->key != r2->key) return false;
5      else return match(r1->lchild, r2->lchild) && match(r1->rchild, r2->rchild);
6  }
7  bool subtree(Node* r1, Node* r2){
8      if(r1 == NULL) return false;
9      else if(r1->key == r2->key){
10         if(match(r1, r2)) return true;
11     }
```

```
12      else return subtree(r1->lchild, r2) || subtree(r1->rchild, r2);
13  }
14  bool contain_tree(Node* r1, Node* r2){
15      if(r2 == NULL) return true;
16      else return subtree(r1, r2);
17  }
```

完整代码如下：

```
1   #include <iostream>
2   #include <cstring>
3   using namespace std;
4
5   const int maxn = 100;
6   struct Node {
7       int key;
8       Node *left, *right, *parent;
9   };
10  Node node[maxn];
11  int cnt;
12
13  void init() {
14      memset(node, '\0', sizeof(node));
15      cnt = 0;
16  }
17
18  void create_minimal_tree(Node * &head, Node *parent, int a[], int start, int end) {
19      if (start <= end ) {
20          int mid = (start + end) >> 1;
21          node[cnt].key = a[mid];
22          node[cnt].parent = parent;
23          head = &node[cnt++];
24          create_minimal_tree(head->left, head, a, start, mid-1);
25          create_minimal_tree(head->right, head, a, mid+1, end);
26      }
27  }
28
29  // if two tree matches
30  bool match(Node *n1, Node *n2) {
31      if (n1 == NULL && n2 == NULL) return true;
32      else if (n1 == NULL || n2 == NULL) return false;
33      else if (n1->key != n2->key) return false;
34      else
35          return match(n1->left, n2->left) && match(n1->right, n2->right);
36  }
37
38  bool subtree(Node *n1, Node *n2) {
39      if (n1 == NULL) return false;
40      else if (n1->key == n2->key) {
41          if ( match(n1, n2) ) return true;
42      }
43      else return subtree(n1->left, n2) || subtree(n1->right, n2);
44  }
45
46  bool contain_tree(Node *n1, Node *n2) {
47      if (n2 == NULL) return true;
48      else return subtree(n1, n2);
49  }
50
51  int main() {
```

```
52      init();
53      int a1[] = {0, 1, 2, 3, 4, 5, 6};
54      int a2[] = {0, 1, 2};
55      Node *r1 = NULL, *r2 = NULL;
56      create_minimal_tree(r1, NULL, a1, 0, 6);
57      create_minimal_tree(r2, NULL, a2, 0, 2);
58      if ( contain_tree(r1, r2) )
59          cout << "tree r1 contains tree r2" << endl;
60      else
61          cout  << "tree r1 does not contain tree r2" << endl;
62
63      return 0;
64  }
```

4.8 You are given a binary tree in which each node contains a value. Design an algorithm to print all paths which sum up to that value. Note that it can be any path in the tree - it does not have to start at the root.

给定一棵二叉树，每个结点包含一个值。打印出所有满足以下条件的路径： 路径上结点的值加起来等于给定的一个值。注意：这些路径不必从根结点开始。

Solution:

方案1：如果结点中包含指向父亲结点的指针，那么，只需要去遍历这棵二叉树， 然后从每个结点开始，不断地去累加上它父亲结点的值直到父亲结点为空(这个具有唯一性， 因为每个结点都只有一个父亲结点。也正因为这个唯一性， 可以不另外开额外的空间来保存路径)，如果等于给定的值sum，则打印输出。

代码如下：

```
1   void find_sum(Node* head, int sum){
2       if(head == NULL) return;
3       Node *no = head;
4       int tmp = 0;
5       for(int i=1; no!=NULL; ++i){
6           tmp += no->key;
7           if(tmp == sum)
8               print(head, i);
9           no = no->parent;
10      }
11      find_sum(head->lchild, sum);
12      find_sum(head->rchild, sum);
13  }
```

打印输出时，只需要提供当前结点的指针，及累加的层数即可。然后从当前结点开始， 不断保存其父亲结点的值(包含当前结点)直到达到累加层数，然后逆序输出即可。

代码如下：

```
1   void print(Node* head, int level){
2       vector<int> v;
3       for(int i=0; i<level; ++i){
4           v.push_back(head->key);
5           head = head->parent;
6       }
7       while(!v.empty()){
8           cout<<v.back()<<" ";
9           v.pop_back();
10      }
11      cout<<endl;
12  }
```

方案2：如果结点中不包含指向父亲结点的指针，则在二叉树从上向下查找路径的过程中， 需要为每一次的路径保存中间结果，累加求和仍然是从下至上的，对应到保存路径的数组， 即是从数组的后面开始累加的，这样能保证遍历到每一条路径。

代码如下：

```
void print2(vector<int> v, int level){
    for(int i=level; i<v.size(); ++i)
        cout<<v.at(i)<<" ";
    cout<<endl;
}
void find_sum2(Node* head, int sum, vector<int> v, int level){
    if(head == NULL) return;
    v.push_back(head->key);
    int tmp = 0;
    for(int i=level; i>-1; --i){
        tmp += v.at(i);
        if(tmp == sum)
            print2(v, i);
    }
    vector<int> v1(v), v2(v);
    find_sum2(head->lchild, sum, v1, level+1);
    find_sum2(head->rchild, sum, v2, level+1);
}
```

方案1和方案2的本质思想其实是一样的，不同的只是有无指向父亲结点的指针这个信息。 如果没有这个信息，则需要增加许多额外的空间来存储中间信息。

注意：方案1和方案2代码中的level并非指同一概念，方案1中level表示层数，最小值为1； 方案2中level表示第几层，最小值为0。

完整代码如下：

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

const int maxn = 100;
struct Node {
    int key;
    Node *left, *right, *parent;
};
Node node[maxn];
int cnt;

void init() {
    memset(node, '\0', sizeof(node));
    cnt = 0;
}

void create_minimal_tree(Node* &head, Node *parent, int a[], int start, int end) {
    if (start <= end) {
        int mid = (start + end) >> 1;
        node[cnt].key = a[mid];
        node[cnt].parent = parent;
        head = &node[cnt++];
        create_minimal_tree(head->left, head, a, start, mid-1);
        create_minimal_tree(head->right, head, a, mid+1, end);
    }
}

// method 1: with pointer pointing to parent
void print(Node *head, int level) {
    vector<int> path;
    for (int i = 0; i < level; ++i) {
        path.push_back(head->key);
        head = head->parent;
```

```
36      }
37
38      vector<int>::iterator it;
39      /*for (it = path.end(); it != path.begin(); --it)
40            cout << *it << endl;  */ // somewhere here still WRONG!!!
41      while ( !path.empty() ) {
42          cout << path.back() << " ";
43          path.pop_back();
44      }
45      cout << endl;
46  }
47
48  void find_sum(Node *head, int sum) {
49      if (head == NULL) return;
50
51      Node *ptr = head;
52      int val = 0;
53
54      for (int i = 1; ptr != NULL; ++i ) {
55          val += ptr->key;
56          if (val == sum)
57              print(head, i);
58          ptr = ptr->parent;
59      }
60      find_sum(head->left, sum);
61      find_sum(head->right, sum);
62  }
63
64
65  // method 2: without pointer pointing to parent, complicated
66  void print1(vector<int> v, int level) {
67      for (int i = level; i < v.size(); ++i)
68          cout << v.at(i) << " ";
69      cout << endl;
70  }
71
72  void find_sum1(Node *head, int sum, vector<int> v, int level) {
73      if (head == NULL) return;
74      v.push_back(head->key);
75
76      int val = 0;
77      for (int i = level; i > -1; --i ) {
78          val += v.at(i);
79          if (val == sum)
80              print1(v, i);
81      }
82      vector<int> v1(v), v2(v);
83      find_sum1(head->left, sum, v1, level+1);
84      find_sum1(head->right, sum, v2, level+1);
85  }
86
87
88  int main() {
89      init();
90      int a[] = {4, 3, 8, 5, 2, 1, 6};
91      Node *head = NULL;
92      create_minimal_tree(head, NULL, a, 0, 6);
93      //find_sum(head, 8);
94      vector<int> v;
```

```
95          find_sum1(head, 8, v, 0);
96          return 0;
97      }
```

Chapter 5

Bit Manipulation

5.1 You are given two 32-bit numbers, N and M, and two bit positions, i and j. Write a method to set all bits between i and j in N equal to M (e.g., M becomes a substring of N located at i and starting at j).

EXAMPLE:

Input: N = 10000000000, M = 10101, i = 2, j = 6

Output: N = 10001010100

给定两个32位的数，N和M，还有两个指示位的数，i和j。 写程序使得N中第i位到第j位的值与M中的相同(即：M变成N的子串且位于N的第i位和第j位之间)

输入: N = 10000000000, M = 10101, i = 2, j = 6

输出: N = 10001010100

Solution:

方案1：先将N中第0位到第i位保存下来(左闭右开：[0, i))，记作ret， 然后将N中第0位到第j位全清0([0, j])，通过向右移动j+1位然后再向左移动j+1位得到。 最后用上面清0后的值或上(m«i)再或上ret即可。

代码如下：

```
int update_bits(int n, int m, int i, int j){
    int ret = (1 << i) -1;
    ret &= n;
    return ((n>>(j+1)) << (j+1)) | (m<<i) | ret;
}
```

方案2：用一个左边全为1，中间一段全为0(这段的长度与m长度一样)， 右边全为1的掩码mask去和n按位与，得到的值是将n中间一段清0的结果。 然后再与m左移i位后按位或，得到最终结果。

代码如下：

```
int update_bits1(int n, int m, int i, int j){
    int max = ~0;
    int left = max - ((1 << j+1) - 1);
    int right = ((1 << i) -1);
    int mask = left | right;
    return (n & mask) | (m << i);
}
```

C++中关于位操作，记录几点需要注意的地方：

一个有符号数，如果它的最高位为1，它右移若干位后到达位置i， 那么最高位到第i位之间全是1，例如：

```
int a = 1;
a <<= 31;
a >>= 31;
cout << a << endl;
```

一个无符号数，如果它的最高位为1，它右移若干位后到达位置i， 那么最高位到第i位之间全是0，例如：

```
unsigned int a = 1;
a <<= 31;      //a:1后面带31个0
a >>= 31;      //a:31个0后面带一个1，即1
cout<<a<<endl;      //输出1
```

无论是有符号数还是无符号数，左移若干位后，原来最低位右边新出现的位全为0

一个有符号的正数，它的最高位为0，如果因为左移使得最高位为1，那么它变为负数， 而后无论怎样右移，它都还是负数。除非因为再次左移使最高位变为0，那么它变回正数。

int的最大值：~(1«31)，即0后面跟31个1

int的最小值：1«31，即1后面跟31个0

unsigned int最大值：~0，即32个1

unsigned int最小值：0

其它数据类型与int类似，只是位长不一样。

完整代码如下：

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

void print_binary(int n) {
    vector<int> v;
    int len = 8*sizeof(int);  // bits count
    int mask = 1;
    while(len--) {
        if (n & mask) v.push_back(1);
        else v.push_back(0);

      //mask <<= 1;       // <<= equivalent !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        mask = mask << 1;
    }
    int cnt = 0;
    while ( !v.empty() ) {
        if (cnt%4==0) cout << " ";
        if (cnt%8==0) cout << "  "; // for printing propose only

        cout << v.back();
        v.pop_back();

        ++cnt;
    }
    cout << endl;
}
/*
jenny@jenny-G50VT ~/docu/iv/ctci $ g++ one.cpp
jenny@jenny-G50VT ~/docu/iv/ctci $ ./a.out
   0000 0000   0000 0000   0000 0100   0000 0000
   0000 0000   0000 0000   0000 0000   0001 0101
   0000 0000   0000 0000   0000 0100   0101 0100
*/

int update_bits(int n, int m, int i, int j) {

    int ret = (1 << i) -1;
    ret &= n;

    /* not sure if this method works
    int ret = n;
    ret = ret << (32-i);
    ret = ret >> (32-i);
```

```
45      */
46      /*
47      n = n >> (j+1);
48      n = n << (j+1);
49      n = n | (m << i);
50      n = n | ret;
51      */
52      return ( (n >> (j+1)) << (j+1) ) | m << i | ret;
53  }
54
55  int update_bits2(int n, int m, int i, int j) {
56      int max = ~0; // 全为1, 32位是32个1, 11111111 11111111 11111111 11111111
57      int left = max - ((1 << j+1)-1) ; // 11111111 11111000 00000000 00000000
58      int right = (1 << i) - 1;         // 00000000 00000000 00000000 00111111
59      int mask = left | right;          // 11111111 11111000 00000000 00111111
60      /*
61        int mask = (1 << (33-j) - 1) << j;
62        mask |= (1 << i)-1 ;
63      */
64      return (n & mask) | (m << i);
65  }
66
67  int main() {
68      int n = 1 << 10, m = 21;
69      int ans = update_bits(n, m, 2, 6);
70      print_binary(n);
71      print_binary(m);
72      print_binary(ans);
73      return 0;
74  }
```

5.2 Given a (decimal - e.g. 3.72) number that is passed in as a string, print the binary representation.If the number can not be represented accurately in binary, print “ERROR”.

给定一个字符串类型(string)表示的小数，打印出它的二进制表示。 如果这个数无法精确地表示为二进制形式，输出"ERROR"。

Solution:

整数部分通过不断地对2取余然后除以2来得到其二进制表示， 或是不断地和1按位与然后除以2得到其二进制表示。小数部分则通过不断地乘以2然后与1比较来得到其二进制表示。 小数部分转化为二进制，通过乘以2然后与1比较，大于等于1则该位为1，并且该值减去1； 否则该位为0。不断地通过这种操作最终能使该小数部分的值变为0的，即可精确表示。 否则将无法用有限的位数来表示这个小数部分。我们可以设定一个长度，比如说32， 在32位之内还无法精确地表示这个小数部分的，我们就认为它无法精确表示了。

代码如下:

```
1   #include <iostream>
2   #include <cstring>
3   #include <cstdlib>
4   using namespace std;
5
6   string print_binary(string val) {
7       int pos = val.find('.', 0);
8       int intpart =  atoi( val.substr(0, pos).c_str() ); // c_str() function
9       double decipart = atof( val.substr(pos, val.length()-pos).c_str() );
10      string left = "", right = "";
11
12      while(intpart > 0) {
13          if (intpart & 1) left = "1"+left;
14          else left = "0" + left;
15          intpart >>= 1;
16      }
```

```
17
18      int cnt = 0;
19      while (decipart > 0) {
20          if ( right.length() > 32 ) return "ERROR";
21
22          decipart *= 2;
23          if (decipart >= 1) {
24              right += "1";
25              decipart -= 1;
26          } else
27              right += "0";
28      }
29
30      return left + "." + right;
31  }
32
33  int main() {
34      string val = "19.25";
35      cout << print_binary(val) << endl;
36      return 0;
37  }
```

5.3 Given an integer, print the next smallest and next largest number that have the same number of 1 bits in their binary representation.

给定一个整数x，找出另外两个整数，这两个整数的二进制表示中1的个数和x相同， 其中一个是比x大的数中最小的，另一个是比x小的数中最大的。

Solution:

对于这道题，我们先完成一个最朴素最直接的版本，以保证其正确性。 这个版本可以作为其它版本的验证工具。

什么方法是最直接的呢? 给定一个数x，计算出它的二进制表示中1的个数为num， 然后x不断加1并计算其二进制表示中1的个数，当它再次等于num时， 就找到了比x大且二进制表示中1的个数相同的最小的数。类似地， 可以找到比x小且二进制表示中1的个数相同的最大的数，只不过x变为不断减1而已。

代码如下:

```
1   int next(int x){
2       int max_int = ~(1<<31);
3       int num = count_one(x);
4       if(num == 0 || x == -1) return -1;
5       for(++x; count_one(x) != num && x < max_int; ++x);
6       if(count_one(x) == num) return x;
7       return -1;
8   }
9
10  int previous(int x){
11      int min_int = (1<<31);
12      int num = count_one(x);
13      if(num == 0 || x == -1) return -1;
14      for(--x; count_one(x) != num && x > min_int; --x);
15      if(count_one(x) == num) return x;
16      return -1;
17  }
```

count_one函数的功能是计算一个数的二进制表示中1的个数，这个要怎么实现呢? 一种方法是通过不断地移位判断最低位是否为1然后计数器累加，代码如下:

```
1   int count_one(int x){
2       int cnt = 0;
3       for(int i=0; i<32; ++i){
4           if(x & 1) ++cnt;
5           x >>= 1;
6       }
```

```
    return cnt;
}
```

这里for循环可否换成while(x > 0)呢，如果x恒为正数是没问题的，可是如果x为负数， 那么x是无法通过不断地右移变为非负数的。所以这里用for循环比较保险。

这种方法非常直观，不过还有更高效更优美的方法。 这种方法先将相邻的位包含的1的个数相加，然后将相邻每2位包含的1的个数相加， 再然后将相邻每4位包含的1的个数相加.....最后即可统计出一个数中包含的1的个数。 代码中大部分操作都是位操作，执行起来非常高效。

代码如下：

```
int count_one(int x){
    x = (x & (0x55555555)) + ((x >> 1) & (0x55555555));
    x = (x & (0x33333333)) + ((x >> 2) & (0x33333333));
    x = (x & (0x0f0f0f0f)) + ((x >> 4) & (0x0f0f0f0f));
    x = (x & (0x00ff00ff)) + ((x >> 8) & (0x00ff00ff));
    x = (x & (0x0000ffff)) + ((x >> 16) & (0x0000ffff));
    return x;
}
```

好了，接下来考虑除了朴素方案外，有没有更高效的方案。假设给定的数的二进制表示为： 1101110，我们从低位看起，找到第一个1，从它开始找到第一个0，然后把这个0变为1， 比这个位低的位全置0，得到1110000，这个数比原数大，但比它少了两个1， 直接在低位补上这两个1，得到，1110011，这就是最终答案。 我们可以通过朴素版本来模拟这个答案是怎么得到的:

1101110->1101111->1110000->1110001->1110010->1110011

接下来，我们来考虑一些边界情况，这是最容易被忽略的地方(感谢细心的读者)。 假设一个32位的整数， 它的第31位为1，即：0100..00，那么按照上面的操作，我们会得到1000..00， 很不幸，这是错误的。因为int是有符号的，意味着我们得到了一个负数。 我们想要得到的是一个比0100..00大的数，结果得到一个负数，自然是不对的。事实上比0100..00大的且1的个数和它一样的整数是不存在的，扩展可知， 对于所有的0111..，都没有比它们大且1的个数和它们一样的整数。对于这种情况， 直接返回-1。-1的所有二进制位全为1，不存在一个数说1的个数和它一样还比它大或小的， 因此适合作为找不到答案时的返回值。

另一个边界情况是什么呢? 就是对于形如11100..00的整数，它是一个负数， 比它大且1的个数相同的整数有好多个，最小的当然是把1都放在最低位了：00..0111。

代码如下：

```
int next1(int x){
    int xx = x, bit = 0;
    for(; (x&1) != 1 && bit < 32; x >>= 1, ++bit);
    for(; (x&1) != 0 && bit < 32; x >>= 1, ++bit);
    if(bit == 31) return -1; //011.., none satisify
    x |= 1;
    x <<= bit; // wtf, x<<32 != 0,so use next line to make x=0
    if(bit == 32) x = 0; // for 11100..00
    int num1 = count_one(xx) - count_one(x);
    int c = 1;
    for(; num1 > 0; x |= c, --num1, c <<= 1);
    return x;
}
```

类似的方法可以求出另一个数，这里不再赘述。代码如下：

```
int previous1(int x){
    int xx = x, bit = 0;
    for(; (x&1) != 0 && bit < 32; x >>= 1, ++bit);
    for(; (x&1) != 1 && bit < 32; x >>= 1, ++bit);
    if(bit == 31) return -1; //100..11, none satisify
    x -= 1;
    x <<= bit;
    if(bit == 32) x = 0;
    int num1 = count_one(xx) - count_one(x);
    x >>= bit;
    for(; num1 > 0; x = (x<<1) | 1, --num1, --bit);
```

```
12      for(; bit > 0; x <<= 1, --bit);
13      return x;
14  }
```

完整代码如下:

```
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   void print_binary(int x) {
6       vector<int> v;
7       int cnt = 0, mask = 1;
8       while (cnt < 32) {
9           if (x & mask) v.push_back(1);
10          else v.push_back(0);
11          mask <<= 1;
12          ++cnt;
13      }
14
15      cnt = 0;
16      while ( !v.empty() ) {
17          if (cnt % 4 == 0) cout << "␣";
18          if (cnt % 8 == 0) cout << "␣␣";
19          cout << v.back();
20          v.pop_back();
21          ++cnt;
22      }
23      cout << endl;
24  }
25
26  int count1(int x) {
27      int cnt = 0;
28      for (int i = 0; i < 32; ++i) {  // for is safer !!!!!!
29          if (x & 1) ++cnt;
30          x >>= 1;
31      }
32      return cnt;
33  }
34
35  int count(int x) {
36      int cnt = 0;
37      while (x > 0) {
38          if (x & 1) ++cnt;
39          x >>= 1;
40      }
41      return cnt;
42  }
43
44  int count_one (int x) {
45      x = (x & (0x55555555)) + ((x >> 1) & (0x55555555));
46      x = (x & (0x33333333)) + ((x >> 2) & (0x33333333));
47      x = (x & (0x0f0f0f0f)) + ((x >> 4) & (0x0f0f0f0f));
48      x = (x & (0x00ff00ff)) + ((x >> 8) & (0x00ff00ff));
49      x = (x & (0x0000ffff)) + ((x >> 16) & (0x0000ffff));
50      return x;
51  }
52
53  int count_oneP (int x) {
54      print_binary(x);
```

```
55      cout << endl;
56
57      cout << "1:" << endl;
58      print_binary(x);
59      print_binary(0x55555555);
60      print_binary(x & (0x55555555));
61      cout << endl;
62      print_binary(x >> 1);
63      print_binary(0x55555555);
64      print_binary((x >> 1) & (0x55555555));
65      cout << endl;
66      print_binary(x & (0x55555555));
67      print_binary((x >> 1) & (0x55555555));
68      x = (x & (0x55555555)) + ((x >> 1) & (0x55555555));
69      print_binary(x);
70      cout << endl;
71
72      cout << "2:" << endl;
73      print_binary(x);
74      print_binary(0x33333333);
75      print_binary(x & (0x33333333));
76      cout << endl;
77      print_binary(x >> 2);
78      print_binary(0x33333333);
79      print_binary((x >> 2) & (0x33333333));
80      cout << endl;
81      print_binary(x & (0x33333333));
82      print_binary((x >> 2) & (0x33333333));
83      x = (x & (0x33333333)) + ((x >> 2) & (0x33333333));
84      print_binary(x);
85      cout << endl;
86
87      cout << "4:" << endl;
88      print_binary(x);
89      print_binary(0x0f0f0f0f);
90      print_binary(x & (0x0f0f0f0f));
91      cout << endl;
92      print_binary(x >> 4);
93      print_binary(0x0f0f0f0f);
94      print_binary((x >> 4) & (0x0f0f0f0f));
95      cout << endl;
96      print_binary(x & (0x0f0f0f0f));
97      print_binary((x >> 4) & (0x0f0f0f0f));
98      x = (x & (0x0f0f0f0f)) + ((x >> 4) & (0x0f0f0f0f));
99      print_binary(x);
100     cout << endl;
101
102     cout << "8:" << endl;
103     print_binary(x);
104     print_binary(0x00ff00ff);
105     print_binary(x & (0x00ff00ff));
106     cout << endl;
107     print_binary(x >> 8);
108     print_binary(0x00ff00ff);
109     print_binary((x >> 8) & (0x00ff00ff));
110     cout << endl;
111     print_binary(x & (0x00ff00ff));
112     print_binary((x >> 8) & (0x00ff00ff));
113     x = (x & (0x00ff00ff)) + ((x >> 8) & (0x00ff00ff));
```

```
114     print_binary(x);
115     cout << endl;
116 
117     cout << "16:" << endl;
118     print_binary(x);
119     print_binary(0x0000ffff);
120     print_binary(x & (0x0000ffff));
121     cout << endl;
122     print_binary(x >> 16);
123     print_binary(0x0000ffff);
124     print_binary((x >> 16) & (0x0000ffff));
125     cout << endl;
126     print_binary(x & (0x0000ffff));
127     print_binary((x >> 16) & (0x0000ffff));
128     x = (x & (0x0000ffff)) + ((x >> 16) & (0x0000ffff));
129     print_binary(x);
130     return x;
131 }
132 
133 /* mine, not complete
134 int next(int x) {
135     int num = count(x);
136     int val = x+1;
137     while ( count(val) != num )
138         val += 1;
139     return val;
140 } */
141 
142 int next1(int x) {
143     int max_int = ~(1<<31);
144     int num = count(x);
145     if (x == 0 || x == -1) return -1;
146     for (++x; count(x) != num && x < max_int; ++x);
147     if (count(x) == num) return x;
148     return -1;
149 }
150 
151 /* mine, not complete
152 int previous(int x) {
153     int num = count(x);
154     int val = x-1;
155     while ( count(val) != num )
156         val -= 1;
157     return val;
158 } */
159 
160 int previous1(int x) {
161     int min_int = (1 << 31);
162     int num = count(x);
163     if (x == 0 || x == -1) return -1;
164     for (--x; count(x) != num && x > min_int; --x);
165     if (count(x) == num) return x;
166     return -1;
167 }
168 
169 int next2(int x) {
170     int xx = x, bit = 0;
171     for (; (x&1) != 1 && bit < 32; x >>= 1, ++bit);
172     for (; (x&1) != 0 && bit < 32; x >>= 1, ++bit);
```

```
173     if (bit == 31) return -1; // 011---, none satisfy
174     x |= 1;
175     x <<= bit; // wtf, x<<32 != 0, so use next line to make x=0
176     if (bit == 32) x = 0; // for 11100---00
177     int num1 = count(xx) - count(x);
178     int c = 1;
179     for (; num1 > 0; x |= c, --num1, c <<= 1);
180     return x;
181 }
182
183 int previous2(int x){
184     int xx = x, bit = 0;
185     for(; (x&1) != 0 && bit < 32; x >>= 1, ++bit);
186     for(; (x&1) != 1 && bit < 32; x >>= 1, ++bit);
187     if(bit == 31) return -1; //100..11, none satisify
188     x -= 1;
189     x <<= bit;
190     if(bit == 32) x = 0;
191     int num1 = count_one(xx) - count_one(x);
192     x >>= bit;
193     for(; num1 > 0; x = (x<<1) | 1, --num1, --bit);
194     for(; bit > 0; x <<= 1, --bit);
195     return x;
196 }
197
198 int main() {
199     //int x = (1<<30) | (1<<28) | (1<<25) | (1<<21) | (1<<19) | (1<<15)
200     //       | (1<<13) | (1<<10) | (1<<8) | (1<<6) | (1<<5) | (1<<2);
201
202     int x = -976756; // (1<<31)+(1<<29); // -8737776;
203
204     //int cnt = count_oneP(x);
205     //cout << "cnt: " << cnt << endl;
206
207     print_binary(x);
208     cout << endl;
209     print_binary( next1(x) );
210     print_binary( next2(x) );
211     cout << endl;
212     print_binary( previous1(x) );
213     print_binary( previous2(x) );
214
215     return 0;  // the result may have problem
216 }
217
218
219 /*
220 jenny@jenny-G50VT ~/docu/iv/ctci $ g++ thr.cpp
221 jenny@jenny-G50VT ~/docu/iv/ctci $ ./a.out
222    0101 0010   0010 1000   1010 0101   0110 0100
223
224 1:
225    0101 0010   0010 1000   1010 0101   0110 0100
226    0101 0101   0101 0101   0101 0101   0101 0101
227    0101 0000   0000 0000   0000 0101   0100 0100
228
229    0010 1001   0001 0100   0101 0010   1011 0010
230    0101 0101   0101 0101   0101 0101   0101 0101
231    0000 0001   0001 0100   0101 0000   0001 0000
```

```
232
233     0101 0000   0000 0000   0000 0101   0100 0100
234     0000 0001   0001 0100   0101 0000   0001 0000
235     0101 0001   0001 0100   0101 0101   0101 0100
236
237 2:
238     0101 0001   0001 0100   0101 0101   0101 0100
239     0011 0011   0011 0011   0011 0011   0011 0011
240     0001 0001   0001 0000   0001 0001   0001 0000
241
242     0001 0100   0100 0101   0001 0101   0101 0101
243     0011 0011   0011 0011   0011 0011   0011 0011
244     0001 0000   0000 0001   0001 0001   0001 0001
245
246     0001 0001   0001 0000   0001 0001   0001 0000
247     0001 0000   0000 0001   0001 0001   0001 0001
248     0010 0001   0001 0001   0010 0010   0010 0001
249
250 4:
251     0010 0001   0001 0001   0010 0010   0010 0001
252     0000 1111   0000 1111   0000 1111   0000 1111
253     0000 0001   0000 0001   0000 0010   0000 0001
254
255     0000 0010   0001 0001   0001 0010   0010 0010
256     0000 1111   0000 1111   0000 1111   0000 1111
257     0000 0010   0000 0001   0000 0010   0000 0010
258
259     0000 0001   0000 0001   0000 0010   0000 0001
260     0000 0010   0000 0001   0000 0010   0000 0010
261     0000 0011   0000 0010   0000 0100   0000 0011
262
263 8:
264     0000 0011   0000 0010   0000 0100   0000 0011
265     0000 0000   1111 1111   0000 0000   1111 1111
266     0000 0000   0000 0010   0000 0000   0000 0011
267
268     0000 0000   0000 0011   0000 0010   0000 0100
269     0000 0000   1111 1111   0000 0000   1111 1111
270     0000 0000   0000 0011   0000 0000   0000 0100
271
272     0000 0000   0000 0010   0000 0000   0000 0011
273     0000 0000   0000 0011   0000 0000   0000 0100
274     0000 0000   0000 0101   0000 0000   0000 0111
275
276 16:
277     0000 0000   0000 0101   0000 0000   0000 0111
278     0000 0000   0000 0000   1111 1111   1111 1111
279     0000 0000   0000 0000   0000 0000   0000 0111
280
281     0000 0000   0000 0000   0000 0000   0000 0101
282     0000 0000   0000 0000   1111 1111   1111 1111
283     0000 0000   0000 0000   0000 0000   0000 0101
284
285     0000 0000   0000 0000   0000 0000   0000 0111
286     0000 0000   0000 0000   0000 0000   0000 0101
287     0000 0000   0000 0000   0000 0000   0000 1100
288 cnt: 12
289
290 */
```

5.4 Explain what the following code does: ((n & (n-1)) == 0).

解答以下代码的作用：((n & (n-1)) == 0)

Solution:

这个比较简单，代码的作用是判断一个数是否为2的整数次幂。题目中的判断代码不够严谨， 因为当n=0时，上述条件为真，但0并不是2的幂。所以，上述语句可以修改为：

(n > 0) && ((n & (n-1)) == 0)

5.5 Write a function to determine the number of bits required to convert integer A to integer B.

Input: 31, 14

Output: 2

写程序计算从整数A变为整数B需要修改的二进制位数。

输入：31，14

输出：2

Solution:

这道题目也比较简单，从整数A变到整数B，所需要修改的就只是A和B二进制表示中不同的位， 先将A和B做异或，然后再统计结果的二进制表示中1的个数即可。

代码如下：

```
int count_one(int x){
    x = (x & (0x55555555)) + ((x >> 1) & (0x55555555));
    x = (x & (0x33333333)) + ((x >> 2) & (0x33333333));
    x = (x & (0x0f0f0f0f)) + ((x >> 4) & (0x0f0f0f0f));
    x = (x & (0x00ff00ff)) + ((x >> 8) & (0x00ff00ff));
    x = (x & (0x0000ffff)) + ((x >> 16) & (0x0000ffff));
    return x;
}

int convert_num(int a, int b){
    return count_one(a^b);
}
```

完整代码如下：

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

int count(int x) {
    int cnt = 0;
    for (int i = 0; i < 32; ++i) {  // refer to ch5thr.cpp
        if (x & 1) ++cnt;
        x >>= 1;
    }
    return cnt;
}

int count_one(int x){
    x = (x & (0x55555555)) + ((x >> 1) & (0x55555555));
    x = (x & (0x33333333)) + ((x >> 2) & (0x33333333));
    x = (x & (0x0f0f0f0f)) + ((x >> 4) & (0x0f0f0f0f));
    x = (x & (0x00ff00ff)) + ((x >> 8) & (0x00ff00ff));
    x = (x & (0x0000ffff)) + ((x >> 16) & (0x0000ffff));
    return x;
}

int convert_num(int a, int b) {
    return count(a ^ b);
}

int main() {
```

```
27      int a = 31, b = 14;
28      cout << "Bits needs to convert: " << convert_num(a, b) << endl;
29      return 0;
30  }
```

5.6 Write a program to swap odd and even bits in an integer with as few instructions as possible (e.g., bit 0 and bit 1 are swapped, bit 2 and bit 3 are swapped, etc).

