```
1.  //BinaryTree
2.  #include<iostream>
3.  #include<stack>
4.  #include<queue>
5.  using namespace std;
6.
7.  class btNode {
8.      public:
9.          double value;
10.         btNode* left;
11.         btNode* right;
12.     public:
13.         btNode() {
14.             this->value=0.0;
15.             this->left=NULL;
16.             this->right=NULL;
17.         }
18.         btNode(double m_value=0.0) {
19.             this->value=m_value;
20.             this->left=NULL;
21.             this->right=NULL;
22.         }
23.         ~btNode() {
24.         }
25.         void setLeft(double m_value) {
26.             this->left=new btNode(m_value);
27. //          this->left->value=m_value;
28.         }
29.         void setRight(double m_value) {
30.             this->right=new btNode(m_value);
31. //          this->right->value=m_value;
32.         }
33. //      int printSelf();
34.         //三种非递归遍历（栈）
35.         int preGo1();
36.         int miGo1();
37.         int postGo1();
38.         //层序遍历 （队列）
39.         int levelGo();
40.         //三种递归遍历
41.         int preGo2();
42.         int miGo2();
43.         int postGo2();
44. };
```

```
45.
46. int btNode::preGo1() {
47.     stack<pair<btNode*,bool>> s;
48.     btNode *p;
49.     bool visited;
50.     //false 表示此点是第一次进栈
51.     s.push(make_pair(this,false));
52.     while(!s.empty()) {
53.         p=s.top().first;
54.         visited=s.top().second;
55.         s.pop();
56.         if(p==NULL){
57.             continue;
58.         }
59.         if(visited){
60.             cout<<"v: "<<p->value<<endl;
61.         }else{
62.             s.push(make_pair(p->right,false));
63.             s.push(make_pair(p->left,false));
64.             s.push(make_pair(p,true));
65.         }
66.     }
67.     return 0;
68. }
69.
70. int btNode::miGo1() {
71.     stack<pair<btNode*,bool>> s;
72.     btNode *p;
73.     bool visited;
74.     //false 表示此点是第一次进栈
75.     s.push(make_pair(this,false));
76.     while(!s.empty()) {
77.         p=s.top().first;
78.         visited=s.top().second;
79.         s.pop();
80.         if(p==NULL){
81.             continue;
82.         }
83.         if(visited){
84.             cout<<"v: "<<p->value<<endl;
85.         }else{
86.             s.push(make_pair(p->right,false));
87.             s.push(make_pair(p,true));
88.             s.push(make_pair(p->left,false));
```

```
        }
    }
    return 0;
}

int btNode::postGo1() {
    stack<pair<btNode*,bool>> s;
    btNode *p;
    bool visited;
    //false 表示此点是第一次进栈
    s.push(make_pair(this,false));
     while(!s.empty()) {
         p=s.top().first;
         visited=s.top().second;
         s.pop();
         if(p==NULL){
             continue;
         }
         if(visited){
             cout<<"v: "<<p->value<<endl;
         }else{
             s.push(make_pair(p,true));
             s.push(make_pair(p->right,false));
             s.push(make_pair(p->left,false));
         }
    }
    return 0;
}

int btNode::levelGo(){
    queue<btNode*> q;
    q.push(this);
    btNode* p;
    while(!q.empty()){
         p=q.front();
         q.pop();
         cout<<"v: "<<p->value<<endl;
         if(p->left!=NULL){
             q.push(p->left);
         }
         if(p->right!=NULL){
             q.push(p->right);
         }
    }
```

```
133. }
134. 
135. int btNode::preGo2() {
136.     cout<<"v: "<<this->value<<endl;
137.     if(this->left!=NULL) {
138.         this->left->preGo2();
139.     }
140.     if(this->right!=NULL) {
141.         this->right->preGo2();
142.     }
143.     return 0;
144. }
145. 
146. int btNode::miGo2() {
147.     if(this->left!=NULL) {
148.         this->left->miGo2();
149.     }
150.     cout<<"v: "<<this->value<<endl;
151.     if(this->right!=NULL) {
152.         this->right->miGo2();
153.     }
154.     return 0;
155. }
156. 
157. int btNode::postGo2() {
158.     if(this->left!=NULL) {
159.         this->left->postGo2();
160.     }
161.     if(this->right!=NULL) {
162.         this->right->postGo2();
163.     }
164.     cout<<"v: "<<this->value<<endl;
165.     return 0;
166. }
167. 
168. int main() {
169.     double a=0,b=1,c=2,d=3,e=4;
170.     //    0
171.     //  1   2
172.     // 3 4
173.     btNode *root=new btNode(a);
174.     root->setLeft(b);
175.     root->setRight(c);
176.     root->left->setLeft(d);
```

```
177.     root->left->setRight(e);
178.     cout<<"迭代式前序遍历："<<endl;
179.     root->preGo1();
180.     cout<<"迭代式中序遍历："<<endl;
181.     root->miGo1();
182.     cout<<"迭代式后序遍历："<<endl;
183.     root->postGo1();
184.     cout<<"----------------------"<<endl;
185.     cout<<"层序遍历："<<endl;
186.     root->levelGo();
187.     cout<<"----------------------"<<endl;
188.     cout<<"递归式前序遍历："<<endl;
189.     root->preGo2();
190.     cout<<"递归式中序遍历："<<endl;
191.     root->miGo2();
192.     cout<<"递归式后序遍历："<<endl;
193.     root->postGo2();
194. }
```