写程序交换一个整数二进制表示中的奇数位和偶数位，用尽可能少的代码实现。(比如，第0位和第1位交换，第2位和第3位交换...)

Solution:

这道题目比较简单。分别将这个整数的奇数位和偶数位提取出来，然后移位取或即可。

代码如下：

```
1  int swap_bits(int x){
2      return ((x & 0x55555555) << 1) | ((x >> 1) & 0x55555555);
3  }
```

当然也可以采用更自然的方式来写这段代码：

```
1  int swap_bits1(int x){
2      return ((x & 0x55555555) << 1) | ((x & 0xAAAAAAAA) >> 1);
3  }
```

上面的代码思路和作用都是一样的，不过按照《Hacker's delight》这本书里的说法， 第一种方法避免了在一个寄存器中生成两个大常量。如果计算机没有与非指令， 将导致第二种方法多使用1个指令。总结之，就是第一种方法更好。:P

完整代码如下：

```
1  #include <iostream>
2  using namespace std;
3
4  int swap_bits(int x) {
5      return ((x & 0x55555555) << 1) | ((x>>1) & 0x55555555);
6  }
7
8  int swap_bits1(int x) {
9      return ((x & 0x55555555) << 1) | ((x & 0xaaaaaaaa) >> 1);
10 }
11
12 void print_binary (int x) {
13     string s = " ";
14     for (int i = 0; i < 32 && x != 0; ++i, x >>= 1) {  // try me !!!
15         if (i % 4 == 0) s += " ";
16         if (i % 8 == 0) s += "  "; // for printing propose
17
18         if (x & 1) s += "1";
19         else s += "0";
20     }
21     cout << s << endl;
22 }
23
24 int main() {
25     int x = -7665543;
26     print_binary(x);
27     print_binary(swap_bits(x));
28     print_binary(swap_bits1(x));
29     return 0;
30 }
31
32 /*
33 jenny@jenny-G50VT ~/docu/iv/ctci $ g++ six.cpp
```

```
jenny@jenny-G50VT ~/docu/iv/ctci $ ./a.out
     1001 1110    0001 0000    1101 0001    1111 1111
     0110 1101    0010 0000    1110 0010    1111 1111
     0110 1101    0010 0000    1110 0010    1111 1111
*/
```

5.7 An array A[1...n] contains all the integers from 0 to n except for one number which is missing. In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the jth bit of A[i]", which takes constant time. Write code to find the missing integer. Can you do it in O(n) time?

数组A[1...n]包含0到n的所有整数，但有一个整数丢失了。在这个问题中， 我们不能直接通过A[i]取得数组中的第i个数。数组A被表示成二进制， 也就是一串的0/1字符，而我们唯一能使用的操作只有"取得A[i]中的第j位"， 这个操作只需要花费常数时间。写程序找出丢失的整数，你能使程序的时间复杂度是O(n)吗?

Solution:

首先，在这个问题中，明确我们唯一能使用的操作是fetch(a, i, j)即： 取得a[i]中的第j位。它是提供给我们的操作，怎么实现的不用去理它， 而我们要利用它来解决这个问题，并且在我们的程序中，不能使用a[i]这样的操作。

如果我们不能直接使用a[i]，那么我们能利用fetch函数来获得a[i]吗? 回答是可以。 数组a中每个元素是一个整型数，所以只要每次取出32位，再计算出它的值即可。

代码如下:

```
int get(int a[], int i){
    int ret = 0;
    for(int j=31; j>=0; --j)
        ret = (ret << 1) | fetch(a, i, j);
    return ret;
}
```

我们已经通过get(a, i)来取得a[i]的值了，这样一来，我们只需要开一个bool数组， 把出现过的整数标记为true，即可找出丢失的那个整数。

代码如下:

```
int missing(int a[], int n){
    bool *b = new bool[n+1];
    memset(b, false, (n+1)*sizeof(bool));
    for(int i=0; i<n; ++i)
        b[get(a, i)] = true;
    for(int i=0; i<n+1; ++i){
        if(!b[i]) return i;
    }
    delete[] b;
}
```

我们把问题再变难一点点，如果我们能取到的只是数组a中的第j位，即有函数fetch(a, j)， 而不是取到a[i]中的第j位，那又应该怎么做呢。其实也很简单， 计算a[i]需要取出a[i]中的第31位到第0位(共32位)，对应到整个数组上，就是数组从头开始数起，取出第32*i+31位到第32*i位，代码如下:

```
int get1(int a[], int i){
    int ret = 0;
    int base = 32*i;
    for(int j=base+31; j>=base; --j)
        ret = (ret << 1) | fetch1(a, j);
    return ret;
}
```

完整代码如下:

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

int fetch (int a[], int i, int j) {
```

```
6       return (a[i] >> j) & 1; // return 0 or 1
7   }
8
9   int get(int a[], int i) {
10      int ret = 0;
11      for (int j = 31; j >= 0; --j)
12          ret = (ret << 1) | fetch(a, i, j);
13      return ret;
14  }
15
16  int fetch1 (int a[], int i) {
17      //return (a[i/32] >> (32 - i % 32)) & 1;  // not sure what's going on
18      return (a[i/32] >> (i % 32)) & 1;
19  }
20
21  int get1(int a[], int i) {
22      int ret = 0;
23      int base = 32 * i;
24      for (int j = base + 31; j >= base; --j)
25          ret = (ret << 1) | fetch1(a, j);
26      return ret;
27  }
28
29  int missing(int a[], int n) {
30      bool *b = new bool[n+1];
31      memset(b, false, (n+1)*sizeof(bool));
32
33      for (int i = 0; i < n; ++i)
34          b[get(a, i)] = true;
35      for (int i = 0; i <= n; ++i)
36          if ( !b[i] )
37              return i;
38      delete[] b;
39  }
40
41  int missing1(int a[], int n){
42      bool *b = new bool[n+1];
43      memset(b, false, (n+1)*sizeof(bool));
44      for(int i=0; i<n; ++i)
45          b[get1(a, i)] = true;
46      for(int i=0; i<n+1; ++i){
47          if(!b[i]) return i;
48      }
49      delete[] b;
50  }
51
52  int main() {
53      int a[] = {0, 1, 2, 3, 4, 5, 7, 8, 9, 10};
54      cout << missing(a, 10) << endl;
55      cout << missing1(a, 10) << endl;
56      return 0;
57  }
```

Chapter 6

Brain Teasers

6.1 Add arithmetic operators (plus, minus, times, divide) to make the following expression true: 3 1 3 6 = 8. You can use any parentheses you'd like.

6.2 There is an 8x8 chess board in which two diagonally opposite corners have been cutoff. You are given 31 dominos, and a single domino can cover exactly two squares. Can you use the 31 dominos to cover the entire board? Prove your answer (by providing an example, or showing why it's impossible).

6.3 You have a five quart jug and a three quart jug, and an unlimited supply of water (but no measuring cups). How would you come up with exactly four quarts of water? NOTE: The jugs are oddly shaped, such that filling up exactly 'half' of the jug would be impossible.

6.4 A bunch of men are on an island. A genie comes down and gathers everyone together and places a magical hat on some people's heads (i.e., at least one person has a hat). The hat is magical: it can be seen by other people, but not by the wearer of the hat himself. To remove the hat, those (and only those who have a hat) must dunk themselves underwater at exactly midnight. If there are n people and c hats, how long does it take the men to remove the hats? The men cannot tell each other (in any way) that they have a hat.

FOLLOW UP

Prove that your solution is correct.

6.5 There is a building of 100 floors. If an egg drops from the Nth floor or above it will break. If it's dropped from any floor below, it will not break. You're given 2 eggs. Find N, while minimizing the number of drops for the worst case.

6.6 There are one hundred closed lockers in a hallway. A man begins by opening all one hundred lockers. Next, he closes every second locker. Then he goes to every third locker and closes it if it is open or opens it if it is closed (e.g., he toggles every third locker). After his one hundredth pass in the hallway, in which he toggles only locker number one hundred, how many lockers are open?

Chapter 7

Object Oriented Design

7.1 Design the data structures for a generic deck of cards. Explain how you would subclass it to implement particular card games.

7.2 Imagine you have a call center with three levels of employees: fresher, technical lead(TL), product manager (PM). There can be multiple employees, but only one TL or PM. An incoming telephone call must be allocated to a fresher who is free. If a fresher can't handle the call, he or she must escalate the call to technical lead. If the TL is not free or not able to handle it, then the call should be escalated to PM. Design the classes and data structures for this problem. Implement a method getCallHandler().

7.3 Design a musical juke box using object oriented principles.

7.4 Design a chess game using object oriented principles.

7.5 Design the data structures for an online book reader system.

7.6 Implement a jigsaw puzzle. Design the data structures and explain an algorithm to solve the puzzle.

7.7 Explain how you would design a chat server. In particular, provide details about the various backend components, classes, and methods. What would be the hardest problems to solve?

7.8 Othello is played as follows: Each Othello piece is white on one side and black on the other. When a piece is surrounded by its opponents on both the left and right sides, or both the top and bottom, it is said to be captured and its color is flipped. On your turn, you must capture at least one of your opponent's pieces. The game ends when either user has no more valid moves, and the win is assigned to the person with the most pieces. Implement the object oriented design for Othello.

7.9 Explain the data structures and algorithms that you would use to design an in-memory file system. Illustrate with an example in code where possible.

7.10 Describe the data structures and algorithms that you would use to implement a garbage collector in C++.

Chapter 8

Recursion

8.1 Write a method to generate the nth Fibonacci number.

写一个函数来产生第n个斐波那契数。

Solution:

斐波那契数列的定义如下：

```
f(1) = f(2) = 1;
f(n) = f(n-1) + f(n-2);
```

这个定义是递归的，因此很容易根据以上的定义写出它的递归解法， 由于这个数列的递增速度飞快XD，我们先重定义一下long long好方便使用：

```
typedef long long ll;
```

递归版本：

```
ll fib(ll n){
    if(n < 1) return -1;
    if(n == 1 || n == 2) return 1;
    else return fib(n-1) + fib(n-2);
}
```

当然了，根据定义我们也可以很容易地写出它的非递归版本(迭代版本)：

```
ll fib1(ll n){
    if(n < 1) return -1;
    if(n == 1 || n == 2) return 1;
    ll a = 1, b = 1;
    for(ll i=3; i<=n; ++i){
        ll c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

空间复杂度O(1)，时间复杂度O(n)，看起来既简单又快速。可是，我们还有更快的解法。 根据上面的递推公式，我们可以得到它的矩阵版本：

从上图可以看出，写成矩阵递推形式，可以让我们一推到底。最后的f(1)=f(2)=1， 因此，这个问题就转换成了，如何求矩阵的幂。当然了，要快速，不然就没有什么意义了。 我们先把问题退化一下，先不考虑求矩阵的幂，而是求一个整数的幂，这个够简单的吧。

先来看看最naive的解法：(方便起见，这里假设n为非负数，不对n小于0的情况做讨论)

```
ll pow(ll m, ll n){
    ll res = 1;
    for(ll i=0; i<n; ++i)
```

```
4           res *= m;
5       return res;
6   }
```

时间复杂度O(n)。现在让我们来考虑一种更快的方法，假设我们要计算m13 ， 然后我们把指数13写成二进制形式13=1101，一开始结果res=1.我们要计算的幂可以写成：

m13 = m1 * m4 * m8

我们可以很直观的得出，如果指数13的二进制形式1101中的某一位为1，那么， res就去乘以那一位对应的一个数。比如，1101从低位起，第1位为1，那么res乘以m1 ， 第二位为0，res不需要乘以m2 ，第三位为1，res乘以m4 ，第四位为1，res乘以m8 ， 最后得到的就是：

res = m1 * m4 * m8

而且由于每次res去乘以的数(如果该位为0则不乘)都是上一次那个数的平方， 所以，这个数我用完一次，就对它取平方，准备下一次的使用即可。看代码：

```
1   ll pow1(ll m, ll n){
2       ll res = 1;
3       while(n > 0){
4           if(n&1) res *= m;
5           m *= m;
6           n >>= 1;
7       }
8       return res;
9   }
```

时间复杂度O(logn)，正是我们想要的快速版本。OK， 这时候如果让你快速求矩阵的幂，是不是很简单了？只需要将实数乘法改成矩阵乘法即可。

```
1   void pow(ll s[2][2], ll a[2][2], ll n){
2       while(n > 0){
3           if(n&1) mul(s, s, a);
4           mul(a, a, a);
5           n >>= 1;
6       }
7   }
```

基本上是一模一样的，只不过由于计算结果是矩阵，不能直接用return进行返回， 而是在函数的参数列表中返回。矩阵乘法函数如下(只考虑2*2的矩阵乘法)

```
1   void mul(ll c[2][2], ll a[2][2], ll b[2][2]){
2       ll t[4];
3       t[0] = a[0][0]*b[0][0] + a[0][1]*b[1][0];
4       t[1] = a[0][0]*b[0][1] + a[0][1]*b[1][1];
5       t[2] = a[1][0]*b[0][0] + a[1][1]*b[1][0];
6       t[3] = a[1][0]*b[0][1] + a[1][1]*b[1][1];
7       c[0][0] = t[0];
8       c[0][1] = t[1];
9       c[1][0] = t[2];
10      c[1][1] = t[3];
11  }
```

于是，求斐波那契数列第n项的O(logn)解法如下：

```
1   ll fib2(ll n){
2       if(n < 1) return -1;
3       if(n == 1 || n == 2) return 1;
4
5       ll a[2][2] = { {1, 1}, {1, 0} };
6       ll s[2][2] = { {1, 0}, {0, 1} };
7       pow(s, a, n-2);
8       return s[0][0] + s[0][1];
9   }
```

完整代码如下：

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

typedef long long ll;

ll fib(ll n) {
    if (n < 1) return -1;
    if (n == 1 | n == 2) return 1;
    else
        return fib(n-1) + fib(n-2);
}

ll fibm(ll n) {     // mine, works
    if (n < 1) return -1;
    if (n == 1 | n == 2) return 1;

    ll fib[n-1];
    fib[0] = 1;
    fib[1] = 1;
    for (int i = 3; i <= n; ++i )
        fib[i-1] = fib[i-2] + fib[i-3];
    return fib[n-1];

}

ll fib1(ll n) {
    if (n < 1) return -1;
    if (n == 1 | n == 2) return 1;

    ll a = 1, b = 1;
    for (int i = 3; i <= n; ++i ) {
        ll c = a + b;
        a = b;
        b = c;
    }
    return b;
}

ll pow(ll m, ll n) {
    ll res = 1;
    for (ll i = 0; i < n; ++i)
        res *= m;
    return res;
}

ll pow1(ll m, ll n) {
    ll res = 1;
    while (n > 0) {
        if (n & 1) res *= m;
                m *= m;
                n >>= 1;
    }
    return res;
}

void mul(ll c[2][2], ll a[2][2], ll b[2][2]) {
    ll t[4];
    t[0] = a[0][0]*b[0][0] + a[0][1]*b[1][0];
```

```
59      t[1] = a[0][0]*b[0][1] + a[0][1]*b[1][1];
60      t[2] = a[1][0]*b[0][0] + a[1][1]*b[1][0];
61      t[3] = a[1][0]*b[0][1] + a[1][1]*b[1][1];
62      c[0][0] = t[0];
63      c[0][1] = t[1];
64      c[1][0] = t[2];
65      c[1][1] = t[3];
66  }
67
68  void pow2(ll s[2][2], ll a[2][2], ll n) {
69      while (n > 0) {
70          if (n & 1) mul(s, s, a);
71          mul(a, a, a);
72          n >>= 1;
73      }
74  }
75
76  ll fib2(ll n) {
77      if (n < 1) return -1;
78      if (n == 1 || n == 2) return 1;
79
80      ll a[2][2] = { {1, 1}, {1, 0} };
81      ll s[2][2] = { {1, 0}, {0, 1} };
82      pow2(s, a, n-2);
83      return s[0][0] + s[0][1];
84  }
85
86  int main() {
87      for (int i = 1; i < 20; ++i)
88          cout << fib1(i) << endl;
89      cout << endl;
90
91      for (int i = 1; i < 20; ++i)
92          cout << fib2(i) << endl;
93
94      return 0;
95  }
```

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

FOLLOW UP

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

在一个N*N矩阵的左上角坐着一个机器人，它只能向右运动或向下运动。那么， 机器人运动到右下角一共有多少种可能的路径?

进一步地，

如果对于其中的一些格子，机器人是不能踏上去的。设计一种算法来获得所有可能的路径。

Solution:

为了一般化这个问题，我们假设这个矩阵是m*n的，左上角的格子是(1, 1)， 右下角的坐标是(m, n)。

解法一

这个题目可以递归来解，如何递归呢? 首先，我们需要一个递推公式， 对于矩阵中的格子(i, j)，假设从(1, 1)到它的路径数量为path(i, j)， 那么，有:

path(i, j) = path(i-1, j) + path(i, j-1)

很好理解，因为机器人只能向右或向下运动，因此它只能是从(i-1, j)或(i, j-1) 运动到(i, j)的，所以路径数量也就是到达这两个格子的路径数量之和。然后， 我们需要一个初始条件，也就是递归终止条件，是什么呢? 可以发现， 当机器人在第一行时，不论它在第一行哪个位置，从(1, 1)到达那个位置都只有一条路径， 那就是一路向右；同理，当机器人在第一列时，也只有一条路径到达它所在位置。 有了初始条件和递推公式，我们就可以写代码了，如下:

```
ll path(ll m, ll n){
    if(m == 1 || n == 1) return 1;
    else return path(m-1, n) + path(m, n-1);
}
```

ll是数据类型long long。

解法二

如果用纯数学的方法来解这道题目，大概也就是个高中排列组合简单题吧。机器人从(1, 1)走到(m, n)一定要向下走m-1次，向右走n-1次，不管这过程中是怎么走的。因此，一共可能的路径数量就是从总的步数(m-1+n-1)里取出(m-1)步，作为向下走的步子，剩余的(n-1)步作为向右走的步子。

C(m-1+n-1, m-1)=(m-1+n-1)! / ((m-1)! * (n-1)!)

代码如下：

```
ll fact(ll n){
    if(n == 0) return 1;
    else return n*fact(n-1);
}
ll path1(ll m, ll n){
    return fact(m-1+n-1)/(fact(m-1)*fact(n-1));
}
```

对于第二问，如果有一些格子，机器人是不能踏上去的(比如说放了地雷XD)，那么，我们如何输出它所有可能的路径呢?

让我们先来考虑简单一点的问题，如果我们只要输出它其中一条可行的路径即可，那么我们可以从终点(m, n)开始回溯，遇到可走的格子就入栈，如果没有格子能到达当前格子，当前格子则出栈。最后到达(1, 1)时，栈中正好保存了一条可行路径。代码如下：

```
bool get_path(int m, int n){
    point p; p.x=n; p.y=m;
    sp.push(p);
    if(n==1 && m==1) return true;
    bool suc = false;
    if(m>1 && g[m-1][n])
        suc = get_path(m-1, n);
    if(!suc && n>1 && g[m][n-1])
        suc = get_path(m, n-1);
    if(!suc) sp.pop();
    return suc;
}
```

其中二维数组g表示的是M*N的矩阵，元素为1表示该位置可以走，为0表示该位置不可走。这个只能得到其中一条可行路径，但题目是要求我们找到所有可行路径，并输出。这样的话，又该怎么办呢? 我们从(1, 1)开始，如果某个格子可以走，我们就将它保存到路径数组中；如果不能走，则回溯到上一个格子，继续选择向右或者向下走。当机器人走到右下角的格子(M, N)时，即可输出一条路径。然后程序会退出递归，回到上一个格子，找寻下一条可行路径。代码如下：

```
void print_paths(int m, int n, int M, int N, int len){
    if(g[m][n] == 0) return;
    point p; p.x=n; p.y=m;
    vp[len++] = p;
    if(m == M && n == N){
        for(int i=0; i<len; ++i)
            cout<<"("<<vp[i].y<<", "<<vp[i].x<<")"<<" ";
        cout<<endl;
    }
    else{
        print_paths(m, n+1, M, N, len);
        print_paths(m+1, n, M, N, len);
    }
}
```

程序使用的输入样例8.2.in如下：

```
3 4
1 1 1 0
0 1 1 1
1 1 1 1
```

输出路径如下：

```
one of the paths:
(1, 1) (1, 2) (1, 3) (2, 3) (2, 4) (3, 4)
all paths:
(1, 1) (1, 2) (1, 3) (2, 3) (2, 4) (3, 4)
(1, 1) (1, 2) (1, 3) (2, 3) (3, 3) (3, 4)
(1, 1) (1, 2) (2, 2) (2, 3) (2, 4) (3, 4)
(1, 1) (1, 2) (2, 2) (2, 3) (3, 3) (3, 4)
(1, 1) (1, 2) (2, 2) (3, 2) (3, 3) (3, 4)
```

完整代码如下：

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

typedef long long ll;

struct point {
    int x;
    int y;
};
stack<point> sp;
const int MAXN = 20;
int g[MAXN][MAXN];
point vp[MAXN*MAXN];

ll path(ll m, ll n) {
    if (m == 1 || n == 1) return 1;
    else return path(m-1, n) + path(m, n-1);
}

ll fact(ll n) {
    if (n == 0) return 1;
    else return n*fact(n-1);
}

ll path1(ll m, ll n) {
    return fact(m-1+n-1)/(fact(m-1)*fact(n-1));
}

bool get_path(int m, int n) {
    point p; p.x = m; p.y = n;
    sp.push(p);
    if (m == 1 && n == 1) return true;

    bool suc = false;
    if (m > 1 && g[m-1][n])
        suc = get_path(m-1, n);
    if (!suc && n>1 && g[m][n-1])
        suc = get_path(m, n-1);
    if (!suc) sp.pop();
    return suc;
}
```

```
44
45  void print_paths(int m, int n, int M, int N, int len) {
46      if (g[m][n] == 0) return;
47      point p; p.x = m; p.y = n;
48      vp[len++] = p;
49      if (m == M && n == N) {
50          for (int i = 0; i < len; ++i)
51              cout << "(" << vp[i].x << ", " << vp[i].y << ")" << " ";
52          cout << endl;
53      } else {
54          print_paths(m+1, n, M, N, len);
55          print_paths(m, n+1, M, N, len);
56      }
57  }
58
59  int main() {
60      freopen("ch88.2.in", "r", stdin);
61
62      for (int i = 1; i < 10; ++i)
63          cout << path(i, i) << endl;
64      cout << endl;
65
66      for (int i = 1; i < 10; ++i)
67          cout << path1(i, i) << endl;
68      cout << endl;
69
70      int M, N;
71      cin >> M >> N;
72      for (int i = 1; i <= M; ++i)
73          for (int j = 1; j <= N; ++j)
74              cin >> g[i][j];
75      cout << "one of the paths: " << endl;
76      get_path(M, N);
77
78      while( !sp.empty() ) {
79          point p = sp.top();
80          cout << "(" << p.x << ", " << p.y << ")" << " ";
81          sp.pop();
82      }
83      cout << endl << "all paths: " << endl;
84      print_paths(1, 1, M, N, 0);
85
86      fclose(stdin);
87      return 0;
88  }
```

8.3 Write a method that returns all subsets of a set.

写一个函数返回一个集合中的所有子集。

Solution:

对于一个集合，它的子集一共有2n 个(包括空集和它本身)。它的任何一个子集， 我们都可以理解为这个集合本身的每个元素是否出现而形成的一个序列。比如说， 对于集合1, 2, 3，空集表示一个元素都没出现，对应0, 0, 0； 子集1, 3，表示元素2没出现(用0表示)，1，3出现了(用1表示)，所以它对应 1, 0, 1。这样一来，我们发现，1, 2, 3的所有子集可以用二进制数000到111 的8个数来指示。泛化一下，如果一个集合有n个元素，那么它可以用0到2n -1 总共2n 个数的二进制形式来指示。每次我们只需要检查某个二进制数的哪一位为1， 就把对应的元素加入到这个子集就OK。

代码如下：

```
1  typedef vector<vector<int> > vvi;
2  typedef vector<int> vi;
3  vvi get_subsets(int a[], int n){ //O(n2^n)
4      vvi subsets;
```

```
    int max = 1<<n;
    for(int i=0; i<max; ++i){
        vi subset;
        int idx = 0;
        int j = i;
        while(j > 0){
            if(j&1){
                subset.push_back(a[idx]);
            }
            j >>= 1;
            ++idx;
        }
        subsets.push_back(subset);
    }
    return subsets;
}
```

解这道题目的另一种思路是递归。这道题目为什么可以用递归? 因为我们能找到比原问题规模小却同质的问题。比如我要求1, 2, 3的所有子集， 我把元素1拿出来，然后去求2, 3的所有子集，2, 3的子集同时也是1, 2, 3 的子集，然后我们把2, 3的所有子集都加上元素1后，又得到同样数量的子集， 它们也是1, 2, 3的子集。这样一来，我们就可以通过求2, 3的所有子集来求 1, 2, 3的所有子集了。而同理，2, 3也可以如法炮制。代码如下:

```
vvi get_subsets1(int a[], int idx, int n){
    vvi subsets;
    if(idx == n){
        vi subset;
        subsets.push_back(subset); //empty set
    }
    else{
        vvi rsubsets = get_subsets1(a, idx+1, n);
        int v = a[idx];
        for(int i=0; i<rsubsets.size(); ++i){
            vi subset = rsubsets[i];
            subsets.push_back(subset);
            subset.push_back(v);
            subsets.push_back(subset);
        }
    }
    return subsets;
}
```

完整代码如下:

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

typedef vector< vector<int> > vvi;
typedef vector<int>  vi;

vvi get_subsets(int a[], int n) {  // O(n*2^n)
    vvi subsets;
    int max = 1 << n;
    for (int i = 0; i < max; ++i) {
        vi subset;
        int idx = 0;
        int j = i;
        while (j > 0) {
         // if (j & 1) subset.push_back(a[idx++]); WRONG, idx++, considered only one condition
            if (j & 1)
                subset.push_back(a[idx]);
```

```
19              ++idx;          // attention
20              j >>= 1;
21          }
22          subsets.push_back(subset);
23      }
24      return subsets;
25  }
26
27  vvi get_subsets1(int a[], int idx, int n) {  // not understanding, difficult !!!
28      vvi subsets;
29      if (idx == n) {
30          vi subset;
31          subsets.push_back(subset);  // empty set
32      } else {
33          vvi rsubsets = get_subsets1(a, idx+1, n);
34          int v = a[idx];
35          for (int i = 0; i < rsubsets.size(); ++i) {
36              vi subset = rsubsets.at(i);
37              subsets.push_back(subset);
38              subset.push_back(v);
39              subsets.push_back(subset);
40          }
41      }
42      return subsets;
43  }
44
45  void print_subsets(vvi subsets) {
46      cout << "Subsets: " << endl;
47      for (int i = 0; i < subsets.size(); ++i) {
48          vi subset = subsets.at(i);
49          cout << "Subset Vector #" << i << ": ";
50          for (int j = 0; j < subset.size(); ++j)
51           // cout << subset.at(j) << " ";
52              cout << subset[j] << " ";    // both works just fine!!!
53          cout << endl;
54      }
55      cout << "End of Subsets!" << endl;
56  }
57
58  int main() {
59      int a[] = {1, 2, 3, 4};
60      vvi sub = get_subsets(a, 4);
61      vvi sub1 = get_subsets1(a, 0, 4);
62      print_subsets(sub);
63      print_subsets(sub1);
64      return 0;
65  }
```

8.4 Write a method to compute all permutations of a string

写一个函数返回一个串的所有排列

Solution:

对于一个长度为n的串，它的全排列共有A(n, n)=n!种。这个问题也是一个递归的问题， 不过我们可以用不同的思路去理解它。为了方便讲解，假设我们要考察的串是"abc"， 递归函数名叫permu。

思路一：

我们可以把串“abc”中的第0个字符a取出来，然后递归调用permu计算剩余的串“bc” 的排列，得到bc, cb。然后再将字符a插入这两个串中的任何一个空位(插空法)， 得到最终所有的排列。比如，a插入串bc的所有(3个)空位，得到abc,bac,bca。 递归的终止条件是什么呢? 当一个串为空，就无法再取出其中的第0个字符了， 所以此时返回一个空的排列。代码如下:

```
typedef vector<string> vs;

vs permu(string s){
    vs result;
    if(s == ""){
        result.push_back("");
        return result;
    }
    string c = s.substr(0, 1);
    vs res = permu(s.substr(1));
    for(int i=0; i<res.size(); ++i){
        string t = res[i];
        for(int j=0; j<=t.length(); ++j){
            string u = t;
            u.insert(j, c);
            result.push_back(u);
        }
    }
    return result; //调用result的拷贝构造函数，返回它的一份copy，然后这个局部变量销毁(与基本类型一样)
}
```

思路二:
我们还可以用另一种思路来递归解这个问题。还是针对串“abc”， 我依次取出这个串中的每个字符，然后调用permu去计算剩余串的排列。 然后只需要把取出的字符加到剩余串排列的每个字符前即可。对于这个例子，程序先取出a，然后计算剩余串的排列得到bc,cb，然后把a加到它们的前面，得到 abc,acb；接着取出b，计算剩余串的排列得到ac,ca，然后把b加到它们前面， 得到bac,bca；后面的同理。最后就可以得到“abc”的全序列。代码如下:

```
vs permu1(string s){
    vs result;
    if(s == ""){
        result.push_back("");
        return result;
    }
    for(int i=0; i<s.length(); ++i){
        string c = s.substr(i, 1);
        string t = s;
        vs res = permu1(t.erase(i, 1));
        for(int j=0; j<res.size(); ++j){
            result.push_back(c + res[j]);
        }
    }
    return result;
}
```

完整代码如下:

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

typedef vector<string> vs;

void print_vs(vs result) {
    cout << "Permutation result size: " << result.size() << endl;

    for (int i = 0; i < result.size(); ++i)
        cout << result[i] << " ";
    cout << endl;
}

```

```
15  vs permu(string s) {
16      vs result;
17      if (s == "") {
18          result.push_back(s);
19          return result;
20      }
21
22      string c = s.substr(0, 1);
23      vs res = permu( s.substr(1) );
24      for (int i = 0; i < res.size(); ++i) {
25          string t = res[i];
26          //result.push_back(t);  // made mistake here, NOT necessary
27          for (int j = 0; j <= t.length(); ++j) {    // one >= length()
28              // string u = t.substr(0,j) + c + t.substr(j);  // equivalent method
29              string u = t;
30              u.insert(j, c);
31              result.push_back(u);
32          }
33      }
34      return result;
35  }
36
37  vs permu1(string s) {
38      vs result;
39      if (s == "") {
40          result.push_back(s);
41          return result;
42      }
43
44      for (int i = 0; i < s.length(); ++i) {
45          string c = s.substr(i,1);  // made mistake s.substr(i, i+1)
46          string rest = s.substr(0,i)+s.substr(i+1);
47       // vs res = permu1( s.substr(0,i)+s.substr(i+1) );  // equivalent
48          string t = s;
49          vs res = permu1( t.erase(i,1) );
50          for (int j = 0; j < res.size(); ++j)
51              result.push_back(c+res[j]);
52      }
53      return result;
54  }
55
56
57  int main() {
58      string s = "abcd";
59      vs result = permu(s);
60      print_vs(result);
61      vs result1 = permu1(s);
62      print_vs(result1);
63      return 0;
64  }
```

8.5 Implement an algorithm to print all valid (e.g., properly opened and closed) combinations of n-pairs of parentheses.
EXAMPLE:
input: 3 (e.g., 3 pairs of parentheses)
output: ((())), (()()), (())(), ()(()), ()()()
实现一个算法打印出n对括号的有效组合。
输入：3 （3对括号）
输出：((())), (()()), (())(), ()(()), ()()()
Solution:

对于括号的组合，要考虑其有效性。比如说，)(， 它虽然也是由一个左括号和一个右括号组成，但它就不是一个有效的括号组合。 那么，怎样的组合是有效的呢? 对于一个左括号，在它右边一定要有一个右括号与之配对， 这样的才能是有效的。所以，对于一个输出，比如说(()())， 从左边起，取到任意的某个位置得到的串，左括号数量一定是大于或等于右括号的数量， 只有在这种情况下，这组输出才是有效的。我们分别记左，右括号的数量为left和right，如下分析可看出，(()())是个有效的括号组合。

```
(, left = 1, right = 0
((, left = 2, right = 0
((), left = 2, right = 1
(()(, left = 3, right = 1
(()(), left = 3, right = 2
(()()), left = 3, right = 3
```

这样一来，在程序中，只要还有左括号，我们就加入输出串，然后递归调用。 当退出递归时，如果剩余的右括号数量比剩余的左括号数量多，我们就将右括号加入输出串。 直到最后剩余的左括号和右括号都为0时，即可打印一个输出串。代码如下:

```
void print_pare(int l, int r, char str[], int cnt){
    if(l<0 || r<l) return;
    if(l==0 && r==0){
        for(int i=0; i<cnt; ++i){
            cout<<str[i];
        }
        cout<<", ";
    }
    else{
        if(l > 0){
            str[cnt] = '(';
            print_pare(l-1, r, str, cnt+1);
        }
        if(r > l){
            str[cnt] = ')';
            print_pare(l, r-1, str, cnt+1);
        }
    }
}
```

完整代码如下:

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

void print_pare(int left, int right, char str[], int cnt) {
    if (left < 0 || right < left) return;
    if (left == 0 && right == 0) {
        for (int i = 0; i < cnt; ++i)
            cout << str[i];
        cout << ", ";
    } else {
        if (left > 0) {
            str[cnt] = '(';
           print_pare(left-1, right, str, cnt+1);
        }
        if (right > left) {
            str[cnt] = ')';
            print_pare(left, right-1, str, cnt+1);
        }
    }
}

int main() {
```

```
23      int cnt = 3;
24      char str[2*cnt];
25      print_pare(cnt, cnt, str, 0);
26      return 0;
27  }
```

8.6 Implement the "paint fill" function that one might see on many image editing programs. That is, given a screen (represented by a 2-dimensional array of Colors), a point, and a new color, fill in the surrounding area until you hit a border of that color.

实现图像处理软件中的"填充"函数，给定一块区域(可以不规则)，一个种子点和一种新颜色， 填充这块区域，直到到达这块区域的边界(边界是用这种新颜色画的一条封闭曲线)

Solution:

两种思路，一种递归，一种迭代。基本上能递归的都能迭代， 只不过有时迭代版本不是太好写。如果我没有记错，填充函数在计算机图像学的书里是会讲的。给你一个种子点和一个颜色， 从这个种子点开始去把这块区域都填充上这种颜色。我们可以从这个种子点开始着色， 然后递归调用本函数去给它的上，下，左，右四个点着色。 递归停止条件是碰到这个区域的边界(边界是一条同种颜色绘制的封闭曲线)。比如说， 我先用绿色画了一个圆形(或是其它任意形状)，然后在这个形状中间再用绿色点击一下， 那么这块区域就被填充成绿色了。下面是代码:

```
1   bool paint_fill(color **screen, int m, int n, int x, int y, color c){
2       if(x<0 || x>=m || y<0 || y>=n) return false;
3       if(screen[x][y] == c) return false;
4       else{
5           screen[x][y] = c;
6           paint_fill(screen, m, n, x-1, y, c);
7           paint_fill(screen, m, n, x+1, y, c);
8           paint_fill(screen, m, n, x, y-1, c);
9           paint_fill(screen, m, n, x, y+1, c);
10      }
11      return true;
12  }
```

迭代版本也不难，用BFS即可。每次给一个点着色后，就把它周围四个点放入队列， 只要队列不为空，就一直处理。这里需要注意一点的是，如果遇到一个点已经是新颜色， (比如上面说的绿色)，那么我就不用处理它而且不需要把它周围四个点入队。 如果不注意这点，那么填充色就会不顾边界直接把整张图片都填充成同一种颜色。 迭代版本的就不写了。以下是完整代码:

```
1   #include <iostream>
2   #include <cstdio>
3   using namespace std;
4
5   enum color {red, yellow, blue, green};
6
7   bool paint_fill(color **screen, int m, int n, int x, int y, color c) {
8       if (x < 0 || x >= m || y < 0 || y >= n) return false;
9
10      if (screen[x][y] == c) return false;
11      else {
12          screen[x][y] = c;
13          paint_fill(screen, m, n, x-1, y, c);
14          paint_fill(screen, m, n, x+1, y, c);
15          paint_fill(screen, m, n, x, y-1, c);
16          paint_fill(screen, m, n, x, y+1, c);
17      }
18      return true;
19  }
20
21  int main() {
22      freopen("ch88.6.in", "r", stdin);
23      int m, n;
24      cin >> m >> n;
```

```
25      color **screen = new color*[m];
26      for (int i = 0; i < m; ++i)
27          screen[i] = new color[n];
28
29      for (int i = 0; i < m; ++i)
30          for (int j = 0; j < n; ++j) {
31              int t;
32              cin >> t;
33              screen[i][j] = (color)t;
34          }
35      paint_fill(screen, 5, 5, 1, 2, green);
36
37      for (int i = 0; i < 5; ++i) {
38          for (int j = 0; j < 5; ++j)
39              cout << screen[i][j] << "␣";
40          cout << endl;
41      }
42      fclose(stdin);
43      return 0;
44  }
```

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

我们有25分，10分，5分和1分的硬币无限个。写一个函数计算组成n分的方式有几种?

Solution:

一开始，我觉得使用递归不断地累加四种币值的硬币，当累加到n分，组成方式就加1。 最后就能得到一共有多少种组合方式，听起来挺正确的，然后写了以下代码:

```
1   int cnt = 0;
2   void sumn(int sum, int n){
3       if(sum >= n){
4           if(sum == n) ++cnt;
5           return;
6       }
7       else{
8           sumn(sum+25, n);
9           sumn(sum+10, n);
10          sumn(sum+5, n);
11          sumn(sum+1, n);
12      }
13  }
```

但，这是错误的。问题出在哪? 有序与无序的区别! 这个函数计算出来的组合是有序的， 也就是它会认为1,5和5,1是不一样的，导致计算出的组合里有大量是重复的。 那要怎么避免这个问题? 1,5和5,1虽然会被视为不一样，但如果它们是排好序的， 比如都按从大到小排序，那么就都是5,1了，这时就不会重复累计组合数量。 可是我们总不能求出答案来后再搞个排序吧，多费劲。这时我们就可以在递归上做个手脚， 让它在计算的过程中就按照从大到小的币值来组合。比如， 现在我拿了一个25分的硬币，那下一次可以取的币值就是25,10,5,1；如果我拿了一个 10分的，下一次可以取的币值就只有10,5,1了；这样一来，就能保证，同样的组合， 我只累计了一次，改造后的代码如下:

```
1   int cnt = 0;
2   void sumN(int sum, int c, int n){
3       if(sum >= n){
4           if(sum == n) ++cnt;
5           return;
6       }
7       else{
8           if(c >= 25)
9               sumN(sum+25, 25, n);
10          if(c >= 10)
11              sumN(sum+10, 10, n);
```

```
12          if(c >= 5)
13              sumN(sum+5, 5, n);
14          if(c >= 1)
15              sumN(sum+1, 1, n);
16      }
17  }
```

有个全局变量总觉得看起来不好看，于是我们可以把这个全局变量去掉， 让函数来返回这个组合数目，代码如下：

```
1   int sum_n(int sum, int c, int n){
2       int ways = 0;
3       if(sum <= n){
4           if(sum == n) return 1;
5           if(c >= 25)
6               ways += sum_n(sum+25, 25, n);
7           if(c >= 10)
8               ways += sum_n(sum+10, 10, n);
9           if(c >= 5)
10              ways += sum_n(sum+5, 5, n);
11          if(c >= 1)
12              ways += sum_n(sum+1, 1, n);
13      }
14      return ways;
15  }
```

CTCI原文中给出的解法如下：

```
1   int make_change(int n, int denom){
2       int next_denom = 0;
3       switch(denom){
4       case 25:
5           next_denom = 10;
6           break;
7       case 10:
8           next_denom = 5;
9           break;
10      case 5:
11          next_denom = 1;
12          break;
13      case 1:
14          return 1;
15      }
16      int ways = 0;
17      for(int i=0; i*denom<=n; ++i)
18          ways += make_change(n-i*denom, next_denom);
19      return ways;
20  }
```

也是从币值大的硬币开始，每次取i个(i乘以币值要小于等于n)， 然后接着去取币值比它小的硬币，这时就是它的一个子问题了，递归调用。 具体来怎么来理解这个事呢？这样，比如我要凑100分，那我先从25分开始， 我先取0个25分，然后递归调用去凑100分；再然后我取1个25分，然后递归调用去凑100-25 =75分；接着我取2个25分，递归调用去凑100-25*2=50分.....这些取了若干个 25分然后再去递归调用，取的就是10分了。一直这样操作下去，我们就会得到， 由若干个25，若干个10分，若干个5分和若干个1分组成的100分，而且， 这里面每种币值的数量都可以为0。

完整代码如下：

```
1   #include <iostream>
2   using namespace std;
3
4   int cnt = 0;
5   void sumn(int sum, int c, int n) {
6       if (sum >= n) {
```

```
7           if (sum == n) ++cnt;
8           return;
9       } else {
10          if (c >= 25) sumn(sum+25, 25, n);
11          if (c >= 10) sumn(sum+10, 10, n);
12          if (c >= 5) sumn(sum+5, 5, n);
13          if (c >= 1) sumn(sum+1, 1, n);
14      }
15  }
16
17  int sum_n(int sum, int c, int n) {
18      int ways = 0;
19      if (sum <= n) {
20          if (sum == n) return 1;
21          if (c >= 25) ways += sum_n(sum+25, 25, n);
22          if (c >= 10) ways += sum_n(sum+10, 10, n);
23          if (c >=  5) ways += sum_n(sum+ 5,  5, n);
24          if (c >=  1) ways += sum_n(sum+ 1,  1, n);
25      }
26      return ways;
27  }
28
29  int make_change(int n, int denom) {
30      int next_denom = 0;
31      switch (denom) {
32      case 25:
33          next_denom = 10;
34          break;
35      case 10:
36          next_denom = 5;
37          break;
38      case 5:
39          next_denom = 1;
40          break;
41      case 1:
42          return 1;
43      }
44      int ways = 0;
45      for (int i = 0; i*denom <= n; ++i)
46          ways += make_change(n-i*denom, next_denom);
47      return ways;
48  }
49
50  int main() {
51      int n = 10;
52      sumn(0, 25, n);
53      cout << "cnt: " << cnt << endl;
54
55      cout << make_change(n, 25) << endl;
56      cout << sum_n(0, 25, n) << endl;
57      return 0;
58  }
```

8.8 Write an algorithm to print all ways of arranging eight queens on a chess board so that none of them share the same row, column or diagonal.

经典的八皇后问题，即在一个8*8的棋盘上放8个皇后，使得这8个皇后无法互相攻击(任意2个皇后不能处于同一行，同一列或是对角线上)，输出所有可能的摆放情况。

Solution:

8皇后是个经典的问题，如果使用暴力法，每个格子都去考虑放皇后与否，一共有264 种可能。所以暴力法并不是个

好办法。由于皇后们是不能放在同一行的， 所以我们可以去掉“行”这个因素，即我第1次考虑把皇后放在第1行的某个位置， 第2次放的时候就不用去放在第一行了，因为这样放皇后间是可以互相攻击的。 第2次我就考虑把皇后放在第2行的某个位置，第3次我考虑把皇后放在第3行的某个位置， 这样依次去递归。每计算1行，递归一次，每次递归里面考虑8列， 即对每一行皇后有8个可能的位置可以放。找到一个与前面行的皇后都不会互相攻击的位置， 然后再递归进入下一行。找到一组可行解即可输出，然后程序回溯去找下一组可靠解。

我们用一个一维数组来表示相应行对应的列，比如c[i]=j表示， 第i行的皇后放在第j列。如果当前行是r，皇后放在哪一列呢？c[r]列。 一共有8列，所以我们要让c[r]依次取第0列，第1列，第2列.....一直到第7列， 每取一次我们就去考虑，皇后放的位置会不会和前面已经放了的皇后有冲突。 怎样是有冲突呢？同行，同列，对角线。由于已经不会同行了，所以不用考虑这一点。 同列：c[r]==c[j]; 同对角线有两种可能，即主对角线方向和副对角线方向。 主对角线方向满足，行之差等于列之差：r-j==c[r]-c[j]; 副对角线方向满足， 行之差等于列之差的相反数：r-j==c[j]-c[r]。只有满足了当前皇后和前面所有的皇后都不会互相攻击的时候，才能进入下一级递归。

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

int c[20], n = 8, cnt = 0;

void print() {
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 8; ++j) {
            if (j == c[i]) cout << "1 ";
            else cout << "0 ";
        }
        cout << endl;
    }
    cout << endl;
}

void search(int r) {
    if (r == n) {
        print();
        ++cnt;
        return;
    }
    for (int i = 0; i < n; ++i) {
        c[r] = i;
        int ok = 1;
        for (int j = 0; j < r; ++j) {
            if ( c[j] == i || (r-j == c[r]-c[j]) || (r-j == c[j]-c[r]) ) {
                ok = 0;
                break;
            }
        }
        if (ok) search(r+1);
    }
}

int main() {
    search(0);
    cout << cnt << endl;
    return 0;
}
```

Chapter 9

Sorting and Searching

9.1 You are given two sorted arrays, A and B, and A has a large enough buffer at the end to hold B. Write a method to merge B into A in sorted order.

A和B是两个有序数组(假设为递增序列)，而且A的长度足以放下A和B中所有的元素， 写一个函数将数组B融入数组A，并使其有序。

Solution:

最简单的方法是开一个大小可以容纳A和B的数组C，然后像归并排序中合并两个数组一样， 按照大小顺序把数组A和数组B的元素一一地放入数组C，然后再将数组C拷贝给A。 这种方法额外地使用了O(n)的空间，显然这是没有必要的开销。由于A的大小已经足够用了， 所以直接在A上直接操作即可。

可是如果我们思维定势地对比数组A和数组B，每次取小的元素放入数组A， 这样就会发现，要放入的位置上正放着有用的元素。处理起来就麻烦了。

相反，如果我们从A和B的尾部元素开始对比，每次取大的元素放在数组A的尾部， 这样一来，要放入的位置就不会出现上面的冲突问题。当对比结束时， 即可得到一个排好序的数组A。代码如下：

```
1  void merge(int a[], int b[], int n, int m){
2      int k = n + m - 1;
3      int i = n - 1, j = m - 1;
4      while(i>=0 && j>=0){
5          if(a[i] > b[j]) a[k--] = a[i--];
6          else a[k--] = b[j--];
7      }
8      while(j>=0) a[k--] = b[j--];
9  }
```

对比结束后，要检查数组B中是否还有元素，有的话要将它们拷贝到数组A。 我们并不需要检查数组A，因为如果数组A还有元素，说明while循环是因为数组B 中没有元素了才退出的。而A中的元素本来就是有序且就位于数组A中，所以不需要再管它。 如果A和B中的元素个数为n和m，则该算法的时间复杂度为O(n+m)。

让我们再加点限制条件，如果两个有序的序列并且没有额外的空间，那要怎么排序。 比如对于数组A，它的前半段和后半段分别有序，不使用额外的空间怎么使A整体有序。

首先，不可避免的我们还是要将两个有序部分中的元素拿出来对比。 我们先拿出前半段的第一个元素和后半段的第一个元素进行对比， 如果后半段的第一个元素要小，就将它们交换。由于这两个元素是各自序列的最小值， 这一对比就将整个数组A的最小值取出放在了正确的位置上。然后呢? 交换到后半段的那个值怎么办? 不理它，不太合适吧。我们可以通过两两对比， 把它移动到后半段的某个位置，使后半段保持有序。接下来呢? 我们取出前半段的第2个元素(第1个元素已经放在它正确的位置上，不用理它了)， 还是和后半段的第1个元素对比，这一对比中较小的就会是整个数组中第2小的元素， 如果是后半段那个元素较小，则交换它们，然后仍然移动后半段使其保持有序。这样不断进行下去，当把前半段的元素都遍历操作一遍，就会将小的元素都移动到前半段， 并且是有序的。而大的元素都在后半段且也是有序的。排序结束。

代码如下：

```
1  #include <iostream>
2  using namespace std;
3
4  void merge(int a[], int b[], int n, int m) {   // O(m+n)
```

```
5       int k = m + n -1;
6       int i = n-1, j = m-1;
7
8       while (i >= 0 && j >= 0) {
9           if (a[i] > b[j]) a[k--] = a[i--];
10          else a[k--] = b[j--];
11      }
12      while (j >= 0) a[k--] = b[j--];
13  }
14
15  void swap(int &a, int &b) {
16      a = a^b;
17      b = a^b;
18      a = a^b;
19  }
20
21  void merge(int a[], int begin, int mid, int end) {
22      for (int i = begin; i <= mid; ++i) {
23          if (a[i] > a[mid+1]) {
24              swap(a[i], a[mid+1]);
25
26              for (int j = mid+1; j < end; ++j) {
27                  if (a[j] <= a[j+1]) break;
28                  swap(a[j], a[j+1]);
29              }
30          }
31      }
32  }
33
34  int main() {
35      int a[15] = {1, 5, 6, 7};
36      int b[] = {2, 3, 4, 8, 10, 12, 14};
37      int n = 4, m = 7;
38      merge(a, b, 4, 7);
39      for (int i = 0; i < m+n; ++i)
40          cout << a[i] << " ";
41      cout << endl;
42
43      int a1[10] = {8, 9, 11, 15, 17, 1, 3, 5, 12, 18};
44      merge(a1, 0, 4, 9);
45      for (int i = 0; i < 10; ++i)
46          cout << a1[i] << " ";
47      cout << endl;
48      return 0;
49  }
```

9.2 Write a method to sort an array of strings so that all the anagrams are next to each other.

写一个函数对字符串数组排序，使得所有的变位词都相邻。

Solution:

首先，要弄清楚什么是变位词。变位词就是组成的字母相同，但顺序不一样的单词。 比如说：live和evil就是一对变位词。OK，那么这道题目的意思就很清楚了， 它并不要求我们将字符串数组中的字符串按字典序排序，否则我们直接调用STL中的sort 函数就可以了。它要求我们在排序的过程中，按照变位词的准则来排序。 这种情况下，我们还是可以调用sort函数，不过要自己写一个对比函数。 一般情况下我们如果要排序一个长度为n的数组A，我们可以这样调用sort：

sort(A, A+n);

但如果我们有一个数组P，里面每个元素都是一个结构体：person，我们想按照person 这个结构体中年龄age来排序，这时候我们就需要自己写一个对比函数cmp：

```
1   bool cmp(person p1, person p2){
```

```
    return p1.age < p2.age;
}
```

然后这样调用sort函数:

sort(P, P+n, cmp);

OK，回到我们的题目，我们的对比函数需要将两个串先按字典序排序，然后再比较， 这样一来，变位词经过字典序排序后就是一样的了。当调用sort函数时将会被排在一起。

代码如下:

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

bool cmp(string s1, string s2) {
    sort(&s1[0], &s1[0]+s1.length());
    sort(&s2[0], &s2[0]+s2.length());
    return s1 < s2;
}

int main() {
    string s[] = {"abc", "cba", "bca", "xyz", "hijklmn", "fg", "gf"};
    sort(s, s+7, cmp);
    for (int i = 0; i < 7; ++i)
        cout << s[i] << endl;
    return 0;
}
```

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

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

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

一个数组有n个整数，它们排好序(假设为升序)但被旋转了未知次， 即每次把最右边的数放到最左边。给出一个O(log n)的算法找到特定的某个元素。

输入：在数组(15 16 19 20 25 1 3 4 5 7 10 14)中找出5

输出：8（5在数组中的下标）

Solution:

题目中提到了几个关键的词：有序，O(log n)。我们马上能联想到的就是二分查找算法。 但简单的二分查找肯定不行，因为这个数组被旋转了。也就是该数组前面一段有序， 后面一段有序，且前面那段的数要大于等于后面那段的数(本题考虑递增序列，递减同理)。 因此，我们需要对二分查找算法做一下修改，来解决这个具体的问题。

首先，我们来看一下函数原型:

int search(int a[], int low, int high, int x);

参数和二分查找一样，数组a，下界low，上界high及要查找的数x。当low <= high时， 我们会去求它们中间的那个数，然后与x对比，如果相同，就返回下标:

```
int mid = low + (high - low)/2;
if(a[mid] == x) return mid;
```

如果a[mid]不等于x，就要分情况讨论了。我们把旋转后的数组分为前半段和后半段， 两段分别都是有序的，且前半段的数都大于后半段的数。假如a[mid]落在前半段 (即a[mid]>=a[low])，这时如果x的值是位于a[low]和a[mid]之间，则更新high = mid - 1；否则更新low = mid + 1。假如a[mid]落在后半段 (即a[mid]<a[low])，这时如果x的值是位于a[mid]和a[low]之间， 则更新low = mid + 1；否则更新high = mid - 1。

代码如下:

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

int find(int a[], int low, int high, int val) {
    if (low <= high) {
        int mid = (low + high) >> 1;  // / 2
```

```
7           if (a[mid] == val) return mid;
8           find(a, low, mid-1, val);
9           find(a, mid+1, high, val);
10      }
11  }
12
13  int search(int a[], int low, int high, int x) {
14      while (low <= high) {
15          int mid = low + (high - low) / 2;  // different from >> 1
16          if (a[mid] == x) return mid;
17
18          if (a[mid] >= a[low]) {
19              if (a[low] <= x && x < a[mid])
20                  high = mid -1;
21              else
22                  low = mid + 1;
23          } else {
24              if (x > a[mid] && x < a[low])
25                  low = mid + 1;
26              else
27                  high = mid -1;
28          }
29      }
30      return -1;
31  }
32
33  int main() {
34      int a[] = {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14};
35      cout << find(a, 0, 11, 5) << endl;
36      cout << search(a, 0, 11, 5) << endl;
37      return 0;
38  }
```

对于有重复元素的数组，上面的算法失效。比如说代码中对于数组b， 它也是一个有序数组然后旋转了若干次。但是调用search函数将返回-1， 也就是查找失败。

9.4 If you have a 2 GB file with one string per line, which sorting algorithm would you use to sort the file and why?

如果你有个2GB的文件，其中每一行是一个字符串，你会使用哪种算法来排序，为什么?

Solution:

当面试官说到2GB文件的时候，他其实就是在暗示你， 他并不希望一次性把所有的数据都载入内存。这样子的话，我们要怎么做呢? 我们每次就将部分数据载入内存就好了。

算法:

首先我们要了解，可以用的内存有多大? 假设我们有X MB的内存可用。

1. 我们将文件分为K份，其中X*K=2GB。每次取其中一份载入到内存中， 用O(nlog n)的算法排序，然后再保存到外部文件。
2. 载入下一份并排序
3. 当我们将K份小文件都排好序，合并它们。

上面的算法就是外排序，步骤3又称为N路归并。

使用外排序是由于数据太大了，无法一次全部加载到内存中，所以需要借助磁盘进行存储， 每次只从磁盘中加载一部分数据进入内存，进行排序。

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

Example: find “ball” in [“at”, “”, “”, “”, “ball”, “”, “”, “car”, “”,“”, “dad”, “”, “”] will return 4

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

给你一个排好序的并且穿插有空字符串的字符串数组，写一个函数找到给定字符串的位置。

例子：在字符串数组 [“at”, “”, “”, “”, “ball”, “”, “”, “car”, “”,“”, “dad”, “”, “”] 中找到"ball"，返回下标4.

例子：在字符串数组 [“at”, “”, “”, “”, “”, “ball”, “car”, “”, “”, “dad”, “”, “”] 中找到"ballcar"，查找失败，返回-1.

Solution:

字符串数组已经是有序的了，所以，还是可以利用二分查找来找到指定的字符串。 当然了，由于数组中有空字符串，因此还要加些额外的处理，否则无法对比大小。 我们可以这样来处理，如果要对比的那个元素为空字符串，就一直向右移动， 直到字符串不为空或是位置已经超出了high下标。如果位置已经超出high下标， 就在[low, mid-1]这一段查找；如果没有超出high下标，那就和要查找的x进行对比。 相等时返回下标，不等时再根据比较出的大小决定到哪一半去查找。

代码如下：

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

int search(string s[], int low, int high, string x) {
    if (x == "") return -1;
    while (low <= high) {
        int mid = (low + high) >> 1;
        int t = mid;
        while ( s[t] == "" && t <= high)  ++t;
        if (t > high) high = mid -1;
        else {
            if (s[t] == x) return t;
            else if (s[t] < x) low = t + 1;
            else
                high = mid -1;
        }
    }
    return -1;
}

int main() {
    string s[] = {"at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""};
    cout << search(s, 0, 12, "ball") << endl;
    return 0;
}
```

9.6 Given a matrix in which each row and each column is sorted, write a method to find an element in it.

给出一个矩阵，其中每一行和每一列都是有序的，写一个函数在矩阵中找出指定的数。

Solution:

我们假设这个矩阵每一行都是递增的，每一列也都是递增的，并把这些数据存入文件 9.6.in(如下)，其中开头的两个数5 5表示该矩阵是5*5的。

```
5 5
1 2 3 4 5
3 7 8 9 11
5 9 10 17 18
7 12 15 19 23
9 13 16 20 25
```

这个矩阵是有序的，因此我们要利用这一点，当矩阵中元素和要查找的元素对比后， 如果相等，我们返回下标；如果不等，我们就排除掉一些元素，而不仅仅是对比的元素。 我们从矩阵的四个角的元素入手看看，有什么特点。左上角是最小的， 因为矩阵向右是递增的，向下也是递增的。同理，右下角是最大的。让我们来看看右上角， 设要查找的元素为x，比如x比右上角元素5大，那么它也会比第一行的其它元素都大。 因此可以去掉第一行；如果它比右上角元素小，那么它也会比最后一列的元素都小， 因此可以去掉最后一列；然后这样不断地比下去，只需要O(m+n)的时间就查找完。 对于左下角的元素，也有一样的特点。就不再分析了。

代码如下：

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

```

```
int d[20][20];
int search(int m, int n, int val) {
    int r = 0, c = n-1;
    while (r < m && c >= 0) {
        if (d[r][c] == val) return r*n+c;
        else if (d[r][c] < val) ++r;
        else --c;
    }
    return -1;
}

int main() {
    freopen("ch99.6.in", "r", stdin);
    int m, n;
    cin >> m >> n;
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j)
            cin >> d[i][j];

    int k = search(m, n, 13);
    if (k == -1) cout << "Not found!!!" << endl;
    else cout << "Position: " << k/n << " " << k%n << endl;
    fclose(stdin);
    return 0;
}
```

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

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

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

马戏团设计了这样一个节目：叠罗汉。一群人往上叠，每个人都踩在另一个人的肩膀上。 要求上面的人要比下面的人矮而且比下面的人轻。给出每个人的身高和体重， 写一个函数计算叠罗汉节目中最多可以叠多少人?

输入(身高 体重)：(65, 100) (70, 150) (56, 90) (75, 190) (60, 95) (68, 110)

输出：最多可叠人数：6 （从上到下是：(56, 90) (60,95) (65,100) (68,110) (70,150) (75,190)）

Solution:

给定了(身高 体重)序列，要求我们排序。不过由于要排序的对象是一个结构， 我们可以先按其中的一个指标进行排序，比如说身高。身高排好序后， 身高这个指标就都是满足叠罗汉的要求了，剩下的就看体重。 我们就只要在体重那个维度找到最长的递增子序列就可以了。

我们先定义一个结构体：

```
const int maxn = 100;
struct person{
    int h, w;
};
person p[maxn];
```

然后为了使STL中的sort函数能对这个结构体先按身高排序，当身高相等时按体重排序， 我们还需要写一个对比函数作为sort的参数：

```
bool cmp(person p1, person p2){
    if(p1.h == p2.h) return p1.w < p2.w;
    else return p1.h < p2.h;
}
```

这样一来，当我们调用sort函数，就能达到先对身高排序，若身高相等时对体重排序的要求。

sort(p, p+n, cmp);

排好序后，只需要求体重序列的最长递增子序列(LIS)即可。关于求LIS，可以参考文章： 动态规划：从新手到专家

```
int lis(person p[], int n){
    int k = 1;
    d[0] = p[0].w;
    for(int i=1; i<n; ++i){
        if(p[i].w >= d[k-1]) d[k++] = p[i].w;
        else{
            int j;
            for(j=k-1; j>=0 && d[j]>p[i].w; --j);//用二分可将复杂度降到O(nlogn)
            d[j+1] = p[i].w;
        }
    }
    return k;
}
```

完整代码如下：

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

const int maxn = 100;
struct person {
    int h;
    int w;
};
person p[maxn];
int d[maxn];

bool cmp(person p1, person p2) {
    if (p1.h == p2.h) return p1.w < p2.w;
    else return p1.h < p2.h;
}

int lis(person p[], int n) {
    d[0] = p[0].w;
    int k = 1;
    for (int i = 1; i < n; ++i) {
        if (p[i].w >= d[k-1])
            d[k++] = p[i].w;
        else { // binary to get O(nlogn)  NOT understanding this partx
            int j;
            for (j = k-1; j >= 0 && d[j] > p[i].w; --j);
            d[j+1] = p[i].w;
        }
    }
    return k;
}

int main() {
    freopen("ch99.7.in", "r", stdin);
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i)
        cin >> p[i].h >> p[i].w;
    sort(p, p+n, cmp);
    cout << lis(p, n) << endl;
    fclose(stdin);
    return 0;
}
```

Chapter 10

Mathematical

10.1 You have a basketball hoop and someone says that you can play 1 of 2 games.

Game #1: You get one shot to make the hoop.

Game #2: You get three shots and you have to make 2 of 3 shots.

If p is the probability of making a particular shot, for which values of p should you pick one game or the other?

你有一个篮球架，可以在以下游戏中选择一个来玩。

游戏1：投一次球，进了算赢。

游戏2：投三次球，至少要进2个才算赢。

如果命中率是p，那么p是什么值时你会选游戏1来玩？或p是什么值时你会选择游戏2？

Solution:

命中率是p，那么对于游戏1，赢的概率是p。对于游戏2，至少进2个球才算赢， 那么赢的概率为：

C(2,3)p$\hat{}$2(1-p) + p$\hat{}$3 = 3p$\hat{}$2 - 2p$\hat{}$3

哪个游戏赢的概率大我们就选哪个游戏，所以当

p > 3p$\hat{}$2 - 2p$\hat{}$3 -> p < 0.5

我们选游戏1。当p > 0.5时，我们选游戏2。当p = 0.5时，选哪个都可以，赢的概率相等。

10.2 There are three ants on different vertices of a triangle. What is the probability of collision (between any two or all of them) if they start walking on the sides of the triangle?Similarly find the probability of collision with 'n' ants on an 'n' vertex polygon.

3只蚂蚁在三角形的3个顶点上，现在让它们沿着三角形的边开始运动， 发生碰撞的概率是多少？如果是n只蚂蚁在n边形的n个顶点上呢，碰撞的概率又是多少?

Solution:

首先，题目中没有提到蚂蚁运动的速度，所以认为它们的速度一样， 不会发生一只蚂蚁追上另一只蚂蚁的情况(面试时可以向面试官确认一下)。 对于任何1只蚂蚁，它有2条边可以选择来走。不会发生冲突的情况是， 所有的蚂蚁都顺时针走或是逆时针走，剩下的情况都是会冲突的。所以，冲突的概率为：

p = 1 - 2 * (1/2)$\hat{}$3 = 3/4

对于n只蚂蚁在n边形上，道理是一样的：

p = 1 - 2 * (1/2)$\hat{}$n = 1 - 1/2$\hat{}$(n-1)

10.3 Given two lines on a Cartesian plane, determine whether the two lines would intersect.

给定笛卡尔坐标系中的两条直线，判断两条线是否相交。

Solution:

这道题目给出的条件非常有限。首先，我们要考虑一些边界条件，比如直线重合， 重合的话是否算相交(假设我们把重合算作是相交)。还有当直线垂直于x轴时， 此时直线的斜率不存在，我们如果用斜率和直线在y轴上的截距来表示直线的话， 这种情况怎么表示？接着是其它一般的情况，斜率不相等时，两直线相交， 但我们不能假设斜率是整数，如果是浮点数，表示两个数不相等是让它们作差， 然后它们的差大于一个足够小的数epsilon(一般为0.000001)。

10.4 Write a method to implement *, - , / operations. You should use only the + operator.

写一个函数实现*, - , /操作，你能使用的操作只有加法+。

Solution:

首先对于这道题目，我们要和面试官确认一下，是不是只针对整数来讨论。 你自己在心里看到这道题目，也要大概估计到应该只在整数范围内考察。否则， 要你只用加法实现个浮点数相乘或是相除，恐怕就不太好办了。

对于乘法，比如a*b(a>0, b>0)，可以视为a个b相加，或是b个a相加。 如果a，b中有负数呢? 我们可以先将它们取绝对值相乘，然后再根据同号相乘为正， 异号相乘为负给结果加上符号。其中，我们还可以做一点小优化， 即如果a<b，就求a个b相加；如果a>b，就求b个a相加。这样可以加法运算次数。

对于减法，a-b，我们只需要转变为a+(-b)即可。

对于除法，a/b，首先我们要考虑的是b不能为0.当b等于0时，抛出异常或返回无穷大(程序 中定义的一个值)。与乘法相同，我们都先对a和b取绝对值，然后不断地从a中减去b， 相应的商数加1。直到a已经不够给b减了，再根据a和b的符号决定是否给商数加上负号即可。

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

const int INF = ~(1 << 31);

void swap(int &a, int &b) {
    a = a^b;
    b = a^b;
    a = a^b;
}

int flipsign(int a) {   // good !
    int d = a < 0 ? 1 : -1;
    int opa = 0;
    while (a != 0) {
        a += d;
        opa += d;
    }
    return opa;
}

int abs(int a) {
    if (a < 0) a = flipsign(a);
    return a;
}

bool opsign(int a, int b) {
    return (a > 0 && b < 0) || (a < 0 && b > 0);
}

int times(int a, int b) {
    if (a == 0 || b == 0) return 0;

    int aa = abs(a);
    int bb = abs(b);
    int res = 0;

    if (aa < bb) swap(aa, bb);
    for (int i = 0; i < bb; ++i)
        res += aa;

    if (opsign(a, b)) return flipsign(res);
    else return  res;
}

int minuss(int a, int b) {
    return a + flipsign(b);
}

```

```
50 int divide(int a, int b) {
51     if (b == 0) return INF;
52     int aa = abs(a), bb = abs(b);
53     int res = 0;
54     for (; (aa += flipsign(bb)) >= 0; ++res);
55     /*
56     while (aa >= bb) {
57             ++res;
58             aa += flipsign(bb);
59     }
60     */
61 
62     if (opsign(a, b)) res = flipsign(res);
63     return res;
64 }
65 
66 int main() {
67     int a[] = {8, 0, -8, -5, 9};
68     int b[] = {3, 5, 3, 0, -3};
69     for (int i = 0; i < 5; ++i) {
70         cout << times(a[i], b[i]) << "␣" << minuss(a[i], b[i]) << "␣" ;
71         cout << divide(a[i], b[i]) << endl;
72     }
73     return 0;
74 }
```

10.5 Given two squares on a two dimensional plane, find a line that would cut these two squares in half.
给出二维平面上的两个正方形，找到一条直线能同时将两个正方形都分为面积相等的两半。

Solution:
一条直线只要过正方形的中心，就一定会将它分为面积相等的两半。(矩形也一样) 那么，我们只要作一条过这两个正方形中心点的直线，即可同时把这两个正方形都分为面积相等的两半。

10.6 Given a two dimensional graph with points on it, find a line which passes the most number of points.
在一个二维图上有许多点，找出一条经过最多点的直线。

Solution:
貌似这是一道Goolge面试题目，好想不好做。我们可以每两个点计算出一条直线，然后将它们放入以直线为键，数量为值的hash map中，以此来求经过点数最多的直线。这样一说，好像OK了，但是这样吗？首先，键值相等就是一个问题。假如我们用斜截式表达直线，由于斜率和截距都是浮点数，那么相等的概念在这里就不能理解为精确的相等，而是两数相差小于一个非常小的数。可是hash map默认可不管，它就认为它们两个是不同的键。导致的结果是什么呢？就是平面上任何两个点之间形成的直线都不是同一条直线。
CTCI书上给出了一种解决方案，就是重写哈希编码方式hashCode和Object类的 equals方法。

```
1  @Override
2  public int hashCode()  {
3      int sl = (int)(slope * 1000);
4      int in = (int)(intercept * 1000);
5      return sl | in;
6  }
7 
8  @Override
9  public boolean equals(Object o) {
10     Line l = (Line) o;
11     if (isEqual(l.slope, slope) && isEqual(l.intercept, intercept)
12         && (infinite_slope == l.infinite_slope)) {
13         return true;
14     }
15     return false;
16 }
```

c++的STL标准里没有hash map，于是我用map来模拟，并写一个hashcode 函数将一条直线转换为一个整型码。这样当两直线的斜率和截距相差都小于epsilon时，都将被映射到一个整数，从而认为它们是一条直线。

```
#include <iostream>
#include <map>
#include <cmath>
#include <cstdlib>
#include <ctime>
using namespace std;

struct point {
    double x;
    double y;
};

class line {
private:
    double epsilon, slope, intercept;
    bool bslope;
public:
    line () {}
    line (point p, point q) {
        epsilon = 0.0001;
        if ( abs(p.x - q.x) > epsilon ) {
            slope = (p.y - q.y) / (p.x - q.x);
            intercept = p.y - slope * p.x;
            bslope = true;
        } else {
            bslope = false;
            intercept = p.x;
        }

    }
    int hashcode() {
        int sl = (int)(slope * 1000);
        int in = (int)(intercept * 1000);
        return sl * 1000 + in;
    }
    void print() {
        cout << "y = " << slope << "x + " << intercept << endl;
    }
};

line find_best_line(point *p, int point_num) {
    line bestline;
    bool first = true;
    map<int, int> mii;
    for (int i = 0; i < point_num; ++i) {
        for (int j = 1; j < point_num; ++j) {
            line l(p[i], p[j]);
            if ( mii.find(l.hashcode()) == mii.end() ) {
                mii[l.hashcode()] = 0;
            }
            mii[l.hashcode()] = mii[l.hashcode()] + 1;    // Not understanding
            if (first) {
                bestline = l;
                first = false;
            } else {
                if ( mii[l.hashcode()] > mii[bestline.hashcode()] )
                    bestline = l;
            }
        }
```

```
60      }
61      return bestline;
62  }
63
64  int main() {
65      srand( (unsigned)time(0) );
66      int graph_size = 100;
67      int point_num = 500;
68      point *p = new point[point_num];
69      for (int i = 0; i < point_num; ++i) {
70          p[i].x = rand()/double(RAND_MAX) * graph_size;
71          p[i].y = rand()/double(RAND_MAX) * graph_size;
72      }
73      line l = find_best_line(p, point_num);
74      l.print();
75      return 0;
76  }
```

10.7 Design an algorithm to find the kth number such that the only prime factors are 3, 5, and 7.
设计算法，找到质因数只有3，5或7的第k个数。

Solution:
首先，我们可以将满足条件的前几个数列出来，以此寻找解题思路。

1	-	3_0 * 5_0 * 7_0
3	3	3_1 * 5_0 * 7_0
5	5	3_0 * 5_1 * 7_0
7	7	3_0 * 5_0 * 7_1
9	3*3	3_2 * 5_0 * 7_0
15	3*5	3_1 * 5_1 * 7_0
21	3*7	3_1 * 5_0 * 7_1
25	5*5	3_0 * 5_2 * 7_0
27	3*9	3_3 * 5_0 * 7_0
35	5*7	3_0 * 5_1 * 7_1
45	5*9	3_2 * 5_1 * 7_0
49	7*7	3_0 * 5_0 * 7_2
63	3*21	3_2 * 5_0 * 7_1

一种简单的思路就是对于已经列出的数，我们依次去乘以3，5，7得到一组数 然后找出最小且还没有列出的数，加入到这个列表。然后重复上面的步骤： 乘以3，5，7，找出最小且还没有列出的数.....这个方法的时间复杂度是O(n2)。
这种思路存在一个问题，就是重复计算。比如对于上面那个表，我想计算下一个数， 那么我用3，5，7去乘以表中的每一个数，然后找出最小且没有用过的数。 可是像3*3,3*5,3*7,5*5,5*7等等都是已经计算过且已经用了的， 按照上面的算法就会不断地重复计算。那我们有没什么办法可以避免重复计算呢？ 那就是将已经计算出来的数保存好，并且保持它们有序。为了避免出现先用3乘以5， 然后又用5去乘以3的这种情况出现(这样会使我们维护的数中出现重复)， 我们可以用3个队列来维护这些数。第1个队列负责乘以3，第2个队列负责乘以5， 第3个队列负责乘以7。算法描述如下：

1. 初始化结果res=1和队列q3,q5,q7
2. 分别往q3,q5,q7插入1*3,1*5,1*7
3. 求出三个队列的队头元素中最小的那个x，更新结果res=x
4. 如果x在:
   q3中，那么从q3中移除x，并向q3，q5，q7插入3*x,5*x,7*x q5中，那么从q5中移除x，并向q5，q7插入5*x,7*x q7中，那么从q7中移除x，并向q7插入7*x
5. 重复步骤3－5，直到找到第k个满足条件的数

注意，当x出现在q5中，我们没往q3中插入3*x，那是因为这个数在q5中已经插入过了。
代码如下：

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

int mini(int a, int b) {
    return a < b ? a : b;     // like
}

int mini(int a, int b, int c)  {
    //return mini(a, b) < c : mini(a, b) : c;
    return mini( mini(a, b), c );
}

int get_num(int val) {
    if (val <= 0) return 0;
    int res = 1, cnt = 1;
    queue<int> q3, q5, q7;
    q3.push(3); q5.push(5); q7.push(7);

    for (; cnt < val; ++cnt) {
        int v3 = q3.front();
        int v5 = q5.front();
        int v7 = q7.front();
        res = mini(v3, v5, v7);
        if (res == v7)
            q7.pop();
        else {
            if (res == v5)
                q5.pop();
            else {
                if (res == v3) {
                    q3.pop();
                    q3.push(3 * res);
                }
            }
            q5.push(5 * res);
        }
        q7.push(7 * res);
    }
    return res;
}

int main() {
    for (int i = 1; i < 20; ++i)
        cout << get_num(i) << endl;
    return 0;
}
```

Chapter 11

Testing

11.1 Find the mistake(s) in the following code:

```
unsigned int i;
for (i = 100; i <= 0; --i)
printf“(%d\”n, i);
```

Solution:

简单题。不过不能粗心，否则可能会以为把i <= 0改为i >= 0就OK了。 这就把一个错误改成了另一个错误。因为i的数据类型是unsigned int， 是恒大于等于0的，这一改就把本来一次都不执行的循环，变成了执行无限次的循环了。 做这种题目，最好看一句，就思考一下这一句可能会带来的限制或是这一句可能的考点XD. 如下:

```
unsigned int i; //unsigned 类型，无符号整型，恒大于等于int0
for (i = 100; i <= 0; --i) //错误，为假，循环一次也不执行。100<=0
                           //改为i错误，因为>=0,恒i，无限循环>=0
                           //改为i，正确，输出到的数。>01001
    printf“(%d\”n, i);
```

上题还可以这样改:

```
int i;
for (i = 100; i >= 0; --i)
    printf“(%d\”n, i);输出到的数。
1000
```

11.2 You are given the source to an application which crashes when it is run. After running it ten times in a debugger, you find it never crashes in the same place. The application is single threaded, and uses only the C standard library. What programming errors could be causing this crash? How would you test each one?

有一个应用程序，运行起来后会崩溃，给你源代码，你在调试器中运行了10次， 发现10次崩溃的地方都不一样。程序是单线程的，只使用了C标准库。 试分析导致程序崩溃的原因可能是什么？你怎么去测试这些原因?

这个问题很大程度上取决于应用程序的类型，当然了， 我们还是可以给出一些导致随机崩溃的常见的原因。

1. 随机的变量：应用程序使用了一些随机的变量或是数值， 由于每次执行这些数都不一样，导致崩溃的地方也不一样。比如：用户输入， 程序产生的随机数，当前系统时间等。
2. 内存泄漏：由于内存泄漏导致程序最终将内存用尽而崩溃。 这个的随机性是由于程序运行时，系统中的进程数量不一样所导致的。 这也包含了堆的溢出或是栈中数据被破坏。

崩溃的原因还可能是由于程序依赖于其它应用程序或是外部模块所导致的。 比如应用程序依赖于系统的某些属性，而它们又被其它应用程序所修改， 那么就有可能导致程序的随机崩溃。与硬件交互得多的应用程序更可能产生这类错误。

在面试中，我们应该问面试官，这个程序是什么类型的，用途是什么？ 这些信息将帮助你找到问题所在(或是面试官所期望的答案)。比如说， web服务器崩溃的原因更可能是由于内存泄漏， 而底层应用程序崩溃的原因就更可能是由于系统依赖。

11.3 We have the following method used in a chess game: boolean canMoveTo(int x, int y) x and y are the coordinates of the chess board and it returns whether or not the piece can move to that position. Explain how you would test this method.

在象棋游戏中，我们有以下的一个函数：boolean canMoveTo(int x, int y)。 x，y是棋盘的坐标，该函数返回棋子是否能移到到那个位置。 解释一下你将如何测试这个函数?

有两个基本的测试类型是我们要做的：验证输入输出的有效性及功能性测试。

a. 输入输出的有效性

我们需要验证输入输出以保证它们是有效的值，这意味着我们需要：

1. 检查是否在棋盘内： 给函数传入负值。 给函数传入大于宽度的x值 给函数传入大于高度的y值 根据具体的实现，以上情况要么返回false，要么抛出异常。
2. 检查输出是否合法（本题目不需要，因为输出是true或者false，无所谓合不合法。）

b. 功能性测试

理论上来说，我们应该去测试象棋棋盘的所有可能布局，但由于这个数量太大了， 这样做根本不现实。因此，我们只去测试这其中比较具有代表性的一些情况。 象棋中有6种棋子(国际象棋：王，后，车，象，马，兵)，我们按照以下算法做测试：

- 取出6种棋子中的一个a
- 除了a的棋子中，取出一个b
- 在方向集合中取出一个方向d
- 用棋子a创建一个棋盘布局(放在棋盘所有可能的地方)
- 将b放在方向d上的一个位置
- 尝试移动，并检查返回值

11.4 How would you load test a webpage without using any test tools?

在不使用测试工具的情况下，你如何去测试一个网页的负载能力?

Solution:

负载测试的一些主要指标包括：

1. 响应时间
2. 吞吐量
3. 资源利用率
4. 系统承受的最大负载

题目说不能使用测试工具，那么我们可以自己写程序针对以上指标来模拟测试。 比如我们可以写一个多线程的程序来模拟成千上万的用户同时访问这个网页。 对于每个线程，我们再去单独测量网页的响应时间，数据IO等。

将程序模拟测试收集到的数据与可接受的数值进行对比，然后对结果进行分析。

11.5 How would you test a pen?

Solution:

你怎么去测试一支笔? 这个问题非常开放，越是开放的问题，我们就越应该给它加上限制。 于是你应该不断地问面试官问题来确定你到底要测试的是一只怎样的笔。 让我们通过以下的模拟对话来学习回答这类题目的技巧：

- 面试官：你怎么去测试一只笔?
- 面试者：这只笔的使用人群是?
- 面试官：可能是儿童
- 面试者：他们用这只笔来干嘛? 写字? 绘画? 还是做其它的?
- 面试官：绘画
- 面试者：在什么东西上画? 纸，布，还是墙上?
- 面试官：在布上
- 面试者：噢。那它是什么笔尖? 毡尖? 圆珠? 它画出来东西能被洗掉吗?
- 面试官：要能被洗掉的 . . . 许多问题过后. . .
- 面试者：嗯。我知道了，我们有一支笔。它的目标人群是5－10岁的儿童， 它是毡尖笔并且有红绿蓝黑等颜色，它可以画在布上并且可以洗掉，是这样吗?

这样一来，面试者就得到了一个与一开始的问题完全不同的问题了，更加的具体， 因而更容易入手。可以进行的测试有：

1. 这支笔画出来的东西可以在什么水温中被洗掉，用热水，温水，冷水测试
2. 画出的东西过了一段时间后还能被洗掉吗? 刚画上去笔迹还是湿的时候就拿去洗会怎样?
3. 对儿童是否安全(无毒)?

等等. . . . . .

11.6 How would you test an ATM in a distributed banking system?

在一个分布式银行系统中，你如何测试一台ATM机?

Solution:

老样子，还是需要弄清楚问题的限制是什么。以下是你可以问的问题:

1. 这台ATM机是给谁用的? 答案可能是任何人，盲人(残疾人)或是其它
2. 这台ATM机的用途是什么? 取款，转账，查询余额，存款?
3. 我们可以使用什么工具来测试? 我们是否能看到系统源码，还是只能面对实体机器?

注意：一个好的测试人员会确保他了解测试对象的方方面面。

以下是一些测试取款功能的测试用例:

1. 取款金额小于余额
2. 取款金额大于余额
3. 取款金额等于余额
4. 同时在ATM机和网上取款
5. 当与银行的网络连接断开后进行取款
6. 同时从多台ATM机取款

Chapter 12

System Design and Memory Limits

12.1 If you were integrating a feed of end of day stock price information (open, high, low,and closing price) for 5,000 companies, how would you do it? You are responsible for the development, rollout and ongoing monitoring and maintenance of the feed. Describe the different methods you considered and why you would recommend your approach. The feed is delivered once per trading day in a comma-separated format via an FTP site. The feed will be used by 1000 daily users in a web application.

如果你要为5000家公司的股价信息整合摘要，你会怎么做？ 你要负责摘要的开发，部署，监控和维护。描述你能想到的不同方法， 及为什么你会推荐这些方法。摘要以逗号分隔的格式经由FTP进行交付，每个交易日一次。 每日有1000个用户在web应用程序中使用这些摘要信息。

Solution:

假设我们有一些脚本在每日结束时通过FTP获取数据。我们把数据存储在哪？ 我们怎样存储数据有助于我们对数据进行分析?

方案一 将数据保存在文本文件中。这样子的话，管理和更新都非常麻烦，而且难以查询。 保存无组织的文本文件是一种非常低效的方法。

方案二 使用数据库。这将带来以下的好处:

1. 数据的逻辑存储
2. 提供了非常便捷的数据查询方式

例子: return all stocks having open $>$ N AND closing price $<$ M (返回开盘价大于N且收盘价小于M的所有股票)

优势:

1. 使得维护更加简单
2. 标准数据库提供了回滚，数据备份和安全保证等功能。我们不需要重复造轮子。

方案三 如果要求不是那么宽泛，我们只想做简单的分析和数据分发，那么XML是另一个很好的选择。 我们的数据有固定的格式和大小：公司名称，开盘价，最高价，最低价和收盘价。 XML看起来应当如下所示:

```
<root>
<date value“”=2008-10-12>
<company name“=”foo>
<open>126.23</open>
<high>130.27</high>
<low>122.83</low>
<closingPrice>127.30</closingPrice>
</company>
<company name“=”bar>
<open>52.73</open>
<high>60.27</high>
<low>50.29</low>
<closingPrice>54.91</closingPrice>
</company>
</date>
```

```
16          <date value“”=2008-10-11> . . . </date>
17          </root>
```

优势:

1. 便于数据分发。这就是为什么XML是共享及分发数据的一个标准模型。
2. 我们有高效的解析器来提取出想要的数据
3. 我们可以往XML文件中追加新数据
4. 不足之处是数据查询比较麻烦(这点比不上数据库)。

12.2 How would you design the data structures for a very large social network (Facebook,LinkedIn, etc)? Describe how you would design an algorithm to show the connection, or path, between two people (e.g., Me -> Bob -> Susan -> Jason -> You).

你会怎样给一个非常大型的社交网站设计数据结构（比如Facebook，LinkedIn）？ 设计一个算法来找到任意两个人之间的联系，比如：我 -> Bob -> Susan -> Jason -> 你

Solution:

方法:

首先，我们先不去考虑数据规模。先从简单的入手。

我们可以把每个人看作一个结点，如果两个人之间是朋友，则这两个结点间有一条边， 这样一来我们就可以构建出一个图。假设我们将“人”这个类设计如下:

```
1 class Person {
2     Person[] friends;
3     // Other info
4 }
```

如果要找到两个人之间的联系，即两个人之间间隔着哪些人。我们就从其中的一个人开始， 做广度优先搜索(BFS)。（做双向的BFS会更快）

但是。。。数据规模太大了!

如果我们去处理Orkut或是Facebook上的数据，单台机器根本无法完成这个任务。 因此，我们考虑用多台机器来处理这个问题。这样一来，Person这个类就需要修改了。 在Person类中，我们存储朋友的ID，然后按照以下方式找到朋友:

- 对于每个朋友id：int machine_index = lookupMachineForUserID(id);
- 转到标号为machine_index的机器去
- Person friend = lookupFriend(machine_index);

对于这个问题，要考虑的优化和问题非常多，这里只列出一些。

优化：减少机器间的跳转次数

机器间的跳转是非常耗时的，因此我们不随机的跳转，而是进行批处理： 比如一个人，他其中的5个朋友在同一台机器上，那么跳转到那台机器后，一次性处理他们。

聪明地划分人与机器

由于同一国家的人更有可能是朋友，因此我们并不随机地把人分配到不同的机器上， 而是以国家，城市，州等进行划分。这样也可以减少机器间的跳转次数。

问题：广度优先搜索会标记已访问结点，对于这个问题，你怎么处理?

在这个问题中，有可能同时有许多人在搜索两人间的联系， 因此直接在原数据上做修改并不好。那怎么办呢? 我们可以对每一个搜索， 使用一个哈希表来记录一个结点是否已经访问过。这样一来， 不同人的搜索之间就不会互相干扰。

其它问题

1. 在真实的世界中，服务器有可能会出故障。你会怎么做?
2. 你怎么利用缓存?
3. 你是否一直搜索直到把图上的结点都遍历一次。如何决定什么时间停止搜索。
4. 在真实世界中，你的朋友里，有一些人的朋友会更多。 那么通过他是否更有可能让你找到与特定某个人的联系。 你怎么利用这个数据来选择遍历的顺序。

下面是算法的示例代码： （java的，就不写成C++了）

```
1  public class Server {
2      ArrayList<Machine> machines = new ArrayList<Machine>();
3  }
4  public class Machine {
5      public ArrayList<Person> persons = new ArrayList<Person>();
6      public int machineID;
7  }
8
9  public class Person {
10     private ArrayList<Integer> friends;
11     private int ID;
12     private int machineID;
13     private String info;
14     private Server server = new Server();
15
16     public String getInfo() { return info; }
17
18     public void setInfo(String info) {
19         this.info = info;
20     }
21
22     public int[] getFriends() {
23         int[] temp = new int[friends.size()];
24         for (int i = 0; i < temp.length; i++) {
25             temp[i] = friends.get(i);
26         }
27         return temp;
28     }
29
30     public int getID() { return ID; }
31     public int getMachineID() { return machineID; }
32     public void addFriend(int id) { friends.add(id); }
33
34     // Look up a person given their ID and Machine ID
35
36     public Person lookUpFriend(int machineID, int ID) {
37         for (Machine m : server.machines) {
38             if (m.machineID == machineID) {
39                 for (Person p : m.persons) {
40                     if (p.ID == ID){
41                         return p;
42                     }
43                 }
44             }
45         }
46         return null;
47     }
48
49     // Look up a machine given the machine ID
50
51     public Machine lookUpMachine(int machineID) {
52         for (Machine m: server.machines) {
53             if (m.machineID == machineID)
54                 return m;
55         }
56         return null;
57     }
58
59     public Person(int iD, int machineID) {
```

```
60          ID = iD;
61          this.machineID = machineID;
62      }
63  }
```

12.3 Given an input file with four billion integers, provide an algorithm to generate an integer which is not contained in the file. Assume you have 1 GB of memory.

FOLLOW UP

What if you have only 10 MB of memory?

给你一个文件，里面包含40亿个整数，写一个算法找出该文件中不包含的一个整数， 假设你有1GB内存可用。

如果你只有10MB的内存呢?

Solution:

我们先来做个算术题，40亿个整数大概有多大?

40 * 10$\hat{8}$ * 4B = 16GB (大约值，因为不是按照2的幂来做单位换算) 这个明显无法一次性装入内存中。但是，如果我们用计算机中的一位来表示某个数出现与否， 就可以减少内存使用量。比如在一块连续的内存区域，15出现，则将第15位置1。 这个就是Bit Map算法。关于这个算法，网上有篇文章写得挺通俗易懂的，推荐:

http://blog.csdn.net/hguisu/article/details/7880288

如果用Bit Map算法，一个整数用一位表示出现与否，需要的内存大小是:

40 * 10$\hat{8}$ b = 5 * 10$\hat{8}$ B = 0.5GB

而我们有1GB的内存，因此该算法可行。由于Bit Map只能处理非负数， (没有说在第-1位上置1的)，因此对于有符号整数，可以将所有的数加上0x7FFFFFFF， 将数据移动到正半轴，找到一个满足条件的数再减去0x7FFFFFFF即可。 因此本文只考虑无符号整数，对于有负数的情况，按照上面的方法处理即可。

我们遍历一遍文件，将出现的数对应的那一位置1，然后遍历这些位， 找到第一个有0的位即可，这一位对应的数没有出现。代码如下:

```
1  #include <iostream>
2  #include <cstdio>
3  using namespace std;
4
5  int main() {
6      freopen("ch1212.3.in", "r", stdin);
7
8      int int_len = sizeof(int) * 8;
9      int bit_len = 0xFFFFFFFF / int_len;
10     int * bit = new int[bit_len];
11     int v;
12
13     while( scanf("%d", &v) != EOF ) {
14         bit[v/int_len] |= 1 << (v % int_len);
15     }
16     bool found = false;
17     for (int i = 0; i < bit_len; ++i) {
18         for (int j = 0; j < int_len; ++j) {
19             if ( (bit[i] & (1<<j)) == 0 ) {
20                 cout << i*int_len + j << endl;
21                 found = true;
22                 break;
23             }
24         }
25         if (found) break;
26     }
27     delete[] bit;
28     fclose(stdin);
29     return 0;
30 }
```

第二问，如果我们只有10MB的内存，明显使用Bit Map也无济于事了。 内存这么小，注定只能分块处理。我们可以将这么多的数据分成许多块， 比如每一个块的大小是1000，那么第一块保存的就是0到999的数，第2块保存的就

是1000 到1999的数.....实际上我们并不保存这些数，而是给每一个块设置一个计数器。 这样每读入一个数，我们就在它所在的块对应的计数器加1。处理结束之后， 我们找到一个块，它的计数器值小于块大小(1000)， 说明了这一段里面一定有数字是文件中所不包含的。然后我们单独处理这个块即可。

接下来我们就可以用Bit Map算法了。我们再遍历一遍数据， 把落在这个块的数对应的位置1(我们要先把这个数归约到0到blocksize之间)。 最后我们找到这个块中第一个为0的位，其对应的数就是一个没有出现在该文件中的数。 代码如下:

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

int main(){
    freopen("12.3.in", "r", stdin);// 20000 number
    int int_len = sizeof(int) * 8;
    int totalnum = 20000;
    int blocksize = 2000;
    int blocknum = totalnum / blocksize;
    int* block = new int[blocknum];
    int* bit = new int[blocksize/int_len+1];
    int v;
    while(scanf("%d", &v) != EOF){
        ++block[v/blocksize];
    }
    fclose(stdin);
    int start;
    for(int i=0; i<blocknum; ++i){
        if(block[i] < blocksize){
            start = i * blocksize;
            break;
        }
    }
    freopen("12.3.in", "r", stdin);
    while(scanf("%d", &v) != EOF){
        if(v>=start && v<start+blocksize){
            v -= start; // make v in [0, blocksize)
            bit[v/int_len] |= 1<<(v%int_len);
        }
    }

    bool found = false;
    for(int i=0; i<blocksize/int_len+1; ++i){
        for(int j=0; j<int_len; ++j){
            if((bit[i] & (1<<j)) == 0){
                cout<<i*int_len+j+start<<endl;
                found = true;
                break;
            }
        }
        if(found) break;
    }

    delete[] block;
    delete[] bit;
    fclose(stdin);
    return 0;
}
```

12.4 You have an array with all the numbers from 1 to N, where N is at most 32,000. The array may have duplicate entries and you do not know what N is. With only 4KB of memory available, how would you print all duplicate elements in the array?

有一个数组，里面的数在1到N之间，N最大为32000.数组中可能有重复的元素(即有的元素 存在2份)，你并不知道N是多少。给你4KB的内存，你怎么把数组中重复的元素打印出来。

我们有4KB的内存，一共有4 * 210 * 8位，大于32000，所以我们可以用Bit Map 来做这道题目。题目很简单，不过我们可以把代码写得漂亮一些。 我们可以写一个Bit Map类来完成基本的位操作。为了代码的简洁， 我们假设程序是运行在32位机器上，即int是32位的。如果考虑代码的通用性， 也可以将32替换成sizeof(int)*8。 代码如下:

Solution:

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

class Bitmap{
private:
    int *bits;
public:
    Bitmap(int size = 32) {
        bits = new int[size/32+1];
    }
    ~Bitmap() {
        delete[] bits;
    }
    bool get(int pos) { // true if bit is 1, else: false
     // return (bits[pos/32] & (1<<pos)) != 0;  // result equavilent
        return (bits[pos/32] & (1<<(pos & 0x1f))) != 0;

    }
    void set(int pos) {
        bits[pos/32] |= ( 1 << (pos & 0x1f) );
    }
};

void print_duplicates(int a[], int n, int bitsize) {
    Bitmap bm(bitsize);
    for (int i = 0; i < n; ++i) {
        if ( bm.get(a[i]-1) ) // bitmap starts at 0, number starts at 1
            cout << a[i] << endl;
        else
            bm.set( a[i]-1 );
    }
}

int main() {
    int a[] = {1, 2, 3, 4, 5, 32000, 7, 8, 9, 10, 11, 1, 2, 13, 15, 16, 32000, 11, 5, 8};
    print_duplicates(a, 20, 32000);
    return 0;
}
```

12.5 If you were designing a web crawler, how would you avoid getting into infinite loops?

如果让你设计一个网络爬虫，你怎么避免陷入无限循环?

Solution:

看完这题，建议用python写个爬虫，对此就能理解的多一些，而且还可以做出好玩的东西。

话说爬虫为什么会陷入循环呢? 答案很简单，当我们重新去解析一个已经解析过的网页时， 就会陷入无限循环。这意味着我们会重新访问那个网页的所有链接， 然后不久后又会访问到这个网页。最简单的例子就是，网页A包含了网页B的链接， 而网页B又包含了网页A的链接，那它们之间就会形成一个闭环。

那么我们怎样防止访问已经访问过的页面呢，答案也很简单，设置一个标志即可。 整个互联网就是一个图结构，我们通常使用DFS(深度优先搜索)和BFS(广度优先搜索) 进行遍历。所以，像遍历一个简单的图一样，将访问过的结点标记一下即可。

12.6 You have a billion urls, where each is a huge page. How do you detect the duplicate documents?

你有10亿个url，每个url对应一个非常大的网页。你怎么检测重复的网页?

Solution:

1. 网页大，数量多，要把它们载入内存是不现实的。 因此我们需要一个更简短的方式来表示这些网页。而hash表正是干这事的。 我们将网页内容做哈希，而不是url，这里不同url可能对应相同的网页内容。
2. 将每个网页转换为一个哈希值后，我们就得到了10亿个哈希值， 很明显，两两对比也是非常耗时的O(n2)。因此我们需要使用其它高效的方法。

根据以上分析，我们可以推出满足条件的以下算法:

1. 遍历网页，并计算每个网页的哈希值。
2. 检查哈希值是否已经在哈希表中，如果是，说明其网页内容是重复的，输出其url。 否则保存url，并将哈希值插入哈希表。 通过这种方法我们可以得到一组url，其对应的网页内容都是唯一的。但有一个问题， 一台计算机可以完成以上任务吗?

1. 一个网页我们要花费多少存储空间?
   - 每个网页转换成一个4字节的哈希值
   - 假设一个url平均是30个字符，那我们至少需要30个字节
   - 因此对应一个url，我们一共要用掉34个字节
2. 34字节 * 10亿 = 31.6GB。很明显，单机的内存是无法搞定的。

我们要如何解决这个问题?

1. 我们可以将这些数据分成多个文件放在磁盘中，分次载入内存处理。 这样一来我们要解决的就是文件的载入/载出问题。
2. 我们可以通过哈希的方式将数据保存在不同文件，这样一来，大小就不是问题了， 但存取时间就成了问题。硬盘上的哈希表随机读写要耗费较多的时间， 主要花费在寻道及旋转延迟上。关于这个问题， 可以使用电梯调度算法来消除磁头在磁道间的随机移动，以此减少消耗的时间。
3. 我们可以使用多台机器来处理这些数据。这样一来，我们要面对的就是网络延迟。 假如我们有n台机器。
   - 首先，我们对网页做哈希，得到一个哈希值v
   - v%n 决定这个网页的哈希值会存放在哪台机器
   - v/n 决定这个哈希值存放在该机器上哈希表的位置

12.7 You have to design a database that can store terabytes of data. It should support efficient range queries. How would you do it?

让你来设计一个能存储TB级数据的数据库，而且要能支持高效的区间查询(范围查询)， 你会怎么做?

Solution:

首先要明确，并不是所有的字段都可以做区间查询。比如对于一个员工， 性别就没有所谓的区间查询；而工资是可以做区间查询的， 例如查询工资大于a元而小于b元的员工。

我们将需要做区间查询的字段对应的字段值提取出来作为关键字构建一棵B+树， 同时保存其对应记录的索引。B+树会对关键字排序，这样我们就可以进行高效的插入， 搜索和删除等操作。我们给定一个查询区间，在B+树中找到对应区间开始的结点只需要O(h)的时间，其中h是树高，一般来说都很小。 叶子结点保存着记录的索引，而且是按关键字(字段值)排好序的。 当我们找到了对应区间开始的叶子结点，再依次从其下一个块中找出对应数量的记录， 直到到到查询区间右端(即最大值)为止。这一步的时间复杂度由其区间中元素数量决定。

避免使用将数据保存在内部结点的树(B+树将数据都保存在叶子结点)， 这样会导致遍历树的开销过大(因为树并非常驻内存)。

讲得这么抽象，还是来张图辅助理解一下吧。

假设这棵B+树上对应的数字表示工资，单位千元。员工对应的工资数据， 其实就都保存在叶子结点上，内部结点和根结点保存的只是其子结点数据的最大值。 这里假设每个叶子结点上的工资值所在的那条记录索引并没有画出来。OK， 现在我们要查询工资大于25k小于60k的员工记录。

区间的开始值是25，我们访问根结点，发现25小于59，于是向左子结点走。 然后继续。发现25大于15且小于44，于是向其中间子结点走。 最后在叶子结点查找到第一个大于25的值是37。接下来再依次地将结点内部的其他值 (44)，和它下一个叶子结点的值(51,59)对应的记录返回(不再往下查找， 因为下面的数已经大于60)。这样一来，即实现了高效的区间查询。

B+树常用于数据库和操作系统的文件系统中，你值得了解XD。

Chapter 13

C++

13.1 Write a method to print the last K lines of an input file using C++.

用C++写一个函数，打印输入文件的最后k行。

Solution:

一种方法是打开文件两次，第一次计算文件的行数N，第二次打开文件，跳过N-K行， 然后开始输出。如果文件很大，这种方法的时间开销会非常大。

我们希望可以只打开文件一次，就可以输出文件中的最后k行。 我们可以开一个大小为k的字符串数组，然后将文件中的每一行循环读入。 怎么样循环读入呢? 就是将k行字符串保存到字符串数组后， 在读第k+1时，将它保存到数组的第1个元素，依次类推。这样一来， 实际上文件中第i行的字符串会被保存到数组的第i%k的位置。最后当文件读完时， 数组保存的就是最后k行的字符串。当然了，它的开始位置不是0，而是i%k。

代码如下:

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

void printLastKLines(ifstream &fin, int k) {
    string line[k];
    int lines = 0;
    string tmp;

 // while (fin.get_line() != EOF) {    // my original, mistake
 // while ( !fin.eof() ) {     // WRONG
    while ( getline(fin, tmp) ) {
        line[lines % k] = tmp;
        ++lines;
    }

    /* // my original
    if (lines % k == 0)
        for (int i = 0; i < k; ++k)
            cout << line[i] << endl;
    else {
        for (int i = lines%k; i < k; ++i)
            cout << line[i] << endl;
        for (int i = 0; i < lines % k; ++i)
            cout << line[i] << endl;
    } */

    int start, cnt;
    if (lines < k) {
        start = 0;
        cnt = k;
```

```
32      } else {
33          start = lines % k;
34          cnt = k;
35      }
36      for (int i = 0; i < cnt; ++i)
37          cout << line[ (start+i) % k ] << endl;
38  }
39
40  int main() {
41      ifstream fin("ch1313.1.in");
42      int k = 4;
43      printLastKLines(fin, k);
44      fin.close();
45      return 0;
46  }
```

注意上面函数中的while循环，不能写成下面的样子：

```
1  while(!fin.eof()){
2      getline(fin, line[lines%k]);
3      ++lines;
4  }
```

eof()函数是在fin中已经没有内容可输入时，才被置为true。 如果使用上面的循环，getline在读入最后一行后，由于这一行仍然是有内容的， 所以eof()返回的仍然为false，表示还没到文件结尾。然后会再进入循环一次， 这时getline读入一个空串，已经没有内容可输入，eof()返回true而退出循环。 结果就是因为多读入一个空串，line数组中保存的是最后k-1行再加一个空串， 两个字：错误。如果我们将循环改成printLastKLines中的样子，那就没问题了。

最后说一句：珍爱生命，远离EOF

13.2 Compare and contrast a hash table vs. an STL map. How is a hash table implemented?If the number of inputs is small, what data structure options can be used instead of a hash table?

对比哈希表和STL map。哈希表是怎么实现的？如果输入数据规模不大， 我们可以使用什么数据结构来代替哈希表。

Solution:

a. 对比哈希表和STL map

在哈希表中，实值的存储位置由其键值对应的哈希函数值决定。因此， 存储在哈希表中的值是无序的。在哈希表中插入元素和查找元素的时间复杂度都是O(1)。 (假设冲突很少)。实现一个哈希表，冲突处理是必须要考虑的。

对于STL中的map，键/值对在其中是根据键进行排序的。它使用一根红黑树来保存数据， 因此插入和查找元素的时间复杂度都是O(logn)。而且不需要处理冲突问题。 STL中的map适合以下情况使用：

1. 查找最小元素
2. 查找最大元素
3. 有序地输出元素
4. 查找某个元素，或是当元素找不到时，查找比它大的最小元素

b. 哈希表是怎么实现的

1. 首先需要一个好的哈希函数来确保哈希值是均匀分布的。比如：对大质数取模
2. 其次需要一个好的冲突解决方法：链表法(chaining，表中元素比较密集时用此法)， 探测法(probing，开放地址法，表中元素比较稀疏时用此法)。
3. 动态地增加或减少哈希表的大小。比如，(表中元素数量)/(表大小)大于一个阈值时， 就增加哈希表的大小。我们新建一个大的哈希表，然后将旧表中的元素值， 通过新的哈希函数映射到新表。

c. 如果输入数据规模不大，我们可以使用什么数据结构来代替哈希表。

你可以使用STL map来代替哈希表，尽管插入和查找元素的时间复杂度是O(logn)， 但由于输入数据的规模不大，因此这点时间差别可以忽略不计。

13.3 How do virtual functions work in C++?

C++中的虚函数是如何工作的?

Solution:

虚函数依赖虚函数表进行工作。如果一个类中，有函数被关键词virtual进行修饰， 那么一个虚函数表就会被构建起来保存这个类中虚函数的地址。同时， 编译器会为这个类添加一个隐藏指针指向虚函数表。如果在派生类中没有重写虚函数， 那么，派生类中虚表存储的是父类虚函数的地址。每当虚函数被调用时， 虚表会决定具体去调用哪个函数。因此，C++中的动态绑定是通过虚函数表机制进行的。

当我们用基类指针指向派生类时，虚表指针vptr指向派生类的虚函数表。 这个机制可以保证派生类中的虚函数被调用到。

```
class Shape{
public:
    int edge_length;
    virtual int circumference () {
        cout<<"Circumference of Base Class\n";
        return 0;
    }
};

class Triangle: public Shape{
public:
    int circumference () {
        cout<<"Circumference of Triangle Class\n";
        return 3 * edge_length;
    }
};

int main(){
    Shape *x = new Shape();
    x->circumference(); // prints "Circumference of Base "Class
    Shape *y = new Triangle();
    y->circumference(); // prints "Circumference of Triangle "Class
    return 0;
}
```

在上面的例子中，circumference是Shape类中的虚函数，因此在它的派生类中， 它也是虚函数。C++中非虚函数的调用是在编译期静态绑定的， 而虚函数的调用则是在执行时才进行动态绑定。

13.4 What is the difference between deep copy and shallow copy? Explain how you would use each.

深拷贝和浅拷贝的区别是什么？你会如何使用它们?

Solution:

浅拷贝并不复制数据，只复制指向数据的指针，因此是多个指针指向同一份数据。 深拷贝会复制原始数据，每个指针指向一份独立的数据。通过下面的代码， 可以清楚地看出它们的区别:

```
struct Test{
    char *ptr;
};

void shallow_copy(Test &src, Test &dest){
    dest.ptr = src.ptr;
}

void deep_copy(Test &src, Test &dest){
    dest.ptr = (char*)malloc(strlen(src.ptr) + 1);
    memcpy(dest.ptr, src.ptr);
}
```

浅拷贝在构造和删除对象时容易产生问题，因此使用时要十分小心。如无必要， 尽量不用浅拷贝。当我们要传递复杂结构的信息，而又不想产生另一份数据时， 可以使用浅拷贝，比如引用传参。浅拷贝特别需要注意的就是析构时的问题， 当多个指针指向同一份内存时，删除这些指针将导致多次释放同一内存而出错。

实际情况下是很少使用浅拷贝的，而智能指针是浅拷贝概念的增强。 比如智能指针可以维护一个引用计数来表明指向某块内存的指针数量， 只有当引用计数减至0时，才真正释放内存。

大部分时候，我们用的是深拷贝，特别是当拷贝的结构不大的时候。

13.5 What is the significance of the keyword “volatile” in C?

谈谈C语言关键字"volatile"的意义(或重要性)?

Solution:

volatile的意思是"易变的"，因为访问寄存器比访问内存要快得多， 所以编译器一般都会做减少存取内存的优化。volatile 这个关键字会提醒编译器，它声明的变量随时可能发生变化(在外部被修改)， 因此，与该变量相关的代码不要进行编译优化，以免出错。

声明一个volatile变量:

```
volatile int x;
int volatile x;
```

声明一个指针，指向volatile型的内存(即指针指向的内存中的变量随时可能变化):

```
volatile int *x;
int volatile *x
```

声明一个volatile指针，指向非volatile内存:

```
int* volatile x;
```

声明一个volatile指针，指向volatile内存(即指针和指针所指物都随机可能变化):

```
volatile int * volatile x;
int volatile * volatile x;
```

volatile在声明上的使用和const是一样的。volatile在*号左边， 修饰的是指针所指物；在*号右边修饰的是指针。

用volatile修饰的变量相关的代码不会被编译器优化，那么它有什么好处呢? 来看下面的例子:

```
int opt = 1;
void Fn(void){
    start:
        if (opt == 1) goto start;
        else break;
}
```

上述代码看起来就是一个无限循环的节奏，编译器可能会将它优化成下面的样子:

```
void Fn(void){
    start:
        int opt = 1;
        if (true)
            goto start;
}
```

由于程序中并没有对opt进行修改，因此将if中的条件设置为恒真。这样一来， 就陷入了无限循环中。但是，如果我们给opt加上volatile修饰， 表明外部程序有可能对它进行修改。那么，编译器就不会做上述优化， 上述程序在opt被外部程序修改后将跳出循环。此外， 当我们在一个多线程程序中声明了一些全局变量，且任何一个线程都可以修改这些变量时， 关键字volatile也会派上用场。在这种情况下， 我们就要明确地告诉编译器不要对这些全局变量的相关代码做优化。

13.6 What is name hiding in C++?

C++中名字隐藏是什么?

Solution:

让我们通过一个例子来讲解C++中的名字隐藏。在C++中，如果一个类里有一个重载的方法， 你用另一个类去继承它并重写(覆盖)那个方法。你必须重写所有的重载方法， 否则未被重写的方法会因为名字相同而被隐藏，从而使它在派生类中不可见。

请看例子:

```
class FirstClass{
public:
    virtual void MethodA(int);
    virtual void MethodA(int, int);
};

```

```
7  void FirstClass::MethodA(int i){
8      cout<<"ONE"<<endl;
9  }
10
11 void FirstClass::MethodA(int i, int j){
12     cout<<"TWO"<<endl;
13 }
```

上面的类中有两个方法(重载的方法)，如果你想在派生类中重写一个参数的函数， 你可以这么做:

```
1  class SecondClass : public FirstClass{
2  public:
3      void MethodA(int);
4  };
5
6  void SecondClass::MethodA(int i){
7      cout<<"THREE"<<endl;
8  }
9
10 int main (){
11     SecondClass a;
12     a.MethodA(1);
13     a.MethodA(1, 1);
14     return 0;
15 }
```

上面的main函数中，第2个MethodA在编译时会报错，提示没有与之匹配的函数。 这是因为两个参数的MethodA在派生类中是不可见的，这就是名字隐藏。

名字隐藏与虚函数无关。所以不管基类中那两个函数是不是虚函数， 在这里都会发生名字隐藏。解决方法有两个。第一个是将2个参数的MethodA换一个名字， 那么它在派生类中就可见了。但我们既然重载了MethodA，说明它们只是参数不同， 而实际上应该是在做相同或是相似的事的。所以换掉名字并不是个好办法。因此， 我们一般采用第二种方法，在派生类中重写所有的重载函数。

13.7 Why does a destructor in base class need to be declared virtual?

为什么基类中的析构函数要声明为虚析构函数?

Solution:

用对象指针来调用一个函数，有以下两种情况:

1. 如果是虚函数，会调用派生类中的版本。
2. 如果是非虚函数，会调用指针所指类型的实现版本。

析构函数也会遵循以上两种情况，因为析构函数也是函数嘛，不要把它看得太特殊。 当对象出了作用域或是我们删除对象指针，析构函数就会被调用。

当派生类对象出了作用域，派生类的析构函数会先调用，然后再调用它父类的析构函数， 这样能保证分配给对象的内存得到正确释放。

但是，如果我们删除一个指向派生类对象的基类指针，而基类析构函数又是非虚的话， 那么就会先调用基类的析构函数(上面第2种情况)，派生类的析构函数得不到调用。

请看例子:

```
1  class Base{
2  public:
3      Base() { cout<<"Base Constructor"<<endl; }
4      ~Base() { cout<<"Base Destructor"<<endl; }
5  };
6
7  class Derived: public Base{
8  public:
9      Derived() { cout<<"Derived Constructor"<<endl; }
10     ~Derived() { cout<<"Derived Destructor"<<endl; }
11 };
12
```

```
13 int main(){
14     Base *p = new Derived();
15     delete p;
16     return 0;
17 }
```

输出是:

- Base Constructor
- Derived Constructor
- Base Destructor

如果我们把基类的析构函数声明为虚析构函数，这会使得所有派生类的析构函数也为虚。 从而使析构函数得到正确调用。

将基类的析构函数声明为虚的之后，得到的输出是:

- Base Constructor
- Derived Constructor
- Derived Destructor
- Base Destructor

因此，如果我们可能会删除一个指向派生类的基类指针时，应该把析构函数声明为虚函数。 事实上，《Effective C++》中的观点是，只要一个类有可能会被其它类所继承， 就应该声明虚析构函数。

13.8 Write a method that takes a pointer to a Node structure as a parameter and returns a complete copy of the passed-in data structure. The Node structure contains two pointers to other Node structures.

写一个函数，其中一个参数是指向Node结构的指针，返回传入数据结构的一份完全拷贝。 Node结构包含两个指针，指向另外两个Node。

Solution:

以下算法将维护一个从原结构的地址到新结构的地址的映射， 这种映射可以让程序发现之前已经拷贝的结点，从而不用为已有结点再拷贝一份。 由于结点中包含指向Node的指针，我们可以通过递归的方式进行结点复制。以下是代码:

```
1  typedef map<Node*, Node*> NodeMap;
2
3  Node* copy_recursive(Node *cur, NodeMap &nodeMap){
4      if(cur == NULL){
5          return NULL;
6      }
7      NodeMap::iterator i = nodeMap.find(cur);
8      if (i != nodeMap.end()){
9          // 'weve been here before, return the copy
10         return i->second;
11     }
12     Node *node = new Node;
13     nodeMap[cur] = node; // map current node before traversing links
14     node->ptr1 = copy_recursive(cur->ptr1, nodeMap);
15     node->ptr2 = copy_recursive(cur->ptr2, nodeMap);
16     return node;
17 }
18
19 Node* copy_structure(Node* root){
20     NodeMap nodeMap; // we will need an empty map
21     return copy_recursive(root, nodeMap);
22 }
```

13.9 Write a smart pointer (smart_ptr) class.

写一个智能指针类(smart_ptr)。

Solution:

比起一般指针，智能指针会自动地管理内存(释放不需要的内存)，而不需要程序员去操心。 它能避免迷途指针(dangling pointers)，内存泄漏(memory leaks)， 分配失败等情况的发生。智能指针需要为所有实例维护一个引用计数， 这样才能在恰当的时刻(引用计数为0时)将内存释放。

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

template <typename T>

class SmartPointer {
private:
    void clear() {
        delete ref;
        free(ref_count);
        ref = NULL;  // prevent it from potentially becoming ''dangling pointer''
        ref_count = NULL;
    }
protected:
    T *ref;
    unsigned *ref_count;
public:
    SmartPointer(T* ptr) {
        ref = ptr;
        ref_count = (unsigned*)malloc(sizeof(unsigned));
        *ref_count = 1;
    }
    SmartPointer(SmartPointer<T> &sptr) {
        ref = sptr.ref;
        ref_count = sptr.ref_count;
        ++*ref_count;
    }
    SmartPointer<T>& operator = (SmartPointer<T> &sptr) {
        if (this != &sptr) {
            if (--*ref_count == 0) {
                clear();
                cout << "operator= clear" << endl;
            }
            ref = sptr.ref;
            ref_count = sptr.ref_count;
            ++*ref_count;
        }
        return *this;
    }
    ~SmartPointer() {
        if (--*ref_count == 0) {
            clear();
            cout << "destructor clear" << endl;
        }
    }

    T getValue() { return *ref; }
};

int main() {
    int *ip1 = new int();
    *ip1 = 11111;
    int *ip2 = new int();
    *ip2 = 22222;
    SmartPointer<int> sp1(ip1), sp2(ip2);
    SmartPointer<int> spa = sp1;
    sp2 = spa; // get different result when commented out
    return 0;
```

```
60 }
```

上述代码有一点值得注意一下，原书在赋值函数中， 并没有检查原指针的引用计数是否已经减为0，然后去释放原指针所指向的内存。 也就是原书的代码有可能导致内存泄漏。正确的做法应该是在把指针指向新的地址前， 将原来指向的引用计数减1，如果为0，说明这个指针在指向新的地址后， 原来指向的内存将不再有指针指向它。那么我们就要把它释放， 否则内存就会在你眼皮底下泄漏的哦。

上述代码main函数中，sp2 = spa这一句如果注释掉，我们得到的输出是:

- destructor clear
- destructor clear

说明内存的清理都是在main函数退出调用析构函数时。如果我们没有注释掉那行代码， 输出是:

- operator= clear
- destructor clear

说明当sp2指向新的内存后，原来的内存由于没有指针指向它而被释放掉。 另一块内存则是在main函数退出时释放的。

如果像CTCI书上所写，当sp2 = spa这一句没有注释掉时，输出是:

- destructor clear

也就是只释放了一块内存(ip1指向的内存)，另一块由于没有指针指向它， 而又不及时清理，结果泄漏了。

Chapter 14

Java

14.1 In terms of inheritance, what is the effect of keeping a constructor private?

14.2 In Java, does the finally block gets executed if we insert a return statement inside the try block of a try-catch-finally?

14.3 What is the difference between final, finally, and finalize?

14.4 Explain the difference between templates in C++ and generics in Java.

14.5 Explain what object reflection is in Java and why it is useful.

14.6 Suppose you are using a map in your program, how would you count the number of times the program calls the put() and get() functions?

Chapter 15

Databases

15.1 Write a method to find the number of employees in each department.

写一条SQL语句找到每一个部门中员工的数量。

Solution:

这个问题可以先将部门表和员工表做连接，然后再统计每个部门中的员工数量。 这里使用左连接，因为对于0个员工的部门，我们也要包含进来。

```
select Dept_Name, Departments.Dept_ID, count(*) as 'num_employees'
from Departments
left join Employees
on Employees.Dept_ID = Departments.Dept_ID
group by Departments.Dept_ID, Dept_Name
```

15.2 What are the different types of joins? Please explain how they differ and why certain types are better in certain situations.

SQL的连接有哪些不同的类型? 解释它们的不同点及在什么情况下用哪种连接，为什么?

Solution:

连接(JOIN)将数据库中的两个或多个表组合起来。为了使用连接， 每个表至少要包含一个相同的字段(属性)。不同的连接类型决定了哪些记录会出现在结果集。

假设我们有以下两张表:

Regular Beverages: Name, Code

Calorie-Free Beverages: Code, Name

一张表列的是常规饮料，另一张表列的是无热量饮料。每张表有两个字段： 饮料名和产品代码。产品代码字段将用于连接时进行匹配。为了简化， 我们用RB代表常规饮料表，CFB代表无热量饮料表。

1. 内连接(INNER JOIN)：结果只包含两个表中严格匹配的记录。 本例中只返回3条记录，1条COCACOLA的和2条PEPSI的。SQL语句如下:

```
SELECT * from RB INNER JOIN CFB ON RB.Code = CFB.Code
```

2. 外连接(OUTER JOIN)：外连接一定包含内连接的结果，但同时还包含一些别的记录。 外连接可分为以下类型:

2.1 左外连接(LEFT OUTER JOIN)：简称左连接(LEFT JOIN)， 结果包含左边的表中所有记录，对于与右表没有匹配的记录，来自右表的所有列设为NULL， 本例中会返回4条记录，除了内连接的3个结果，BUDWEISER也会被列出来。SQL语句如下:

```
SELECT * from RB LEFT OUTER JOIN CFB ON RB.Code = CFB.Code
```

2.2 右外连接(RIGHT OUTER JOIN)简称右连接(RIGHT JOIN)， 右连接与左连接相反，结果将包含右表中的所有记录。右连接很少使用， 因为它总是可以被替换成左连接，只需要交换两表的位置即可。本例中，右连接会得到5条记录，除了内连接的3条，FRESCA和WATER也会被列出来。SQL语句如下:

```
SELECT * from RB RIGHT OUTER JOIN CFB ON RB.Code = CFB.Code
```

2.3 全连接(FULL OUTER JOIN)全连接是左右连接的并集， 结果集中包含被连接表的所有记录，如果缺少匹配的记录，即以NULL填充。本例子中， 全连接会得到6条记录。SQL语句如下:

```
SELECT * from RB FULL OUTER JOIN CFB ON RB.Code = CFB.Code
```

关于连接，维基百科讲得非常好，看它就OK了： 连接(SQL)
http://zh.wikipedia.org/zh/%E8%BF%9E%E6%8E%A5_(SQL)

15.3 What is denormalization? Explain the pros and cons.
什么是反范式? 它优缺点是什么?

Solution:
反范式是通过增加冗余数据或数据分组来提高数据库读性能的过程。在某些情况下， 反范式有助于掩盖关系型数据库软件的低效。关系型的范式数据库即使做过优化， 也常常会带来沉重的访问负载。
数据库的范式设计会存储不同但相关的信息在不同的逻辑表， 如果这些表的存储在物理上也是分离的，那么从几个表中完成数据库的查询可能就会很慢 (比如JOIN操作)。如果JOIN操作的表很多，那么可能会慢得离谱。 有两个办法可以解决这个问题。首选的方法是使逻辑上的设计遵循范式， 但允许数据库管理系统(DBMS)在磁盘上存储额外的冗余信息来加快查询响应。 在这种情况下，DBMS还要保证冗余副本与原始数据的一致性。 这种方法通常在SQL中以索引视图(微软的SQL Server)或物化视图(Oracle)实现。 视图将信息表示为方便查询的格式，索引确保视图上的查询进行了优化。
更常见的做法是对数据做反范式设计。这种方法同样能提高查询响应速度， 但此时不再是DBMS而是数据库设计者去保证数据的一致性。 数据库设计者们通过在数据库中创建规则来保证数据的一致性，这些规则叫约束。 这样一来，数据库设计的逻辑复杂度就增加了，同时额外约束的复杂度也增加了， 这使该方法变得危险。此外，“约束”在加快读操作(SELECT)的同时，减慢了写操作 (INSERT, UPDATE和DELETE)。这意味着一个反范式设计的数据库， 可能比它的范式版本有着更差的写性能。
反范式数据模型与没有范式化的数据模型不同。 只有在范式化已经达到一定的满意水平并且所需的约束和规则都已经建立起来， 才进行反范式化。例如，所有的关系都属于第三范式， 连接的关系和多值依赖得到了妥善处理。
参考链接: http://en.wikipedia.org/wiki/Denormalization

15.4 Draw an entity-relationship diagram for a database with companies, people, and professionals (people who work for companies).
画出一个数据库的实体关系图(ER图)，其中的实体有公司(companies)，人(people)， 专业人士(professionals)。

Solution:
在公司中工作的人(people)是专业人士(professionals)，因此， professionals和people间是ISA(is a)的关系。 或者，我们可以说professionals是从people派生出来的。
除了people的属性，每个professional还有自己额外的属性，如：级别，工作经验等。
一个professional只能去一家公司上班(一般情况下是这样)， 而一家公司可以雇佣很多的professional。因此，它们之间是多对一的关系。“工作(work for)"关系可以有如下属性：加入公司的时间，工资等。 为什么这两个属性是关系的属性而不是professional的属性呢? 因为只有当我们将professional和companies联系起来，才会有这些属性， 或是说这些属性才有意义。
一个人可以有多个电话号码，因此电话号码是一个多值属性。
ER图如下： http://hawstein.com/posts/15.4.html

15.5 Imagine a simple database storing information for students' grades. Design what this database might look like, and provide a SQL query to return a list of the honor roll students (top 10%), sorted by their grade point average.
有一个简单的数据库，存储学生的成绩信息。尝试设计这个数据库，它会是怎样的? 提供一个SQL查询语句来返回光荣榜学生信息(前10%)，按照他们的GPA排序。

Solution:
在一个简单的数据库中，我们至少需要以下三张表：学生表(Students)，课程表(Courses)， 及课程登记表(CourseEnrollment)。学生表中至少需要有学生姓名和学生ID， 及其它的个人信息。课程表包含课程名和课程ID，还可以包含课程描述和授课老师等。 课程登记表将包含学生和课程对(即哪个学生选了什么课，某课程有哪些学生选)， 还包含课程成绩等。我们假设课程成绩是一个整数。
获取光荣榜学生的SQL语句如下:

```
SELECT StudentName, GPA
FROM (
SELECT   top 10 percent Avg(CourseEnrollment.Grade) AS GPA,
 CourseEnrollment.StudentID
FROM CourseEnrollment
GROUP BY CourseEnrollment.StudentID
ORDER BY Avg(CourseEnrollment.Grade)) Honors
INNER JOIN Students ON Honors.StudentID = Students.StudentID
```

Chapter 16

Low Level

16.1 Explain the following terms: virtual memory, page fault, thrashing.
解释下列术语：虚拟内存，缺页中断，抖动。

Solution:

虚拟内存 是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存 （一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片， 还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。 与没有使用虚拟内存技术的系统相比，使用这种技术的系统使得大型程序的编写变得更容易， 对真正的物理内存（例如RAM）的使用也更有效率。

http://zh.wikipedia.org/wiki/

缺页中断 一个页(Page)是一个固定容量的内存区块，是物理内存和外部存储(如硬盘等) 传输的单位。当一个程序访问一个映射到地址空间却实际并未加载到物理内存的页（page）时， 硬件向软件发出的一次中断（或异常）就是一个缺页中断或叫页错误（page fault）。

http://zh.wikipedia.org/wiki/

抖动 在分页存储管理系统中，内存中只存放了那些经常使用的页面， 而其它页面则存放在外存中，当进程运行需要的内容不在内存时， 便启动磁盘读操作将所需内容调入内存，若内存中没有空闲物理块， 还需要将内存中的某页面置换出去。也就是说，系统需要不断地在内外存之间交换信息。 若在系统运行过程中，刚被淘汰出内存的页面，过后不久又要访问它， 需要再次将其调入。而该页面调入内存后不久又再次被淘汰出内存，然后又要访问它。 如此反复，使得系统把大部分时间用在了页面的调入/换出上， 而几乎不能完成任何有效的工作，这种现象称为抖动。

参考资料: 《操作系统教程》 http://en.wikipedia.org/wiki/Thrash_(computer_science)

16.2 What is a Branch Target buffer? Explain how it can be used in reducing bubble cycles in cases of branch misprediction.

16.3 Describe direct memory access (DMA). Can a user level buffer / pointer be used by kernel or drivers?

16.4 Write a step by step execution of things that happen after a user presses a key on the keyboard. Use as much detail as possible.

16.5 Write a program to find whether a machine is big endian or little endian.
写一个程序判断一台机器是大端序还是小端序。

Solution:
关于什么是大端序和小端序，可参见维基百科： 字节序 http://zh.wikipedia.org/wiki/%E5%AD%97%E8%8A%82%E5%BA
判断程序如下:

```
#define BIG_ENDIAN 0
#define LITTLE_ENDIAN 1
int TestByteOrder(){
    short int word = 0x0001;
    char *byte = (char *) &word;
    return (byte[0] ? LITTLE_ENDIAN : BIG_ENDIAN);
}
```

16.6 Discuss how would you make sure that a process doesn't access an unauthorized part of the stack.

16.7 What are the best practices to prevent reverse engineering of DLLs?

16.8 A device boots with an empty FIFO queue. In the first 400 ns period after startup, and in each subsequent 400 ns period, a maximum of 80 words will be written to the queue. Each write takes 4 ns. A worker thread requires 3 ns to read a word, and 2 ns to process it before reading the next word. What is the shortest depth of the FIFO such that no data is lost?

16.9 Write an aligned malloc & free function that takes number of bytes and aligned byte (which is always power of 2)

EXAMPLE

align_malloc (1000,128) will return a memory address that is a multiple of 128 and that points to memory of size 1000 bytes.

aligned_free() will free memory allocated by align_malloc.

16.10 Write a function called my2DAlloc which allocates a two dimensional array. Minimize the number of calls to malloc and make sure that the memory is accessible by the notation arr[i][j].

写一个名为my2DAlloc的函数，用它开一个二维数组。尽可能地少用malloc函数， 确保可以用arr[i][j]这种形式来访问第i行第j列的元素。

Solution:

这道题目最简单的方法就是先开一个数组来存储指向每一行的指针， 然后再为每一行动态地分配空间。这是非常常见的动态申请二维数组空间的方法:

```
1 int** My2DAlloc(int rows, int cols){
2     int **arr = (int**)malloc(rows*sizeof(int*));
3     for(int i=0; i<rows; ++i)
4         arr[i] = (int*)malloc(cols*sizeof(int));
5     return arr;
6 }
```

上述方法使用了(rows+1)次的malloc，malloc使用过多会影响程序的运行效率， 那么有没有办法减少malloc的使用呢。

虽然我们做的事情是动态申请二维数组空间，但这些申请的空间本质上是一维， 只不过有些空间存储了地址，而有些空间则存储了数据。比如上面的方法， 申请了一个长度为rows的一维数组，里面存放的是指针(int*)，指向每一行的地址。 然后又申请了rows*cols大小的空间，里面存放的是整型数据(int)。既然如此， 我们一次性将这么多的空间申请下来，然后在该存放地址的空间存放地址， 在该存放数据的空间存放数据就OK了。

我们需要存储指向每一行的地址，大小为:

```
1 int header = rows * sizeof(int*);
```

同时需要存储rows*cols的整型数据，大小为:

```
1 int data = rows * cols * sizeof(int);
```

我们一次性将这些空间申请下来:

```
1 int **arr = (int**)malloc(header + data);
```

由于前面rows * sizeof(int*)的大小存放的是指针，因此arr类型是int**。 而跨过rows个单元后，后面存放的是整型数据，因此需要将其类型转为int*:

```
1 int *buf = (int*)(arr + rows);
```

最后，从buf指向的地址开始，每cols个单元组成一行，将行首地址存放到arr 的相应位置即可。

```
1 for(int i=0; i<rows; ++i)
2     arr[i] = buf + i * cols;
```

代码如下:

```
1 int** My2DAlloc1(int rows, int cols){
2     int header = rows * sizeof(int*);
3     int data = rows * cols * sizeof(int);
4     int **arr = (int**)malloc(header + data);
5     int *buf = (int*)(arr + rows);
6     for(int i=0; i<rows; ++i)
```

```
7             arr[i] = buf + i * cols;
8         return arr;
9     }
```

这样一来，我们使用一次的malloc就可以动态地申请二维数组空间， 并且可以用arr[i][j]对数组元素进行访问。

Chapter 17

Networking

17.1 Explain what happens, step by step, after you type a URL into a browser. Use as much detail as possible.

一步一步地解释一下，在你往浏览器中输入一个URL后都发生了什么。要尽可能详细。

Solution:

这道题目没有所谓的完全的正确答案，这个题目可以让你在任意的一个点深入下去， 只要你对这个点是熟悉的。以下是一个大概流程:

1. 浏览器向DNS服务器查找输入URL对应的IP地址。
2. DNS服务器返回网站的IP地址。
3. 浏览器根据IP地址与目标web服务器在80端口上建立TCP连接
4. 浏览器获取请求页面的html代码。
5. 浏览器在显示窗口内渲染HTML。
6. 窗口关闭时，浏览器终止与服务器的连接。

这其中最有趣的是第1步和第2步(域名解析)。我们输入的网址(域名)是IP地址的一个别名， 在一个DNS内，一个域名对应一个IP地址。域名系统(DNS) 的工作就是将域名与它的IP地址对应起来。DNS是分布式的，同时也是具有层级关系的。

一个域名服务器虽然只记录一个小的子网内的主机名和IP地址， 但所有的域名服务器联合起来工作，就能将全网内的域名与它们的IP地址对应起来。 这也就意味着，如果一个域名服务器无法找到某个请求域名所对应的IP地址， 它就会向其它的域名服务器发出请求进行寻找。

关于这个问题，推荐阅读以下文章:

原文： What really happens when you navigate to a URL

http://igoro.com/archive/what-really-happens-when-you-navigate-to-a-url/

译文： 当你在浏览器地址栏输入一个URL后回车，将会发生的事情

http://www.cnblogs.com/panxueji/archive/2013/05/12/3073924.html

17.2 Explain any common routing protocol in detail. For example: BGP, OSPF, RIP.

介绍常用路由协议。例如：BGP，OSPF，RIP

Solution:

这个可以根据具体以后从事职位与其的相关度做不同程度的了解。这里只做一个简单介绍。

BGP : Border Gateway Protocol (边界网关协议)

BGP是互联网上的一个核心路由协议。当BGP路由器第一次出现在互联网上时， 它会与那些和它能直接通信的其它BGP路由器建立连接。建立连接后的第一件事， 是从它相邻的路由器下载整个路由表。做完这件事后， 它与其它路由器就只需要交换少量更新信息即可。

BGP路由器通过发送和更新消息来表示到达给定IP地址的计算机的首选路径的变化。 如果一个BGP路由器发现一条新的更好的路径，它将更新自己的路由表， 并向与它直接连接的BGP路由器传播这个信息。 这些BGP路由器将依次决定是否更新自己的路由表及传播这个信息。

参考资料：http://www.livinginternet.com/i/iw_route_egp_bgp.htm

RIP : Routing Information Protocol (路由信息协议)

RIP为局域网提供一个标准的IGP(内部网关协议)协议，它提供了非常好的网络稳定性， 可以保证当一个网络连接断开时，能快速地由另一个网络连接进行包的发送。

RIP的工作倚仗于一个路由数据库，其中保存了计算机之间的快速路由信息。 每次更新中，每一个路由器会去告诉其它路由器它认为哪条路由是最快的。 更新算法将保证每个路由会用与相邻路由通信得到的最快路由去更新它的数据库。

参考资料：http://www.livinginternet.com/i/iw_route_igp_rip.htm

OSPF : Open Shortest Path First (开放式最短路径优先协议)

OSPF是一个特别高效的IGP路由协议。它比RIP要快，但也更复杂。

OSPF和RIP的主要区别是：RIP只保存到达目标地址的下一跳路由的信息； 而OSPF保存了局域网内所有连接的拓扑结构。OSPF算法描述如下：

1. 开始。当一个路由被打开，它向所有的邻居发送"hello"的数据包， 然后接收它们返回的"hello"包。如果它的邻接路由同意同步， 那么它们将建立连接并同步数据库。
2. 更新。每隔一定的时间间隔，每个路由发送一个更新信息给其它路由。 这个更新信息叫“连接状态”，用于向其它路由描述当前路由的数据库信息。 这样一来，所有的路由都将保存相同的局域网拓扑描述。
3. 最短路径树。每个路由会计算一个数据结构叫“最短路径树” 用于描述到达目标地址的最短路径。这样一来，在每次通信中就知道哪个路由是最近的。

参考资料：http://www.livinginternet.com/i/iw_route_igp_ospf.htm

17.3 Compare and contrast the IPv4 and IPv6 protocols.

比较IPv4和IPv6协议。

Solution:

IPv4和IPv6是因特网协议，应用于网络层。IPv4是现在应用得最广泛的协议， 而IPv6是因特网的下一代协议。

- IPv4是因特网协议的第4个版本，它使用32位寻址技术。IPv6是下一代因特网协议， 用的是128位寻址。
- IPv4最多允许4,294,967,296个独立IP地址，而IPv6可以允许34, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000, 000个IP地址。 (34后面带36个0)
- IPv4中，IP地址分为以下几类：A，B，C，D和E。 A，B，C是3类实际在网络中使用的IP地址。D类地址为组播保留。E类地址是保留地址， 意味着它们不应该在IP网络上使用。(除非是一些研究机构出于实验目的才会使用)
- IPv6地址大致可分为3类：
  1. 单播地址。一个单播地址就是一个单一接口的标识符。 一个发送到单播地址的IPv6报文会被传送到该地址所标识的接口。
  2. 组播地址。组播地址就是一组接口的标识符，这些接口可能属于不同的节点。 一个发送到组播地址的IPv6报文会被传送到多个接口。
  3. 任播地址(任一广播地址)。任播地址是一组接口的标识符， 这些接口可能属于不同节点(这点和组播地址相同)。一个发送到任播地址的IPv6 报文会被传送到这组接口中的其中一个。
- IPv4地址表示：239.255.255.255, 255.255.255.0
- IPv6地址用分号“:”分成8段，用16进制表示。比如： 2001:cdba:0000:0000:0000:0000:3257:9652

由于人口数量的增加(对IP地址的需求增加)，IPv6的需求日益明显，它能带来以下好处：

1. 增加地址空间
2. 更高效的路由
3. 降低管理要求
4. 改善更换ISP的方法
5. 更好的移动支持
6. 多宿主
7. 安全
8. 作用域地址：本地链路，本地站点和全局地址空间

17.4 What is a network / subnet mask? Explain how host A sends a message / packet to host B when: (a) both are on same network and (b) both are on different networks. Explain which layer makes the routing decision and how.

网络/子网掩码是什么？解释主机A是如何发送消息/包给主机B的当：(a) A，B在同一网络上。 (b) A，B在不同网络上。解释哪一层做出路由决定，是怎么做的?

Solution:

掩码是一个用于识别网络/子网地址的位模式。IP地址由两部分组成：网络地址和主机地址。

IP地址被分为不同的类，这些类标识了它的网络地址。

例如：IP地址：152.210.011.002，它属于B类地址。那么：

- 网络掩码：11111111.11111111.00000000.00000000
- IP 地址：10011000.11010101.00001011.00000010

对网络掩码和IP地址做与操作，我们得到以下网络地址和主机地址：

- 网络地址：10011000.11010101.00000000.00000000 (152.213.0.0)
- 主机地址：00001011.00000010

同样地，网络管理员可以用子网掩码将任何网络划分成子网。要做到这一点， 我们可以进一步将主机地址划分成更多子网。

例如，如果要将上述网络划分成18个子网(至少需要5位二进制位$25 > 18$)， 我们可以将主机地址的前5位用于标识子网地址。

- 子网掩码：11111111.11111111.11111000.00000000 (255.255.248.0)
- IP 地址：10011000.11010101.00001011.00000010

对子网掩码和IP地址做与操作，我们得到以下的子网地址：

- 10011000.11010101.00001000.00000000 (152.213.8.0)

主机A是如何发送消息/包给主机B

当两台主机在同一网络上时，通过主机地址的二进制位识别其它主机。

当两台主机不在同一网络时，路由器通过网络掩码来识别网络并路由数据包。 每台主机可以通过网络主机地址来识别。

网络层负责进行路由决策。路由表存储路径信息及其对应的开销， 路由算法使用路由表来决定路由这些数据包的路径。

路由一般分为静态路由和动态路由。划分依据是： 路由表是固定的还是会根据当前网络状况而改变。

17.5 What are the differences between TCP and UDP? Explain how TCP handles reliable delivery (explain ACK mechanism), flow control (explain TCP sender / receiver window) and congestion control.

TCP和UDP之间有什么区别？解释TCP如何处理可靠交付(解释ACK机制)，流量控制 (解释TCP发送器/接收器窗口)及拥塞控制。

Solution:

TCP (传输控制协议)：TCP是一个面向连接的协议。从客户端到服务器可以建立一个连接， 然后数据通过这个连接进行传输。TCP协议具有以下特点：

- 可靠的。当你沿着一个TCP套接字发送消息，只要连接没有完全中断， 你就有把握它会到达目的地。如果数据包在传输过程中丢失了， 服务器将重新请求丢失的数据包。这意味着服务器将接收到完整而没有损坏的数据。
- 有序的。TCP数据包中包括序号和确认，所以未按照顺序收到的包可以被排序。 你不必为数据包的到达顺序而担心。
- 重量级的。当TCP字节流以错误的顺序到达，计算机将发出重新发送请求。 序列中已经发出的部分必须一起放回，这个需要一些工作量。

UDP (用户数据报协议)：UDP是无连接的协议。UDP以数据块的形式在网络中发送消息/数据包。 UDP协议的特点如下：

- 不可靠的。当你发送一个消息，你并不能确保它会到达。它有可能在传输中丢失。
- 无序的。当你发送出两个消息。你无法确保哪一个会先到达。
- 轻量的。不会考虑发送消息的顺序，没有连接跟踪，仅仅是发送出去，然后就不管它了。 这也意味着它非常快速，网卡和操作系统只需要做很少的工作来数据包中的数据。

C. 解释TCP如何处理可靠交付(解释ACK机制)和流量控制(解释TCP发送器/接收器窗口)

对于每个TCP数据包，如果接收者收到了数据包，那么它必须发出确认。 发送方如果没有收到确认，会重新发送数据包。这个机制保证每个数据包都被传送到接收方。 ACK是TCP协议中用于确认收到数据包的数据包。TCP窗口是发送方在得到接收方的确认前， 在特定连接上能够发送的数据量。

例如，一对主机在TCP连接上进行通信，其中TCP窗口大小是64KB。那么， 发送方发送完64KB的数据后就必须等待接收方的确认， 接收方发送确认时可能已经收到了全部数据，也可能只收到了部分数据。 如果接收方发出确认所有的数据都已经接收到，那么发送方就会发送另外64KB的数据。 如果发送方收到的确认中表明接收方只收到了32KB的数据(另外32KB的数据可能还在传输， 或者已经丢失)，那么发送方最多只能再发送32KB数据，因为同一时刻不能有超过64KB 的未确认数据。(第二个32KB和此次发送的32KB都未确认)

D. 拥塞控制: TCP使用网络拥塞避免算法进行拥塞控制。有许多不同的算法可以解决这个问题。其中， Tahoe和Reno算法是最有名。为了避免拥塞，TCP使用多方位的拥塞控制策略。 对于每个连接，TCP维护一个拥塞窗口，控制未确认数据包的总大小， 这有点类似于TCP中用于流量控制的滑动窗口。TCP使用一种称为“慢启动” 的机制来增加连接初始化后及超时后的拥塞窗口大小。一开始， 拥塞窗口的大小是最大段大小(MSS)的2倍。尽管初始速率很低，但它增加得很快。 每收到一个确认，拥塞窗口就增加1个MSS。这样，经过一个往返时间(RTT)， 拥塞窗口就增加了一倍。当拥塞窗口的大小超过一个阈值，算法就进入一个新的状态， 叫拥塞避免。

补充阅读： TCP拥塞控制算法 http://www.cnblogs.com/fll/archive/2008/06/10/1217013.html

Chapter 18

Threads and Locks

18.1 What's the difference between a thread and a process?

线程和进程的区别是什么?

Solution:

这是一道出现频率极高的面试题，考察基本概念。

进程可以认为是程序执行时的一个实例。进程是系统进行资源分配的独立实体， 且每个进程拥有独立的地址空间。一个进程无法直接访问另一个进程的变量和数据结构， 如果希望让一个进程访问另一个进程的资源，需要使用进程间通信，比如：管道，文件， 套接字等。

一个进程可以拥有多个线程，每个线程使用其所属进程的栈空间。 线程与进程的一个主要区别是，同一进程内的多个线程会共享部分状态， 多个线程可以读写同一块内存(一个进程无法直接访问另一进程的内存)。同时， 每个线程还拥有自己的寄存器和栈，其它线程可以读写这些栈内存。

线程是进程的一个特定执行路径。当一个线程修改了进程中的资源， 它的兄弟线程可以立即看到这种变化。

以下是分点小结:

1. 进程是系统进行资源分配的基本单位，有独立的内存地址空间； 线程是CPU调度的基本单位，没有单独地址空间，有独立的栈，局部变量，寄存器， 程序计数器等。
2. 创建进程的开销大，包括创建虚拟地址空间等需要大量系统资源； 创建线程开销小，基本上只有一个内核对象和一个堆栈。
3. 一个进程无法直接访问另一个进程的资源；同一进程内的多个线程共享进程的资源。
4. 进程切换开销大，线程切换开销小；进程间通信开销大，线程间通信开销小。
5. 线程属于进程，不能独立执行。每个进程至少要有一个线程，成为主线程

18.2 How can you measure the time spent in a context switch?

你如何测量一次上下文切换所需时间?

Solution:

这是一个棘手的问题，让我们先从可能的解答入手。

上下文切换(有时也称为进程切换或任务切换)是指CPU 的控制权从一个进程或线程切换到另一个。 （参考资料） 例如让一个正在执行的进程进入等待状态(或终止它)，同时去执行另一个正在等待的进程。 上下文切换一般发生在多任务系统中，操作系统必须把等待进程的状态信息载入内存， 同时保存正在运行的进程的状态信息(因为它马上就要变成等待状态了)。

为了解决这个问题，我们需要记录两个进程切换时第一条指令和最后一条指令的时间戳， 上下文切换就是这两个时间戳的差。

来看一个简单的例子，假设只有两个进程：P1和P2。

P1正在执行，而P2在等待。在某个时刻，操作系统从P1切换到P2。假设此时， P1执行到第N条指令，记录时间戳Time_Stamp(P1_N)，当本来在等待的P2 开始执行第1条指令，说明切换完成，记录时间戳Time_Stamp(P2_1)。因此， 上下文切换的时间为：Time_Stamp(P2_1) - Time_Stamp(P1_N)

思路非常简单。问题在于，我们如何知道上下文切换是何时发生的? 进程的切换是由操作系统的调度算法决定的。我们也无法记录进程中每个指令的时间戳。

另一个问题是：许多内核级线程也做上下文切换，而用户对于它们是没有任何控制权限的。

总而言之，我们可以认为，这最多只能是依赖于底层操作系统的近似计算。 一个近似的解法是记录一个进程结束时的时间戳，另一个进程开始的时间戳及排除等待时间。

如果所有进程总共用时为T，那么总的上下文切换时间为： T - (所有进程的等待时间和执行时间)

18.3 Implement a singleton design pattern as a template such that, for any given class Foo, you can call Singleton::instance() and get a pointer to an instance of a singleton of type Foo. Assume the existence of a class Lock which has acquire() and release() methods. How could you make your implementation thread safe and exception safe?

实现一个单例模式的模板，当给一个类Foo时，你可以通过Singleton::instance() 来得到一个指向Foo类单例的指针。假设我们现在已经有了Lock类，其中有acquire() 和release()两个方法，你要如何使你的实现线程安全且异常安全?

Solution:

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

/* 线程同步锁 */
class Lock {
public:
    Lock() {  /* 构造锁 */ }
    ~Lock() { /* 析构锁 */ }
    void AcquireLock() { /* 加锁操作 */ }
    void ReleaseLock() { /* 解锁操作 */ }
};

// 单例模式模板，只实例化一次
template <typename T>
class Singleton{
private:
    static Lock lock;
    static T* object;
protected:
    Singleton() { };
public:
    static T* Instance();
};

template <typename T>
Lock Singleton<T>::lock;

template <typename T>
T* Singleton<T>::object = NULL;

template <typename T>
T* Singleton<T>::Instance(){
    if (object == NULL){// 如果object未初始化，加锁初始化
        lock.AcquireLock();
        //这里再判断一次，因为多个线程可能同时通过第一个if
        //只有第一个线程去实例化object，之后object非NULL
        //后面的线程不再实例化它
        if (object == NULL){
            object = new T;
        }
        lock.ReleaseLock();
    }
    return object;
}
class Foo{

};
int main(){
```

```
49      Foo* singleton_foo = Singleton<Foo>::Instance();
50      return 0;
51  }
```

使一个程序线程安全的一般方法是，当线程获得对共享资源的写权限时，需要对共享资源加锁。这样一来，当一个线程在修改资源时，另外的线程就不能修改它。

关于这个问题，可以参考《剑指offer》中的面试题2：实现Singleton模式。那里有详细的解答。

18.4 Design a class which provides a lock only if there are no possible deadlocks.

18.5 Suppose we have the following code:

```
1  class Foo{
2  public:
3      A(.....); /*If A is called, a new thread will be created and
4                  the corresponding function will be executed. */
5      B(.....); /*same as above */
6      C(.....); /*same as above */
7  };
8  Foo f;
9  f.A(.....);
10 f.B(.....);
11 f.C(.....);
```

i) Can you design a mechanism to make sure that B is executed after A, and C is executed after B?

ii) Suppose we have the following code to use class Foo. We do not know how the threads will be scheduled in the OS.

```
1  Foo f;
2  f.A(.....); f.B(.....); f.C(.....);
3  f.A(.....); f.B(.....); f.C(.....);
```

Can you design a mechanism to make sure that all the methods will be executed in sequence?

假设我们有以下代码：

```
1  class Foo{
2  public:
3      A(.....); /*当A被调用时，会创建一个新的线程并执行相应的函数*/
4      B(.....); /*同上*/
5      C(.....); /*同上*/
6  };
7  Foo f;
8  f.A(.....);
9  f.B(.....);
10 f.C(.....);
```

i) 你能设计一种机制确保B在A后执行，C在B后执行吗?

ii) 假设我们有以下代码，我们并不知道操作系统如何调度线程。你能设计一种机制来确保所有的方法都按顺序执行吗?

```
1  Foo f;
2  f.A(.....); f.B(.....); f.C(.....);
3  f.A(.....); f.B(.....); f.C(.....);
```

Solution:

第一问，初始的两个信号量都为0，函数A执行完后，信号量s_a会加1，这时B才可执行。B执行完后信号量s_b加1，这时C才可执行。以此保证A，B，C的执行顺序。注意到函数A其实没有受到限制，所以A可以被多个线程多次执行。比如A执行3次，此时s_a=3；然后执行B，s_a=2,s_b=1；然后执行C，s_a=2,s_b=0；然后执行B，s_a=1,s_b=1。即可以出现类似这种序列：AAABCB。

```
Semaphore s_a(0);
Semaphore s_b(0);
A {
    s_a.release(1); // 信号量s_a加1
}
B {
    s_a.acquire(1); // 信号量s_a减1
    s_b.release(1); // 信号量s_b加1
}
C {
    s_b.acquire(1); // 信号量s_b减1
}
```

第二问代码如下，与第一问不同，以下代码可以确保执行顺序一定严格按照： ABCABCABC...进行。因为每一时刻都只有一个信号量不为0， 且B中要获取的信号量在A中释放，C中要获取的信号量在B中释放， A中要获取的信号量在C中释放。这个保证了执行顺序一定是ABC。

```
Semaphore s_a(0);
Semaphore s_b(0);
Semaphore s_c(1);
A {
    s_c.acquire(1);
    /***/
    s_a.release(1);
}
B {
    s_a.acquire(1);
    /****/
    s_b.release(1);
}
C {
    s_b.acquire(1);
    /******/
    s_c.release(1);
}
```

18.6 You are given a class with synchronized method A, and a normal method C. If you have two threads in one instance of a program, can they call A at the same time? Can they call A and C at the same time?

Chapter 19

Moderate

19.1 Write a function to swap a number in place without temporary variables.
写一个函数交换两个数，不能使用临时变量。

Solution:
交换函数swap是经常用到的函数，小巧简单，以下两种实现方式都不需要使用临时变量:

```
// 实现1
void swap(int &a, int &b){
    b = a - b;
    a = a - b;
    b = a + b;
}
// 实现2
void swap(int &a, int &b){
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}
```

以上的swap函数，尤其是第2个实现，简洁美观高效，乃居家旅行必备良品。但是， 使用它们之前一定要想一想，你的程序中，是否有可能会让swap中的两个形参引用同一变量。 如果是，那么上述两个swap函数都将出问题。有人说，谁那么无聊去swap同一个变量。 那可不好说，比如你在操作一个数组中的元素，然后用到了以下语句:

```
swap(a[i], a[j]); // i==j时，出问题
```

你并没有注意到swap会去操作同一变量，可是当i等于j时，就相当于你这么干了。 然后呢，上面两个实现执行完第一条语句后，操作的那个内存中的数就变成0了。 后面的语句不会起到什么实际作用。
所以如果程序中有可能让swap函数去操作同一变量，就老老实实用最朴素的版本:

```
void swap(int &a, int &b){
    int t = a;
    a = b;
    b = t;
}
```

19.2 Design an algorithm to figure out if someone has won in a game of tic-tac-toe.
设计算法检查某人是否赢得了井字游戏。

Solution:
假设这个检查某人是否赢得了井字游戏的函数为HasWon，那么我们第一步要向面试官确认， 这个函数是只调用一次，还是要多次频繁调用。如果是多次调用， 我们可以通过预处理来得到一个非常快速的版本。

方法一： 如果HasWon函数需要被频繁调用
对于井字游戏，每个格子可以是空，我的棋子和对方的棋子3种可能，总共有$39 = 19683$ 种可能状态。我们可以把每一种状态转换成一个整数， 预处理时把这个状态下是否有人赢得了井字游戏保存起来，每次检索时就只需要O(1)时间。 比如每个格子上的3种状态为0(空)，1(我方棋子)，2(对方棋子)，棋盘从左到右， 从上到下依次记为0到8，那么任何一个状态都可以写成一个3进制的数，再转成10进制即可。 比如，下面的状态:

- 1 2 2
- 2 1 1
- 2 0 1
- 可以写成:
- 122211201=1*3$\hat{8}$ + 2*3$\hat{7}$ +...+ 0*3$\hat{1}$ + 1*3$\hat{0}$

如果只需要返回是否有人赢，而不需要知道是我方还是对方。 那可以用一个二进制位来表示是否有人赢。比如上面的状态，1赢， 就可以把那个状态转换成的数对应的位置1。如果需要知道是谁赢， 可以用一个char数组来保存预处理结果。

方法二： 如果HasWon函数只被调用一次或很少次

如果HasWon函数只被调用一次或很少次，那我们就没必要去预存结果了， 直接判断一下就好。只为一次调用去做预处理是不值得的。

代码如下，判断n*n的棋盘是否有人赢，即同一棋子连成一线:

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

enum Check {ROW, COLUMN, DIAGONAL, REDIAGONAL};

int CheckRowColumn(int board[], int n, Check check) {
    int type = 0;
    for (int i = 0; i < n; ++i) {
        bool found = true;
        for (int j = 0; j < n; ++j) {
            int k = 0;
            if (check == ROW)
                k = i * n + j;
            else
                k = i + j * n;

            if (j == 0)
                type = board[k];
            else if ( board[k] != type ) {
                found = false;
                break; //有一个不满足，检查下一行
            }
        }
        if (found) return type;
    }
    return 0;
}

int CheckDiagonal(int board[], int n, Check check) {
    int type = 0;
    bool found = true;
    // 主对角
    for (int i = 0; i < n; ++i) {
        int k = 0;
        if (check == DIAGONAL)
            k = i + i * n;
        else
            k = i + (n-1-i) * n;
        if (i == 0)
            type = board[k];
        else if (board[k] != type) {
            found = false;
            break;
        }
    }
    if (found) return type;
    return 0;
```

```
48  }
49
50  int HasWon(int board[], int n) {
51      int type = 0;
52      type = CheckRowColumn(board, n, ROW);
53      if (type != 0) return type;
54      type = CheckRowColumn(board, n, COLUMN);
55      if (type != 0) return type;
56      type = CheckDiagonal(board, n, DIAGONAL);
57      if (type != 0) return type;
58      type = CheckDiagonal(board, n, REDIAGONAL);
59      if (type != 0) return type;
60      return 0;
61  }
62
63  int main() {
64      int n = 3; // 3 * 3
65      int board[] = {2, 2, 1, 2, 1, 1, 1, 2, 0};
66      int type = HasWon(board, n);
67      if (type != 0)
68          cout << type << " won!" << endl;
69      else
70          cout << "Nobody won!" << endl;
71      return 0;
72  }
```

19.3 Write an algorithm which computes the number of trailing zeros in n factorial.

写一个算法计算n的阶乘末尾0的个数。

Solution:

首先，算出n的阶乘的结果再去计算末尾有多少个0这种方法是不可取的， 因为n的阶乘是一个非常大的数，分分种就会溢出。我们应当去分析， 是什么使n的阶乘结果末尾出现0。

n阶乘末尾的0来自因子5和2相乘，5*2=10。因此，我们只需要计算n的阶乘里， 有多少对5和2。注意到2出现的频率比5多，因此，我们只需要计算有多少个因子5即可。 我们可以列举一些例子，看看需要注意些什么：

- 5!, 包含1*5, 1个5
- 10!, 包含1*5,2*5, 2个5
- 15!, 包含1*5,2*5,3*5, 3个5
- 20!, 包含1*5,2*5,3*5,4*5, 4个5
- 25!, 包含1*5,2*5,3*5,4*5,5*5, 6个5
- ...

给定一个n，用n除以5，得到的是从1到n中包含1个5的数的个数；然后用n除以5去更新n， 相当于把每一个包含5的数中的因子5取出来一个。然后继续同样的操作，让n除以5， 将得到此时仍包含有5的数的个数，依次类推。最后把计算出来的个数相加即可。 比如计算25的阶乘中末尾有几个0， 先用25除以5得到5，表示我们从5,10,15,20,25中各拿一个因子5出来，总共拿了5个。 更新n=25/5=5，再用n除以5得到1，表示我们从25中拿出另一个因子5， 其它的在第一次除以5后就不再包含因子5了。

代码如下：

```
1  int NumZeros(int n){
2      if(n < 0) return -1;
3      int num = 0;
4      while((n /= 5) > 0){
5          num += n;
6      }
7      return num;
8  }
```

19.4 Write a method which finds the maximum of two numbers. You should not use if-else or any other comparison operator.

Input: 5, 10

Output: 10

写一个函数返回两个数中的较大者，你不能使用if-else及任何比较操作符。

Solution:

我们可以通过一步步的分析来将需要用到的if-else和比较操作符去掉:

- If a > b, return a; else, return b.
- If (a - b) < 0, return b; else, return a.
- If (a - b) < 0, 令k = 1; else, 令k = 0. return a - k * (a - b).
- 令z = a - b. 令k是z的最高位，return a - k * z.

当a大于b的时候，a-b为正数，最高位为0，返回的a-k*z = a；当a小于b的时候， a-b为负数，最高位为1，返回的a-k*z = b。可以正确返回两数中较大的。

另外，k是z的最高位(0或1)，我们也可以用一个数组c来存a,b，然后返回c[k]即可。

代码如下:

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

int Max1(int a, int b) {
    int c[2] = {a, b};
    int z = a - b;
    z = (z >> 31) & 1;
    return c[z];
}

int Max2(int a, int b) {
    int z = a - b;
    int k = (z >> 31) & 1;
    return a - k*z;
}

int main() {
    int a = 5, b = 10;
    cout << Max1(a, b) << endl;
    cout << Max2(a, b) << endl;
    return 0;
}
```

19.5 The Game of Master Mind is played as follows:

The computer has four slots containing balls that are red (R), yellow (Y), green (G) or blue (B). For example, the computer might have RGGB (e.g., Slot #1 is red, Slots #2 and #3 are green, Slot #4 is blue).

You, the user, are trying to guess the solution. You might, for example, guess YRGB.When you guess the correct color for the correct slot, you get a "hit". If you guess a color that exists but is in the wrong slot, you get a "pseudo-hit". For example, the guess YRGB has 2 hits and one pseudo hit.

For each guess, you are told the number of hits and pseudo-hits. Write a method that, given a guess and a solution, returns the number of hits and pseudo hits.

Master Mind游戏规则如下:

4个槽，里面放4个球，球的颜色有4种，红(R)，黄(Y)，绿(G)，蓝(B)。比如， 给出一个排列RGGB，表示第一个槽放红色球，第二和第三个槽放绿色球，第四个槽放蓝色球。

你要去猜这个排列。比如你可能猜排列是：YRGB。当你猜的颜色是正确的，位置也是正确的， 你就得到一个hit，比如上面第3和第4个槽猜的和真实排列一样(都是GB)，所以得到2个hit。 如果你猜的颜色在真实排列中是存在的，但位置没猜对，你就得到一个pseudo-hit。比如， 上面的R，猜对了颜色，但位置没对，得到一个pseudo-hit。

对于你的每次猜测，你会得到两个数：hits和pseudo-hits。写一个函数， 输入一个真实排列和一个猜测，返回hits和pseudo-hits。

Solution:

这个问题十分直观，但有一个地方需要去向面试官明确一下题意。关于pseudo-hits的定义， 猜对颜色但位置没对，得到一个pseudo-hit，这里是否可以包含重复？举个例子， 比如真实排列是：RYGB，猜测是：YRRR。那么hits很明显为0了。pseudo-hits呢？ 猜测中Y对应真实排列中的Y，得到一个pseudo-hits。猜测中有3个R， 而真实排列中只有一个，那这里应该认为得到1个pseudo-hits还是3个？ CTCI书认为是3个，想必理由是猜测中的3个R都满足：出

现在真实排列中，位置不正确。 所以算3个。但我认为，这里算1个比较合理，真实排列中的R只和猜测中的一个R配对， 剩余的没有配对，不算。弄清题意后，代码就不难写出了。

以下是两种不同理解的实现:

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

struct Result {
    int hits;
    int pseudo_hits;
};

Result Estimate(const char* solution, const char* guess) {
    Result res;
    res.hits = 0;
    res.pseudo_hits = 0;
    int solution_mask = 0;

    for (int i = 0; i < 4; ++i)
        solution_mask |= 1 << (solution[i] - 'A');

    for (int i = 0; i < 4; ++i) {
        if (guess[i] == solution[i])
            ++res.hits;
        else if ( solution_mask & (1 << (guess[i]-'A')) )
            ++res.pseudo_hits;
    }
    return res;
}

int Min(int a, int b) {
    return a < b ? a : b;
}

Result Estimate1(const char* solution, const char* guess) {
    Result res;
    res.hits = 0;
    res.pseudo_hits = 0;
    int num = 26 + 5;
    int guess_count[num], solution_count[num];
    memset(guess_count, 0, sizeof(guess_count));
    memset(solution_count, 0, sizeof(solution_count));

    for (int i = 0; i < 4; ++i) {
        if (guess[i] == solution[i])
            ++res.hits;
        ++guess_count[(int)(guess[i]-'A')];
        ++solution_count[(int)(solution[i]-'A')];
    }
    char color[] = "RGBY";
    for (int i = 0; i < 4; ++i) {
        int idx = (int)(color[i] - 'A');
        res.pseudo_hits += Min(guess_count[idx], solution_count[idx]);
    }
    res.pseudo_hits -= res.hits;
    return res;
}

int main() {
```

```
57      char solution[] = "RYGB";
58      char guess[] = "YRRR";
59      Result res = Estimate(solution, guess);
60      cout << res.hits << " " << res.pseudo_hits << endl;
61      Result res1 = Estimate1(solution, guess);
62      cout << res1.hits << " " << res1.pseudo_hits << endl;
63
64      return 0;
65  }
```

19.6 Given an integer between 0 and 999,999, print an English phrase that describes the integer (eg, "One Thousand, Two Hundred and Thirty Four").

19.7 You are given an array of integers (both positive and negative). Find the continuous sequence with the largest sum. Return the sum.

EXAMPLE

Input: 2, -8, 3, -2, 4, -10

Output: 5 (i.e., 3, -2, 4)

给出一个整数数组(包含正数和负数)，找到和最大的连续子序列，返回和。

输入: 2, -8, 3, -2, 4, -10

输出: 5 (即, 3, -2, 4)

Solution:

经典的面试题目，遍历一遍数组，用变量maxsum保存遍历过程中的最大和， 用变量cursum保存遍历过程中的当前和。在遍历的过程中，我们只需要做3件事， 第一：如果当前和cursum小于等于0，说明前面的连续和不会对后面的连续和产生贡献， 要么使后面的连续和减少，要么不变。因此舍弃cursum，用当前的元素更新它。 第二：如果当前和cursum是大于0的，累加当前元素。第三：如果当前和cursum 大于最大和maxsum，则更新最大和maxsum。

此外，我们可以用一个全局变量来标示非法输入。代码如下：

```
1   #include <iostream>
2   using namespace std;
3
4   bool g_Invalid = false;
5
6   int GetMaxSum(int a[], int n) {
7       if (a == NULL || n <= 0) {
8           g_Invalid = true;
9           return 0;
10      }
11      g_Invalid = false;
12
13      int max_sum = 1 << 31; // Min Int
14      int cur_sum = 0;
15      for (int i = 0; i < n; ++i) {
16          if (cur_sum <= 0)
17              cur_sum = a[i];
18          else if (cur_sum > 0)
19              cur_sum += a[i];
20
21          if (cur_sum > max_sum)
22              max_sum = cur_sum;
23      }
24      return max_sum;
25  }
26
27  int main() {
28      int a[] = {2, -8, 3, -2, 4, -10};
29      int max_sum = GetMaxSum(a, 6);
30      if (g_Invalid)
31          cout << "Invalid Input!" << endl;
```

```
32      else
33          cout << max_sum << endl;
34      return 0;
35  }
```

19.8 Design a method to find the frequency of occurrences of any given word in a book.
设计一个函数，找到给定单词在一本书中的出现次数。

Solution:

这道题目和19.2是一个思路。如果只需要查询一次， 那就直接做；如果要多次查询，而且要查询各种不同的单词，那就先预处理一遍， 接下来就只需要用O(1)的时间进行查询。

只查询一次 遍历这本书的每个单词，计算给定单词出现的次数。时间复杂度O(n)，我们无法继续优化它， 因为书中每个单词都需要访问一次。当然，如果我们假设书中的单词是均匀分布， 那我们就可以只统计前半本书某个单词出现的次数，然后乘以2； 或是只统计前四分之一本书某个单词出现的次数，然后乘以4。这样能计算出一个大概值。 当然，单词均匀分布这个假设太强了，一般是不成立的。

多次查询 如果我们要对一本书进行多次的查询，就可以遍历一次这本书的单词， 把它出现的次数存入哈希表中。查询的时候即可用O(1)的时间完成。

19.9 Since XML is very verbose, you are given a way of encoding it where each tag gets mapped to a pre-defined integer value. The language/grammar is as follows:

- Element –> Element Attr* END Element END [aka, encode the element tag, then its attributes, then tack on an END character, then encode its children, then another end tag]
- Attr –> Tag Value [assume all values are strings]
- END –> 01
- Tag –> some predefined mapping to int
- Value –> string value END

Write code to print the encoded version of an xml element (passed in as string).

FOLLOW UP

Is there anything else you could do to (in many cases) compress this even further?

19.10 Write a method to generate a random number between 1 and 7, given a method that generates a random number between 1 and 5 (i.e., implement rand7() using rand5()).
给你一个能生成1到5随机数的函数，用它写一个函数生成1到7的随机数。(即，使用函数rand5()来实现函数rand7())。

Solution:

rand5可以随机生成1,2,3,4,5；rand7可以随机生成1,2,3,4,5,6,7。 rand5并不能直接产生6，7，所以直接用rand5去实现函数rand7似乎不太好入手。 如果反过来呢？给你rand7，让你实现rand5，这个好实现吗？

一个非常直观的想法就是不断地调用rand7，直到它产生1到5之间的数，然后返回。 代码如下：

```
1  int Rand5(){
2      int x = ~(1<<31); // max int
3      while(x > 5)
4          x = Rand7();
5      return x;
6  }
```

等等，这个函数可以等概率地产生1到5的数吗？首先，它确确实实只会返回1到5这几个数， 其次，对于这些数，都是由Rand7等概率产生的(1/7)，没有对任何一个数有偏袒， 直觉告诉我们，Rand5就是等概率地产生1到5的。事实呢？让我们来计算一下， 产生1到5中的数的概率是不是1/5就OK了。比如说，让我们来计算一下Rand5生成1 的概率是多少。上面的函数中有个while循环，只要没生成1到5间的数就会一直执行下去。 因此，我们要的1可能是第一次调用Rand7时产生，也可能是第二次，第三次，...第n次。 第1次就生成1，概率是1/7；第2次生成1，说明第1次没生成1到5间的数而生成了6，7， 所以概率是(2/7)*(1/7)，依次类推。生成1的概率计算如下：

- P(x=1)=1/7 + (2/7) * 1/7 + (2/7)$\hat{}$2 * 1/7 + (2/7)$\hat{}$3 * 1/7 + ...
- =1/7 * (1 + 2/7 + (2/7)$\hat{}$2 + ...) // 等比数列
- =1/7 * 1 / (1 - 2/7)
- =1/7 * 7/5
- =1/5

上述计算说明Rand5是等概率地生成1,2,3,4,5的(1/5的概率)。从上面的分析中， 我们可以得到一个一般的结论，如果a > b，那么一定可以用Randa去实现Randb。其中， Randa表示等概率生成1到a的函数，Randb表示等概率生成1到b的函数。代码如下：

```
// a > b
int Randb(){
    int x = ~(1<<31); // max int
    while(x > b)
        x = Randa();
    return x;
}
```

回到正题，现在题目要求我们要用Rand5来实现Rand7，只要我们将Rand5 映射到一个能产生更大随机数的Randa，其中a > 7，就可以套用上面的模板了。 这里要注意一点的是，你映射后的Randa一定是要满足等概率生成1到a的。比如，

- Rand5() + Rand5() - 1

上述代码可以生成1到9的数，但它们是等概率生成的吗？不是。生成1只有一种组合： 两个Rand5()都生成1时：(1, 1)；而生成2有两种：(1, 2)和(2, 1)；生成6更多。 它们的生成是不等概率的。那要怎样找到一个等概率生成数的组合呢?

我们先给出一个组合，再来进行分析。组合如下：

- 5 * (Rand5() - 1) + Rand5()

Rand5产生1到5的数，减1就产生0到4的数，乘以5后可以产生的数是：0,5,10,15,20。 再加上第二个Rand5()产生的1,2,3,4,5。我们可以得到1到25， 而且每个数都只由一种组合得到，即上述代码可以等概率地生成1到25。OK，到这基本上也就解决了。

套用上面的模板，我们可以得到如下代码：

```
int Rand7(){
    int x = ~(1<<31); // max int
    while(x > 7)
        x = 5 * (Rand5() - 1) + Rand5() // Rand25
    return x;
}
```

上面的代码有什么问题呢？可能while循环要进行很多次才能返回。 因为Rand25会产生1到25的数，而只有1到7时才跳出while循环， 生成大部分的数都舍弃掉了。这样的实现明显不好。我们应该让舍弃的数尽量少， 于是我们可以修改while中的判断条件，让x与最接近25且小于25的7的倍数相比。 于是判断条件可改为x > 21，于是x的取值就是1到21。 我们再通过取模运算把它映射到1-7即可。代码如下：

```
int Rand7(){
    int x = ~(1<<31); // max int
    while(x > 21)
        x = 5 * (Rand5() - 1) + Rand5() // Rand25
    return x%7 + 1;
}
```

这个实现就比上面的实现要好，并且可以保证等概率生成1到7的数。

让我们把这个问题泛化一下，从特殊到一般。现在我给你两个生成随机数的函数Randa， Randb。Randa和Randb分别产生1到a的随机数和1到b的随机数，a，b不相等 (相等就没必要做转换了)。现在让你用Randa实现Randb。

通过上文分析，我们可以得到步骤如下：

1. 如果a > b，进入步骤2；否则构造Randa2 = a * (Randa - 1) + Randa， 表示生成1到a2 随机数的函数。如果a2 仍小于b，继教构造 Randa3 = a * (Randa2 - 1) + Randa... 直到ak > b，这时我们得到Randak , 我们记为RandA。
2. 步骤1中我们得到了RandA(可能是Randa或Randak)，其中A > b， 我们用下述代码构造Randb：

```
    // A > b
    int Randb(){
      int x = ~(1<<31); // max int
      while(x > b*(A/b)) // b*(A/b表示最接近)且小于的的倍数AAb
```

```
5           x = RandA();
6           return x%b + 1;
7       }
```

从上面一系列的分析可以发现，如果给你两个生成随机数的函数Randa和Randb， 你可以通过以下方式轻松构造Randab，生成1到a*b的随机数。

- Randab = b * (Randa - 1) + Randb
- Randab = a * (Randb - 1) + Randa

如果再一般化一下，我们还可以把问题变成：给你一个随机生成a到b的函数， 用它去实现一个随机生成c到d的函数。有兴趣的同学可以思考一下，这里不再讨论。

19.11 Design an algorithm to find all pairs of integers within an array which sum to a specified value.

设计一个算法，找到数组中所有和为指定值的整数对。

Solution:

时间复杂度O(n)的解法 我们可以用一个哈希表或数组或bitmap(后两者要求数组中的整数非负)来保存sum-x的值， 这样我们就只需要遍历数组两次即可找到和为指定值的整数对。这种方法需要O(n) 的辅助空间。如果直接用数组或是bitmap来做，辅助空间的大小与数组中的最大整数相关， 常常导致大量空间浪费。比如原数组中有5个数：1亿，2亿，3亿，4亿，5亿。sum为5亿， 那么我们将bitmap中的sum-x位置1，即第4亿位，第3亿位，第2亿位，第1亿位，第0位置1. 而其它位置都浪费了。

如果使用哈希表，虽然不会有大量空间浪费，但要考虑冲突问题。

时间复杂度为O(nlogn)的解法 我们来考虑一种空间复杂度为O(1)，而且实现也很简单的算法。首先，将数组排序。 比如排序后得到的数组a是：-2 -1 0 3 5 6 7 9 13 14。然后使用low和high 两个下标指向数组的首尾元素。如果a[low]+a[high] > sum，那么说明a[high] 和数组中的任何其它一个数的和都一定大于sum(因为它和最小的a[low]相加都大于sum)。 因此，a[high]不会与数组中任何一个数相加得到sum，于是我们可以直接不要它，即让high向前移动一位。同样的，如果a[low]+a[high] < sum，那么说明a[low] 和数组中的任何其它一个数的和都一定小于sum(因为它和最大的a[high]相加都小于sum)。 因此，我们也可以直接不要它，让low向前移动一位。如果a[low]+a[high]等于sum， 则输出。当low小于high时，不断地重复上面的操作即可。

代码如下：

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

void printPairSum(int a[], int n, int sum) {
    if (a == NULL || n <= 0) return;

    sort(a, a+n);
    int low = 0, high = n-1;

    while (low < high) {
        if ( a[low] + a[high] > sum )
            --high;
        else if ( a[low] + a[high] < sum )
            ++low;
        else {
            cout << a[low] << " " << a[high] << endl;
            --high;
            ++low;
        }
    }
}

int main() {
    int n = 6, sum = 6;
    int a[] = {1, 2, 3, 4, 5, 6};
    printPairSum(a, n, sum);
    return 0;
}
```

Chapter 20

Hard

20.1 Write a function that adds two numbers. You should not use + or any arithmetic operators.

写一个Add函数求两个数的和，不能使用+号或其它算术运算符。

Solution:

为了解决这个问题，让我们来深入地思考一下，我们是如何去加两个数的。为了易于理解， 我们考虑10进制的情况。比如我们要算759 + 674，我们通常从最低位开始加， 考虑进位；然后加第二位，考虑进位...对于二进制，我们可以使用相同的方法， 每一位求和，然后考虑进位。

能把这个过程弄得更简单点吗？答案是YES，我们可以把求两个数的和分成两步，“加"与"进位"，看例子：

1. 计算759 + 674，但不考虑进位，得到323。
2. 计算759 + 674，只考虑进位，而不是去加每一位，得到1110。
3. 把上面得到的两个数加起来(这里又要用到加，所以使用递归调用即可)

由于我们不能使用任何算术运算符，因此可供我们使用的就只有位运算符了。 于是我们把操作数看成二进制表示，然后对它们做类似的操作：

1. 不考虑进位的按位求和，(0,0),(1,1)得0，(1,0),(0,1)得1， 使用异或操作可以满足要求。
2. 只考虑进位，只有(1,1)才会产生进位，使用按位与可以满足要求。 当前位产生进位，要参与高一位的运算，因此按位与后要向左移动一位。
3. 递归求和，直到进位为0

代码如下：

```
int Add2(int a, int b){
    if(b == 0) return a;
    int sum = a ^ b; // 各位相加，不计进位
    int carry = (a & b) << 1; // 记下进位
    return Add2(sum, carry); // 求sum和carry的和
}
```

递归的迭代版本如下：

```
int Add3(int a, int b){
    while(b != 0){
        int sum = a ^ b;
        int carry = (a & b) << 1;
        a = sum;
        b = carry;
    }
    return a;
}
```

对于这道题目，还有一个非常巧妙的解法。我们知道，数组操作本质上其实是指针操作。 数组名其实是指向数组首元素地址的指针。比如说整数数组a，a[1]表示的是数组中的第 1个元素，这是一直以来我们的理解。而编译器看到a[1]，它是怎么去理解的呢?

首先，它会用数组首元素地址，加上偏移量，得到目标数据的地址， 然后再把里面的数据按指针指向类型的大小取出。所以，当编译器看到a[1]， 它实际上做了下面的事：假设a指向的地址为0xbfc86d98

- 得到目标数据地址：0xbfc86d98 + sizeof(int) * 1 = 0xbfc86d9c
- 取出0xbfc86d9中的数据

从上面可以看出，操作数组元素其实隐含了加法！所以呢，如果我们要求两个数的和， 只需要把其中一个看成地址，另一个看成偏移量，然后用返回它们对应数组元素的地址即可。 看代码：

```
int Add1(int a, int b){
    char *c = (char*)a;
    return (int)&c[b]; // c+sizeof(char)*b=a + b
}
```

上述代码将a强制转换为指向char的指针c，然后返回c[b]的地址即可。c[b] 的地址就等于c + sizeof(char)*b = a + b。有人会问，它b是负数时OK吗？ OK，没问题的。它代表偏移量为负，往反方向计算地址就是了。

由于编译器对数组的解释方式如上所述，因此上面代码中的c[b]也可以写成b[c]，或 a[5]可以写成5[a]，效果是一样的，因为编译器都会先去求地址和偏移量的和。

如果还想知道更多关于c语言的奇奇怪怪的点，推荐阅读《C陷阱与缺陷》。

20.2 Write a method to shuffle a deck of cards. It must be a perfect shuffle - in other words, each 52! permutations of the deck has to be equally likely. Assume that you are given a random number generator which is perfect.

写一个随机洗牌函数。要求洗出的52!种组合都是等概率的。 也就是你洗出的一种组合的概率是1/(52!)。假设已经给你一个完美的随机数发生器。

Solution:

这是一道非常有名的面试题，及非常有名的算法——随机洗牌算法。

最直观的思路是什么？很简单，每次从牌堆中随机地拿一张出来。那么， 第一次拿有52种可能，拿完后剩下51张；第二次拿有51种可能，第三次拿有50种可能， ...，一直这样随机地拿下去直到拿完最后1张，我们就从52!种可能中取出了一种排列， 这个排列对应的概率是1/(52!)，正好是题目所要求的。

接下来的问题是，如何写代码去实现上面的算法？假设扑克牌是一个52维的数组cards， 我们要做的就是从这个数组中随机取一个元素，然后在剩下的元素里再随机取一个元素... 这里涉及到一个问题，就是每次取完元素后，我们就不会让这个元素参与下一次的选取。 这个要怎么做呢。

我们先假设一个5维数组：1，2，3，4，5。如果第1次随机取到的数是4， 那么我们希望参与第2次随机选取的只有1，2，3，5。既然4已经不用， 我们可以把它和1交换，第2次就只需要从后面4位(2,3,1,5)中随机选取即可。同理， 第2次随机选取的元素和数组中第2个元素交换，然后再从后面3个元素中随机选取元素， 依次类推。

代码如下：

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

void swap(int &a, int &b) { // 有可能swap同一变量，不能用异或版本
    int t = a;
    a = b;
    b = t;
}

void RandomShuffle(int a[], int n) {
    for (int i = 0; i < n; ++i) {
        int j = rand() % (n-i) + i; // 产生i到n-1间的随机数 // [i, n-1]
        swap(a[i], a[j]);
    }
}

int main() {
    srand( (unsigned)time(0) );
    int n = 9;
    int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    RandomShuffle(a, n);
    for (int i = 0; i < 9; ++i)
        cout << a[i] << endl;
    return 0;
}
```

20.3 Write a method to randomly generate a set of m integers from an array of size n. Each element must have equal probability of being chosen.

写一个函数，随机地从大小为n的数组中选取m个整数。要求每个元素被选中的概率相等。

Solution:

这道题目和随机洗牌问题类似，只需要随机选取1个元素， 然后在剩下的元素里面随机选取下一个元素，不断这样操作即可。

这样做能保证每个元素选中的概率一样吗? 也就是选中每个元素的概率都是1/n? 答案是YES，让我们来做一下简单的计算。

- 选第1个元素：在n个中随机选，因此概率为1/n
- 选第2个元素：在剩下的n-1个中随机选：1/(n-1)，由于第1次没有选中它， 而是在另外n-1个中选：(n-1)/n，因此概率为：(n-1)/n * 1/(n-1) = 1/n
- 选第3个元素：同上：(n-1)/n * (n-2)/(n-1) * 1/(n-2) = 1/n

。。。

因此，按照这种方法选取k个元素，每个元素都是以1/n的概率被选出来的。代码如下： 选出的m个数放到数组前m个位置。

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

void swap(int &a, int &b) { // 有可能swap同一变量，不能用异或版本
    int t = a;
    a = b;
    b = t;
}

void PickMRandomly(int a[], int n, int m) {
    for (int i = 0; i < m; ++i) {
        int j = rand() % (n-i) + i;   // 产生i到n-1间的随机数
        swap(a[i], a[j]);
    }
}

int main() {
    srand( (unsigned)time(0) );
    int n = 9, m = 5;
    int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    PickMRandomly(a, n, m);
    for (int i = 0; i < m; ++i)
        cout << a[i] << endl;
    return 0;
}
```

20.4 Write a method to count the number of 2s between 0 and n.

写一个函数，计算0到n之间2的个数。

Solution:

最简单直观的方法就是对于0到n之间的数，一个个地去统计2在它们上出现的个数， 然后累加起来即可。求2在某个数上出现的次数需要O(logn)的时间，一共有n个数， 所以共需要O(nlogn)的时间。

代码如下：

```
int Count2(int n) {
    int count = 0;
    while (n > 0) {
        if (n % 10 == 2) {
            ++count;
            n /= 10;
        }
```

```
8       }
9       return count;
10  }
11
12  int Count2s1(int n) {
13      int count = 0;
14      for (int i = 0; i <= n; ++i) {
15          count += Count2(i);
16      }
17      return count;
18  }
```

上述方法最大的问题就是效率，当n非常大时，就需要很长的运行时间。 想要提高效率，就要避开暴力法，从数字中找出规律。对于这道题， 《编程之美》已经给出了很漂亮的解法，这里再简述一下。

假设一个5位数N=abcde，我们现在来考虑百位上出现2的次数，即，从0到abcde的数中， 有多少个数的百位上是2。分析完它，就可以用同样的方法去计算个位，十位，千位， 万位等各个位上出现2的次数。

当百位c为0时，比如说12013，0到12013中哪些数的百位会出现2? 我们从小的数起， 200 299, 1200 1299, 2200 2299, ... , 11200 11299, 也就是固定低3位为200 299，然后高位依次从0到11，共12个。再往下12200 12299 已经大于12013，因此不再往下。所以，当百位为0时，百位出现2的次数只由更高位决定， 等于更高位数字(12)x当前位数(100)=1200个。

当百位c为1时，比如说12113。分析同上，并且和上面的情况一模一样。 最大也只能到11200 11299，所以百位出现2的次数也是1200个。

上面两步综合起来，可以得到以下结论：

- 当某一位的数字小于2时，那么该位出现2的次数为：更高位数字x当前位数

当百位c为2时，比如说12213。那么，我们还是有200 299, 1200 1299, 2200 2299, ... , 11200 11299这1200个数，他们的百位为2。但同时，还有一部分12200 12213， 共14个(低位数字+1)。所以，当百位数字为2时， 百位出现2的次数既受高位影响也受低位影响，结论如下：

- 当某一位的数字等于2时，那么该位出现2的次数为：更高位数字x当前位数+低位数字+1

当百位c大于2时，比如说12313，那么固定低3位为200 299，高位依次可以从0到12， 这一次就把12200 12299也包含了，同时也没低位什么事情。因此出现2的次数是： (更高位数字+1)x当前位数。结论如下：

- 当某一位的数字大于2时，那么该位出现2的次数为：(更高位数字+1)x当前位数

根据上面得出的3条结论，我们可以编写出代码如下：

```
1   int Count2s(int n) {
2       int count = 0;
3       int factor = 1;
4       int low = 0, curr = 0, high = 0;
5
6       while(n/factor != 0) {
7           low = n - (n/factor)*factor; // 低位数字
8           curr = (n/factor) % 10;     // 当前位数字
9           high = n / (factor*10);     // 高位数字
10
11          switch(curr) {
12          case 0:
13          case 1:
14              count += high * factor;
15              break;
16          case 2:
17              count += high * factor + low + 1;
18              break;
19          default:
20              count += (high + 1) * factor;
21              break;
22          }
23          factor *= 10;
```

```
24      }
25      return count;
26  }
```

如果我们把问题一般化一下：写一个函数，计算0到n之间i出现的次数，i是1到9的数。 这里为了简化，i没有包含0，因为按以上的算法计算0出现的次数， 比如计算0到11间出现的0的次数，会把1，2，3，4...视为01，02，03，04...从而得出错误的结果。所以0是需要单独考虑的，为了保持一致性，这里不做讨论。 将上面的三条结论应用到这就是：

- 当某一位的数字小于i时，那么该位出现i的次数为：更高位数字x当前位数
- 当某一位的数字等于i时，那么该位出现i的次数为：更高位数字x当前位数+低位数字+1
- 当某一位的数字大于i时，那么该位出现i的次数为：(更高位数字+1)x当前位数

代码如下：

```
1  int Countis(int n, int i) {
2      if (i < 1 || i > 9) return -1; // i只能是1到9 i should be in range [1, 9]
3
4      int count = 0;
5      int factor = 1;
6      int low = 0, curr = 0, high = 0;
7
8      while (n / factor != 0) {
9          low = n - (n/factor)*factor; // 低位数字
10         curr = (n/factor) % 10;     // 当前位数字
11         high = n / (factor*10);     // 高位数字
12
13         if (curr < i)
14             count += high * factor;
15         else if (curr == i)
16             count += high * factor + low + 1;
17         else
18             count += (high + 1) * factor;
19         factor *= 10;
20     }
21     return count;
22 }
```

20.5 You have a large text file containing words. Given any two words, find the shortest distance (in terms of number of words) between them in the file. Can you make the searching operation in O(1) time? What about the space complexity for your solution?

有一个很大的文本文件，里面包含许多英文单词。给出两个单词，找到它们的最短距离 (以它们之间隔了多少个单词计数)。你能在O(1)的时间内返回任意两个单词间的最短距离吗？ 你的解法空间复杂度是多少?

Solution:

先看一个例子，为了简单起见，我们假设文件里就只有以下两句话。然后， 我们现在来求is和name的最短距离。假设相邻的两个单词距离为1。

- What is your name My name is Hawstein

首先，我们遇到的第一个问题是：是否要考虑顺序? 我们求的是is和name间的距离， 那么文本中先出现name再出现is的情况要不要算进来。这一点是要和面试官进行交流确认的。 这里我们假设不考虑顺序，并且认为本文中只有单词，没有标点。 为了进一步简化问题，我们可以用一个字符串数组来保存单词， 接下来考虑如何计算两个单词间的最短距离。

最直观的一个解法是，遍历单词数组，遇到is或name就更新它们的位置， 然后计算is和name之间的距离，如果这个距离小于之前的最小距离，则更新这个最小距离。 看图示：

- What is your name My name is Hawstein
- 0 1 2 3 4 5 6 7
- p

p表示遍历的当前位置。此时已经经过了前面的一个is和name，is位置为1，name位置为3， 最小距离min=3-1=2。当p移动到下一个单词，发现是name，则更新name的位置为5， 减去is的位置1得到4，并不小于min，不更新，继

续。当p移动到is，更新is的位置为6， 减去name的位置5，得到距离为1，小于min，更新min=1。p之后一直移动到末尾， 没遇到is或name，不再更新。最后返回最小值min。时间复杂度O(n)，空间复杂度O(1)。

代码如下：

```
int ShortestDist(string text[], int n, string word1, string word2){
    int min = kMaxInt / 2;
    int pos1 = -min;
    int pos2 = -min;

    for(int pos=0; pos<n; ++pos){
        if(text[pos] == word1){
            pos1 = pos;
            int dist = pos1 - pos2;
            if(dist < min)
                min = dist;
        }
        else if(text[pos] == word2){
            pos2 = pos;
            int dist = pos2 - pos1;
            if(dist < min)
                min = dist;
        }
    }
    return min;
}
```

题目要求在O(1)的时间内返回两个单词的最短距离，上述代码肯定是无法满足要求的。 那要怎么做呢？只能用哈希表做预处理了，空间换时间。

方法一 遍历一次文本，用哈希函数将每个单词映射到不同结点，结点后保存该单词出现的位置。 比如对于上面的例子

- What is your name My name is Hawstein
- 0 1 2 3 4 5 6 7

遍历一次并处理后，我们得到每个单词在文本中出现的位置：(哈希值是随便写的，示意用)

- 单词 哈希值 出现位置
- What: 3 0
- is: 7 1, 6
- your: 13 2
- name: 14 3, 5
- My: 25 4
- Hawstein: 27 7

求两个单词间的最小距离时，首先用O(1)时间通过哈希函数映射到指定结点， 然后对于其中一个单词的每个位置，去与第二个单词的所有位置比较，找到最小的差值。 由于位置是递增的，因此可以修改二分查找进行搜索。

该方法的平均查找复杂度应该是O(1)的，但最坏情况下无法保证O(1)的查找时间， 考虑一种极端情况，文本中的单词就只有is和name，它们的数量各为()n， 使用这种算法，我们需要O(nlogn)的时间。

方法二 预处理阶段把文本中任意两个单词间的最小距离计算出来， key是两个单词连接后的哈希值，value保存的就是最小距离。 查找阶段就只需要把两个单词连接求其哈希值，然后直接返回其对应的value即可。 查找两个单词的最小距离时间复杂度O(1)。需要O(n2)的时间来做预处理。

由于我们是不考虑顺序的，因此做两个单词的连接时，不能直接连接， 这样会导致is和name连接后是isname，而name和is连接后nameis， 它们的哈希值不一样，这并不是我们想要的。因此，在做两个单词的连接时， 我们可以让第一个字符较小的单词放在前面(反正定义一个规则来保证连接的唯一性即可)。 比如对于name和is，由于在字典序中，i<n，所以连接是isname。

还是用上面的例子，预处理后得到：(哈希值是随便写的数字，示意用)

- 单词连接 哈希值 最小距离
- (isWhat) 8 1
- ... ...
- (isname) 12 1
- ... ...
- (isMy) 33 2
- ... ...

这样当我要求is和name之间的最小距离时，就只需要先连接它们得到isname， 然后用哈希函数求出isname的哈希值12，然后直接返回它对应的最小距离即可。

如果有冲突怎么办? 即两个不同的字符串映射到同一个哈希值，我们可以用链地址法， 把冲突的连接字符串链接起来，这样每个结点就需要保存连接字符及其对应的最小距离。 比如对于上面的例子，假设isname和isMy的哈希值相同，我们可以按如下所示去做:

- 哈希值　最小距离
- 8　　(isWhat,1)
- ...　　...
- 12　　(isname,1) -> (isMy,2)
- ...　　...

这样一来，当我们求得一个连接字符串str的哈希值是12， 就依次去与其后面的结点做比较。如果str等于isname，返回1；否则，移动到下一个结点， 继续比较。如果str等于isMy，返回2。

方法三 也可以先将两个单词分别映射到两个哈希值，比如is映射到哈希值i，name映射到哈希值j， 然后将它们的最小距离保存在d[i][j]中。这里由于是不考虑单词顺序的，因此， 我们可以将较小的哈希值放在d的第一维，较大的放在第二维。也就是对于d[i][j]， 有i<j。同样，这种方法也要考虑冲突问题。

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

const int kMaxInt = ~(1<<31);

int ShortestDist(string text[], int n, string word1, string word2) {
    int min = kMaxInt / 2;
    int pos1 = -min;
    int pos2 = -min;

    for (int pos = 0; pos < n; ++pos) {
        if (text[pos] == word1) {
            pos1 = pos;
            int dist = pos1 - pos2;
            if (dist < min)
                min = dist;
        }
        if (text[pos] == word2) {
            pos2 = pos;
            int dist = pos2 - pos1;
            if (dist < min)
                min = dist;
        }
    }
    return min;
}

int main() {
    string text[] = {"What", "is", "your", "name", "My", "name", "is", "Hawstein"};
    int len = 8;
    cout << ShortestDist(text, len, "is", "name") << endl;
    return 0;
}
```

20.6 Describe an algorithm to find the largest 1 million numbers in 1 billion numbers. Assume that the computer memory can hold all one billion numbers.

描述一个算法，在10亿个数中找到最大的1百万个数。假设内存可以一次性装入这10亿个数。

Solution:

虽然这道题的数据量很大，但由于题目已经假设所有的数据可以一次性装入内存， 所以题目中的10亿，1百万也就没有什么特殊含义了。 我们完全可以想像成在100个数中查找最大的10个数。

这是一道经典的面试题，一般有以下几种解法:

排序法 最直观的方法就是将数组从大到小排序，然后取前1百万个数即可。时间复杂度O(nlogn)。

最小堆 利用最小堆来维护最大的1百万个数，堆顶元素是这1百万个数中最小的。遍历剩下的元素，当某一元素大于堆顶元素，则用该元素替换堆顶元素，然后调整堆结构，使其仍为最小堆。当遍历完所有10亿个数后，堆中维护的就是最大的1百万个数。在n个数中查找最大的k个数，该算法需要O(nlogk) 的时间。由于k一般要比n小得多，所以该算法比排序法要快。

该算法还有一个优点，就是便于处理大数据。比如说，我们一般需要在非常多的数中找到最大(最小)的k个数，这个k一般比较小，而n却可能大得无法一次性载入内存。这时候我们就可以在内存中维护一个k 个元素的最小(最大)堆，然后把数据分多次从磁盘读入内存进行处理。

线性求k最大 线性求k最大利用的是快排中的partition函数。每次选取一个基准元素pivot (可以用第1个元素，也可以随机选)，然后将其它元素与pivot对比。大于等于pivot 的放到左边，小于pivot的放到右边。调用一次partition后，pivot左边的数都是大于等于它的，pivot右边的数都是小于它的。如果pivot此时正好是第k-1个元素，那么它左边加上它一共有k个元素，而且这k个元素都是比右边的元素要大的，即它们就是最大的k个元素。如果pivot 左边不足k-1个元素，则在它右边进行同样的partition操作。如果pivot 左边是多于k-1个元素的，则在它左边进行partition操作。

该算法会改变数组中元素的顺序，期望时间复杂度是O(n)。

20.7 Write a program to find the longest word made of other words.

写一个程序，找到由其它单词组成的最长单词。

Solution:

我们从例子着手开始分析问题。假如我们有一个单词数组，如下:

```
1 string arr[] = {"test", "tester", "testertest", "testing", "apple", "seattle", "banana",
"batting", "ngcat", "batti","bat", "testingtester", "testbattingcat"};
```

哪一个是题目要求的最长单词? 这时候假如你有另一个"我"能跳出来，观察自己的思考过程。你就会发现自己是怎么去解这个问题的，然后只需要把你的思考过程变成算法，写成代码就OK了。

题目说要找最长单词，所以你的眼睛自然会去寻找那些长单词，至少你不会从bat 开始找起，对吧。找到最长的单词是testbattingcat，下一步去看它是否可以由其它单词组成。我们发现test是testbattingcat的一部分，bat也是它的一部分，然后呢? 剩下的tingcat不能由其它单词构成。不过，我们可以用test，batti，ngcat来组成它。所以，它就是我们要找的可以由其它单词组成的最长单词。

把上面的思考过程转换成算法，可以描述如下:

1. 按单词的长度从大到小排序。(先寻找最长的单词)
2. 不断地取单词的前缀s，当s存在于单词数组中，递归调用该函数，判断剩余串是否可以由其它单词组成。如果可以，返回true。

对于上面的例子testbattingcat，我们通过不断取前缀：t不在数组中，te不在数组中，tes不在数组中，test在数组中；递归调用去处理剩余串battingcat，b不在数组中，ba不在数组中，bat在数组中；递归调用去处理剩余串tingcat，发现它所有的前缀都不存在于数组中，退递归来到处理battingcat那一层。接着上次的bat继续处理：batt不在数组中，batti在数组中；递归调用去处理剩余串 ngcat，n，ng，ngc，ngca都不在数组中，ngcat存在数组中。递归调用处理剩余串，发现剩余串为空，返回真。

代码如下:

```
1  #include <cstring>
2
3  const int kWordSize = 26 + 5;
4  const int kNodeSize = 1200 + 5;
5  const int kHashSize = 10001;
6
7  struct Node{
8      char word[kWordSize];
9      Node *next;
10 };
11 Node node[kNodeSize + 1];
12 Node* head[kHashSize + 1];
13
14 class Hash{
15 public:
16     Hash();
17     unsigned int hash(const char* str);
18     void insert(const char* str);
```

```
19      bool find(const char* str);
20  private:
21      unsigned int seed;
22      unsigned int size;
23  };
24
25  Hash::Hash():seed(131),size(0){
26      memset(head, 0, sizeof(head));
27  }
28
29  unsigned int Hash::hash(const char* str){
30      unsigned int hash = 0;
31      while(*str++)
32          hash = hash * seed + (*str);
33      return (hash & 0x7FFFFFFF) % kHashSize;
34  }
35
36  void Hash::insert(const char* str){
37      unsigned int id = hash(str);
38      char *dst = (char*)node[size].word;
39      while(*dst++ = *str++);
40      node[size].next = head[id];
41      head[id] = &node[size];
42      ++size;
43  }
44
45  bool Hash::find(const char* str){
46      unsigned int id = hash(str);
47      for(Node* p=head[id]; p ; p=p->next){
48          char *dst = (char*)p->word;
49          int i = 0;
50          while(*(str+i) && *(dst+i)==*(str+i))
51              ++i;
52          if(*(str+i)=='\0' && *(dst+i)=='\0')
53              return true;
54      }
55      return false;
56  }
```

```
1   #include <iostream>
2   #include <algorithm>
3   #include "hash.h"
4   using namespace std;
5
6   Hash hash;
7
8   inline bool cmp (string s1, string s2) { //按长度从大到小排
9       return s2.length() < s1.length();
10  }
11
12  bool MakeOfWords(string word, int length) {
13      int len = word.length();
14      if (len == 0) return true;
15
16      for (int i = 0; i <= len; ++i) {
17          if (i == length) return false; //取到原始串，即自身
18          string str = word.substr(0, i);
19          if ( hash.find((char*)&str[0]) ) {
20              if ( MakeOfWords(word.substr(i), length) )
```

```
21                  return true;
22              }
23          }
24          return false;
25      }
26
27      void PrintLongestWord(string word[], int n) {
28          for (int i = 0; i < n; ++i)
29              hash.insert( (char*)&word[i][0] );
30          sort(word, word+n, cmp);
31
32          for (int i = 0; i < n; ++i) {
33              if ( MakeOfWords(word[i], word[i].length()) ) {
34                  cout << "Longest Word: " << word[i] << endl;
35                  return;
36              }
37          }
38      }
39
40      int main() {
41          string arr[] = {"test", "tester", "testertest", "testing",
42                          "apple", "seattle", "banana", "batting", "ngcat",
43                          "batti", "bat", "testingtester", "testbattingcat"};
44          int len = 13;
45          PrintLongestWord(arr, len);
46          return 0;
47      }
```

上述代码将单词存放在哈希表中，以得到O(1)的查找时间。排序需要用O(nlogn)的时间， 判断某个单词是否可以由其它单词组成平均需要O(d)的时间(d为单词长度)， 总共有n个单词，需要O(nd)的时间。所以时间复杂度为：O(nlogn + nd)。 n比较小时，时间复杂度可以认为是O(nd)；n比较大时，时间复杂度可以认为是O(nlogn)。

注意上述代码中有一句：

```
1   if(i == length) return false;//取到原始串，即自身
```

意思是当我们取一个单词前缀，最后取到整个单词时， 这种情况就认为是没有其它单词可以组成它。如果不要这一句， 那么你在哈希表中总是能查找到和它自身相等的串(就是它自己)，从而返回true。 而这明显不是我们想要的。我们要的是其它单词来组成它，不包括它自己。

这样一来，又引出一个问题，如果单词中就是存在两个相同的单词。 比如一个单词数组中最长的单词是abcdefg，并且存在2个，而它又不能被更小的单词组成， 那么我们可以认为这个abcdefg是由另一个abcdefg组成的吗？ 关于这一点，你可以和面试官进行讨论。(上述代码认为是不能的。)

由于使用哈希表会占用较多空间，一种不使用额外空间的算法是直接在单词数组中查找， 由于单词数组已经按长度从大小到排，因此单次查找时间为O(n)。一共有n个单词， 平均长度为d，所以总共需要的时间为O(nd*n)=O(dn2)。 如果我们再开一个数组来保存所有单词，并将它按字典序排序， 那么我们可以使用二分查找，单次查找时间变为O(logn)，总共需要O(dnlogn)。

20.8 Given a string s and an array of smaller strings T, design a method to search s for each small string in T.

给一个字符串S和一个字符串数组T(T中的字符串要比S短许多)，设计一个算法， 在字符串S中查找T中的字符串。

Solution:

字符串的多模式匹配问题。

我们把S称为目标串，T中的字符串称为模式串。设目标串S的长度为m，模式串的平均长度为 n，共有k个模式串。如果我们用KMP算法(或BM算法)去处理每个模式串， 判断模式串是否在目标串中出现， 匹配一个模式串和目标串的时间为O(m+n)，所以总时间复杂度为：O(k(m+n))。 一般实际应用中，目标串往往是一段文本，一篇文章，甚至是一个基因库， 而模式串则是一些较短的字符串，也就是m一般要远大于n。 这时候如果我们要匹配的模式串非常多(即k非常大)，那么我们使用上述算法就会非常慢。 这也是为什么KMP或BM一般只用于单模式匹配，而不用于多模式匹配。

那么有哪些算法可以解决多模式匹配问题呢？貌似还挺多的，Trie树，AC自动机，WM算法， 后缀树等等。我们先从简单的Trie树入手来解决这个问题。

Trie树，又称为字典树，单词查找树或前缀树，是一种用于快速检索的多叉树结构。 比如英文字母的字典树是一个26叉树，数字的字典树是一个10叉树。

Trie树可以利用字符串的公共前缀来节约存储空间，这也是为什么它被叫前缀树。

如果我们有以下单词：abc, abcd, abd, b, bcd, efg, hig, 可以构造如下Trie树： (最右边的最后一条边少了一个字母)

http://hawstein.com/posts/20.8.html

回到我们的题目，现在要在字符串S中查找T中的字符串是否出现(或查找它们出现的位置)， 这要怎么和Trie扯上关系呢?

假设字符串S = “abcd"，那么它的所有后缀是：

- abcd
- bcd
- cd
- d

我们发现，如果一个串t是S的子串，那么t一定是S某个后缀的前缀。比如t = bc， 那么它是后缀bcd的前缀；又比如说t = c，那么它是后缀cd的前缀。

因此，我们只需要将字符串S的所有后缀构成一棵Trie树(后缀Trie)， 然后查询模式串是否在该Trie树中出现即可。如果模式串t的长度为n， 那么我们从根结点向下匹配，可以用O(n)的时间得出t是否为S的子串。

下图是BANANAS的后缀Trie:

后缀Trie的查找效率很优秀，如果你要查找一个长度为n的字符串，只需要O(n)的时间， 比较次数就是字符串的长度，相当给力。 但是，构造字符串S的后缀Trie却需要O(m2)的时间， (m为S的长度)，及O(m2)的空间。

代码如下：

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

class Trie {
public:
    static const int MAX_N = 100 * 100; // 100为主串长度
    static const int CLD_NUM = 26;      // 每个结点的儿子数量(26个字母)
    int size;       // 用到的当前结点编号
    int trie[MAX_N][CLD_NUM];

    Trie(const char* s);
    void insert(const char* s);
    bool find(const char* s);
};

Trie::Trie(const char* s) {
    memset(trie[0], -1, sizeof(trie[0]));
    size = 1;
    while(*s) {
        insert(s);
        ++s;
    }
}

void Trie::insert(const char* s) {
    int p = 0;
    while (*s) {
        int i = *s - 'a';
        if ( -1 == trie[p][i] ) {
            memset( trie[size], -1, sizeof(trie[size]) );
            trie[p][i] = size++;
        }
        p = trie[p][i];
        ++s;
    }
}

bool Trie::find(const char* s) {
```

```
40      int p = 0;
41      while(*s) {
42          int i = *s - 'a';
43          if ( -1 == trie[p][i] )
44              return false;
45          p = trie[p][i];
46          ++s;
47      }
48      return true;
49  }
50
51  int main() {
52      Trie tree("mississippi");
53      string patt[] = {"is", "sip", "hi", "sis", "mississippa"};
54      int n = 5;
55      for (int i = 0; i < 5; ++i)
56          cout << tree.find((char*)&patt[i][0]) << endl;
57      return 0;
58  }
```

上述方法总的时间复杂度是：O(m2 + kn)，有没更快的算法呢？答案是肯定的。 下面就来简单介绍一下AC自动机。

AC自动机算法是解决字符串多模式匹配的一个经典方法，时间复杂度为：O(m+kn+z)， 其中：m是目标串S的长度，k是模式串个数，n是模式串平均长度，z是S 中出现的模式串数量。从时间复杂度上可以看出，AC自动机比后缀Trie方法要快， m从2次方降到了1次方。

AC自动机也会先构造一棵Trie树，不同的是，它用模式串来构造Trie树。 然后遍历一次目标串S，即可求出哪些模式串出现在目标串S中。

关于AC自动机，比较好的资料是： Set Matching and Aho-Corasick Algorithm (http://www.cs.uku.fi/~kilpelai/BSA05/lectures/slide 是 生物序列算法(http://www.cs.uku.fi/~kilpelai/BSA05/)课的一个课件， 这个课的课件基本上都是关于字符串算法的，讲得挺好，推荐一读。

AC自动机代码如下：

```
1   #include <iostream>
2   #include <queue>
3   #include <cstring>
4   using namespace std;
5
6   class ACAutomation {
7   public:
8       static const int MAX_N = 1000 * 50;
9       static const int CLD_NUM = 26;
10
11      int size;
12      int fail[MAX_N];
13      int tag[MAX_N];
14      int trie[MAX_N][CLD_NUM];
15
16      void reset();
17      void insert(const char* s);
18      void construct();
19      int query(const char* s);
20  };
21
22  void ACAutomation::reset() {
23      memset( trie[0], -1, sizeof(trie[0]) );
24      tag[0] = 0;
25      size = 1;
26  }
27
28  void ACAutomation::insert(const char* s) {
```

```
29      int p = 0;
30      while (*s) {
31          int i = *s - 'a';
32          if ( -1 == trie[p][i] ) {
33              memset( trie[size], -1, sizeof(trie[size]) );
34              tag[size] = 0;
35              trie[p][i] = size++;
36          }
37          p = trie[p][i];
38          ++s;
39      }
40      ++tag[p];
41  }
42
43  void ACAutomation::construct() {
44      queue<int> q;
45      fail[0] = 0;
46      for (int i = 0; i < CLD_NUM; ++i) {
47          if ( trie[0][i] != -1 ) {
48              fail[ trie[0][i] ] = 0;
49              q.push( trie[0][i] );
50          } else
51              trie[0][i] = 0;
52      }
53      while ( !q.empty() ) {
54          int u = q.front();
55          q.pop();
56          for (int i = 0; i < CLD_NUM; ++i) {
57              int &v = trie[u][i];
58              if (v != -1) {
59                  q.push(v);
60                  fail[v] = trie[ fail[u] ][i];
61                  // tag[u] += tag[ fail[u] ];
62              } else
63                  v = trie[ fail[u] ][i];
64          }
65      }
66  }
67
68  //返回匹配的模式串个数
69  int ACAutomation::query(const char* s) {
70      int p = 0, cnt = 0;
71      while (*s) {
72          int i = *s - 'a';
73          while ( trie[p][i] == -1 && p != 0 ) //无法匹配当前字符，回退到其fail指针
74              p = fail[p];
75          p = trie[p][i];
76          p = p==-1 ? 0 : p;
77          int t = p;
78          while (t != 0 && tag[t] != -1) {
79              cnt += tag[t];
80              tag[t] = -1;
81              t = fail[t]; //跳到当前字符的最大后缀，统计模式串出现个数
82          }
83          ++s;
84      }
85      return cnt;
86  }
87
```

```
88  int main() {
89      ACAutomation ac;
90      ac.reset();
91      string patt[] = {"is", "sip", "is", "sis", "mississipp"};
92      int n = 5;
93      for (int i = 0; i < n; ++i)
94          ac.insert( (char*)&patt[i][0] );
95      ac.construct();
96      cout << ac.query("mississippi") << endl;
97      return 0;
98  }
```

对于这道题目，还有更给力的解法：Ukkonen算法， 该算法可以在线性时间和空间下构造后缀树，而且它还是在线算法！ 不过它实现起来就没那么简单了，尤其是3个加速部分，分分钟把人弄晕。 上面说到的生物序列算法课件对此也有不错的讲解，同时推荐Gusfield的书： Algorithms on strings, trees, and sequences。

另外推荐一些资料：

- 一篇介绍后缀树的好文章：(附代码) Fast String Searching With Suffix Trees: (http://marknelson.us/1996/08/01/suffix-trees/)
- 三鲜对上文的翻译(原文找不到了，这是别人转三鲜的)：后缀树的构造方法-Ukkonen详解: http://blog.163.com/lazy_p/blog/sta
- 下面一篇来自stackoverflow： Ukkonen's suffix tree algorithm in plain English: (http://stackoverflow.com/questions/9452701/ul suffix-tree-algorithm-in-plain-english)
- 上文的译文： 介绍后缀树: (http://www.ibaiyang.org/2013/01/06/suffix-tree-introduction/)

20.9 Numbers are randomly generated and passed to a method. Write a program to find and maintain the median value as new values are generated.

随机产生一些数传递给一个函数，写程序找出并维护这些数的中位数。

Solution:

方法一 最简单直观的方法是用一个足够大的数组A来维护这些数，使其按升序排列。 这样一来，可以用O(1)的时间找到中位数。下面是图示：

- 元素：1 3 5　　元素：1 3 5 7
- 下标：0 1 2　　下标：0 1 2 3
- 中位数：A[n/2]　　中位数：(A[(n-1)/2] + A[n/2])/2

这种方法插入一个新来的元素需要O(n)的时间， 需要在原来有序的数组中找到一个合适的位置插入它，并把比它大的元素都向后移动1位。

方法二 用一个最大堆(或最小堆)来维护这些数。那么插入一个新元素需要O(logn)的时间， 比方法一要好。但取中位数需要先排序，时间复杂度O(nlogn)。

方法三 使用堆来维护数据是个不错的选择，因为插入一个新元素只需要O(logn)的时间， 但取中位数比较耗时，时间主要花在排序上。有没方法可以不排序呢？ 我们知道，中位数是一个有序序列中排在中间的数，它左右两边的数相当。 从方法一的示意图可以看出，当数组大小n为奇数时，中位数就是有序序列中正中间那个数， 如果n为偶数，它是中间两个数的平均数。它只和序列中间的一个或两个数有关， 和其它的元素无关。那么，如果我用一个最大堆维护中位数左边的数(包含它)， 用一个最小堆维护中位数右边的数。当n为偶数时，我只需要把左边的数最大那个， 和右边的数最小那个相加除以2即可。左边的最大数即最大堆的堆顶元素， 右边最小数即最小堆的堆顶元素。当n为奇数时，如果最大堆的元素比最小堆的元素多1， 则最大堆的堆顶元素是中位数；如果最小堆的元素比最大堆的元素多1， 则最小堆的堆顶元素是中位数。在插入新元素的时候，我们只要维护两个堆， 使其堆中元素的数量差别不超过1即可。

这样一来，插入新元素还是O(logn)的时间，而取中位数只需要O(1)的时间， 要优于方法一和方法二。

```
1   #include <iostream>
2   #include <algorithm>
3   #include <queue>
4   #include <cstdlib>
5   using namespace std;
6
7   class Median {
8   private:
9       priority_queue< int, vector<int>, less<int> > max_heap;    // 左边的数
10      priority_queue< int, vector<int>, greater<int> > min_heap;// 右边的数
11  public:
```

```
12      void Insert(int v);
13      int GetValue();
14  };
15
16  void Median::Insert(int v) {
17      if ( max_heap.empty() && min_heap.empty() )
18          max_heap.push(v);
19      else if ( !max_heap.empty() && min_heap.empty() )
20          max_heap.push(v);
21      else if ( max_heap.empty() && !min_heap.empty() )
22          min_heap.push(v);
23      else {
24          if (v < max_heap.top() )
25              max_heap.push(v);
26          else
27              min_heap.push(v);
28      }
29
30      // 调整，保证两个堆的元素数量差别不大于1
31      // 不要用max_heap.size()-min_heap.size()>1
32      // 因为size返回的是unsigned类型，当左边相减得到一个负数时，本来为false
33      // 但会被转为一个大的正数，结果为true，出问题
34      while( max_heap.size() > min_heap.size() + 1) {
35          int data = max_heap.top();
36          min_heap.push(data);
37          max_heap.pop();
38      }
39      while( min_heap.size() > max_heap.size() + 1) {
40          int data = min_heap.top();
41          max_heap.push(data);
42          min_heap.pop();
43      }
44  }
45
46  int Median::GetValue() {   //中位数为int，由于有除法，也可改为float
47      if (max_heap.empty() && min_heap.empty())
48          return (1<<31);    //都为空时，返回int最小值
49
50      if (max_heap.size() == min_heap.size())
51          return (max_heap.top() + min_heap.top())/2;
52      else if (max_heap.size() < min_heap.size())
53          return min_heap.top();
54      else
55          return max_heap.top();
56  }
57
58  int main() {
59      srand( (unsigned)time(0) );
60      Median md;
61      vector<int> vi;
62      int num = rand() % 30;   //数量是30以内的随机数
63      for (int i = 0; i < num; ++i) {
64          int data = rand() % 100; //元素是100内的数
65          vi.push_back(data);
66          md.Insert(data);
67      }
68      sort(vi.begin(), vi.end());
69      for (int i = 0; i < num; ++i)
70          cout << vi.at(i) << "␣";          //排序的序列
```

```
71      cout << endl << md.GetValue() << endl;   //中位数
72      return 0;
73  }
```

20.10 Given two words of equal length that are in a dictionary, write a method to transform one word into another word by changing only one letter at a time. The new word you get in each step must be in the dictionary.

EXAMPLE

Input: DAMP, LIKE

Output: DAMP -> LAMP -> LIMP -> LIME -> LIKE

20.11 Imagine you have a square matrix, where each cell is filled with either black or white. Design an algorithm to find the maximum subsquare such that all four borders are filled with black pixels.

有一个正方形矩阵，里面的每一个小格子要么被涂上黑色要么被涂上白色。 设计算法，找到四条边都是黑色格子的最大子正方形。

Solution:

暴力法，从左到右，从上到下遍历格子，将它作为子正方形左上角的点。 固定了子正方形左上角的点，我们只需要知道边长，就能把子正方形确定下来。 我们按边长从大到小开始，去检查每一个子正方形的四条边是否都为黑色格子。 如果是，则记下当前最大的边长值。将子正方形左上角的点移动到下一行(即向下移动一格)， 进入下一轮循环。

代码如下:

```
1   #include <iostream>
2   #include <cstdio>
3   using namespace std;
4
5   const int MAX_N = 100;
6   int matrix[MAX_N][MAX_N];
7
8   struct SubSquare {
9       int row, col, size;
10  };
11
12  inline int max(int a, int b) {
13      return a > b ? a : b;
14  }
15
16  bool IsSquare(int row, int col, int size) {
17      for (int i = 0; i < size; ++i){
18          if (matrix[row][col+i] == 1)    // 1代表白色，0代表黑色 1: while; 0: black
19              return false;
20          if (matrix[row+size-1][col+i] == 1)
21              return false;
22          if (matrix[row+i][col] == 1)
23              return false;
24          if (matrix[row+i][col+size-1] == 1)
25              return false;
26      }
27      return true;
28  }
29
30  SubSquare FindSubSquare(int n) {
31      int max_size = 0;    //最大边长 // max length for one size
32      int col = 0;
33      SubSquare sq;
34      while (n-col > max_size) {
35          for (int row = 0; row < n; ++row) {
36              int size = n - max(row, col);
37              while (size > max_size) {
38                  if ( IsSquare(row, col, size) ) {
39                      max_size = size;
```

```
40                          sq.row = row;
41                          sq.col = col;
42                          sq.size = size;
43                          break;
44                      }
45                      --size;
46                  }
47              }
48              ++col;
49          }
50          return sq;
51      }
52
53      int main() {
54          freopen("ch20.11.in", "r", stdin);
55          int n;
56          cin >> n;
57          for (int i = 0; i < n; ++i)
58              for (int j = 0; j < n; ++j)
59                  cin >> matrix[i][j];
60          SubSquare sq = FindSubSquare(n);
61          cout << "top: " << sq.row << endl;
62          cout << "bottom: " << sq.row + sq.size -1 << endl;
63          cout << "left: " << sq.col << endl;
64          cout << "right: " << sq.col + sq.size -1 << endl;
65          fclose(stdin);
66          return 0;
67      }
```

20.12 Given an NxN matrix of positive and negative integers, write code to find the submatrix with the largest possible sum.

给一个NxN的矩阵，矩阵上每个格子中填有一个整数(正，负或0)， 写代码计算子矩阵之和的最大值。

Solution:

暴力法 ：时间复杂度O(n6)

最简单粗暴的方法就是枚举所有的子矩阵，求和，然后找出最大值。 枚举子矩阵一共有C(n, 2)*C(n, 2)个(水平方向选两条边，垂直方向选两条边)， 时间复杂度O(n4)，求子矩阵中元素的和需要O(n2)的时间。 因此总的时间复杂度为O(n6)。

部分和预处理 ：时间复杂度降到O(n4)

上面的方法需要O(n2)去计算子矩阵中元素的和。 这一部分我们可以在预处理的时候求出部分和，在使用的时候就只需要O(1) 的时间来得到子矩阵中元素的和。

我们用一个二维数组p来保存矩阵的部分和，p[i][j]表示左上角是(1, 1)，(下标从1开始)， 右下角是(i, j)的矩阵中元素的和。这样一来，如果我们要求矩阵(x1, x2, y1, y2) 中元素的和(即上图矩阵D)，我们可以通过以下式子计算得出：

```
1  sum(D) = p[y2][x2] - p[y2][x1-1] - p[y1-1][x2] + p[y1-1][x1-1]
```

只需要O(1)的时间。

部分和p[i][j]要怎么计算呢? 我们可以通过更小的部分和来计算得到它：

```
1  p[i][j] = p[i-1][j] + p[i][j-1] - p[i-1][j-1] + A[i][j]
```

其中A[i][j]是格子(i, j)中的整数。我们只需要O(n2) 的时间即可预处理得到所有的部分和。

降维 ：O(n3)的解法

如果有一个一维的数组，我们要求它子数组之和的最大值，最好的时间复杂度是O(n)。

既然如此，我们可以把二维数组一个方向的数累加起来，将它变为一维数组， 然后就转化成了求一维数组子数组之和的最大值。看示意图：

第k列 第l列 第i行：... ... ... 第j行：... ... ... 在同一列中，我们把第i行到第j行的数加起来，得到如下：

第k列 第l列 只剩下一行：... ... ... 这时候我们可以用O(n)的时候算出子数组之和的最大值，假设是第k个元素到第l 个元素的子数组。那么它实际上就对应二维数组中第i，j行，第k，l 列组成的子矩阵的元素和。

枚举i，j行需要O(n2)的时间，求一维情况的子数组最大和需要O(n)的时间， 所以总的时间复杂度为O(n3)。其中求第k列元素中， 第i行到第j行的元素和可以用部分和求解，仅需要O(1)的时间：

```
sum(i,j,k) = p[j][k] - p[j][k-1] - p[i-1][k] + p[i-1][k-1]
```

代码如下：

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

const int MAX_N = 100;
int p[MAX_N][MAX_N], A[MAX_N][MAX_N];

void PreCompute(int n){
    for (int i = 0; i <= n; ++i)
        p [0][i] = p[i][0] = 0;
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j)
            p[i][j] = p[i-1][j] + p[i][j-1] - p[i-1][j-1] + A[i][j];
}

int MaxSum(int n) {
    int max_sum = 1 << 31; // Min Int
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j) {
            int curr_sum = 0;
            for (int k = 1; k <= n; ++k) {
                int val = p[j][k] - p[j][k-1] - p[i-1][k] + p[i-1][k-1];
                if (curr_sum <= 0)
                    curr_sum = val;
                else
                    curr_sum += val;
                if (curr_sum > max_sum)
                    max_sum = curr_sum;
            }
        }
    return max_sum;
}

int main() {
    freopen("ch20.12.in", "r", stdin);
    int n;
    cin >> n;
    for (int i = 1; i <= n; ++i)    //元素存储从1开始
        for (int j = 1; j <= n; ++j)
            cin >> A[i][j];
    PreCompute(n);
    cout << MaxSum(n) << endl;
    fclose(stdin);
    return 0;
}
```

20.13 Given a dictionary of millions of words, give an algorithm to find the largest possible rectangle of letters such that every row forms a word (reading left to right) and every column forms a word (reading top to bottom).