数据结构

Processon：https://www.processon.com/view/link/5caf6129e4b0773d8c0be48b

数据结构 1
冒泡（基于交换） 1
快排（基于交换） 2
简单选择 4
归并 5
桶排序 8

冒泡（基于交换）

```
1.  #include <iostream>
2.  using namespace std;
3.  template<typename T>
4.  //整数或浮点数皆可使用
5.  void bubble_sort(T arr[], int len)
6.  {
7.      int i, j;  T temp;
8.      for (i = 0; i < len - 1; i++)
9.          for (j = 0; j < len - 1 - i; j++)
10.         if (arr[j] > arr[j + 1])
11.         {
12.             temp = arr[j];
13.             arr[j] = arr[j + 1];
14.             arr[j + 1] = temp;
15.         }
16. }
17. int main()
18. {
19.     int arr[] = { 61, 17, 29, 22, 34, 60, 72, 21, 50, 1, 62 };
20.     int len = (int) sizeof(arr) / sizeof(*arr);
21.     bubble_sort(arr, len);
22.     for (int i = 0; i < len; i++)
23.         cout << arr[i] << ' ';
24.
25.     cout << endl;
26.
27.     float arrf[] = { 17.5, 19.1, 0.6, 1.9, 10.5, 12.4, 3.8, 19.7, 1.5, 25.4, 28.6, 4.4, 23.8, 5.4 };
28.     len = (int) sizeof(arrf) / sizeof(*arrf);
29.     bubble_sort(arrf, len);
```

```
    for (int i = 0; i < len; i++)
        cout << arrf[i] << ' ';

    return 0;
}
```

快排（基于交换）

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

void Qsort(int arr[], int low, int high){
    if (high <= low) return;
    int i = low;
    int j = high + 1;
    int key = arr[low];
    while (true)
    {
        /*从左向右找比 key 大的值*/
        while (arr[++i] < key)
        {
            if (i == high){
                break;
            }
        }
        /*从右向左找比 key 小的值*/
        while (arr[--j] > key)
        {
            if (j == low){
                break;
            }
        }
        if (i >= j) break;
        /*交换 i,j 对应的值*/
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
    /*中枢值与 j 对应值交换*/
    int temp = arr[low];
    arr[low] = arr[j];
    arr[j] = temp;
    Qsort(arr, low, j - 1);
```

```
    Qsort(arr, j + 1, high);
}

int main()
{
    int a[] = {57, 68, 59, 52, 72, 28, 96, 33, 24};

    Qsort(a, 0, sizeof(a) / sizeof(a[0]) - 1);/*这里原文第三个参数要减 1 否则内存越界*/

    for(int i = 0; i < sizeof(a) / sizeof(a[0]); i++)
    {
        cout << a[i] << "";
    }

    return 0;
}/*参考数据结构 p274(清华大学出版社，严蔚敏)*/
```

简单选择

```
#include<iostream>
#include<time.h>
#include<iomanip>
using namespace std;
const int N=10;
int main()
{
   int a[N],i,j,temp,b;
   srand(time(NULL));
   for(i=0;i<N;i++)
      a[i]=rand()%100;
   for(i=0;i<N;i++)
      cout<<setw(3)<<a[i];
   cout<<endl;
   for(i=0;i<N-1;i++)
   {
      temp=i;
      for(j=i+1;j<N;j++)
      {
         if(a[temp]>a[j])
            temp=j;
      }
      if(i!=temp)
      {
         b=a[temp];
         a[temp]=a[i];
         a[i]=b;
      }
   }
   for(i=0;i<N;i++)
      cout<<setw(3)<<a[i];
   cout<<endl;
}
```

归并

//非递归

```
#include<iostream>
#include<ctime>
#include<cstring>
#include<cstdlib>
using namespace std;
/**将 a 开头的长为 length 的数组和 b 开头长为 right 的数组合并 n 为数组长度，用于最后一组
*/
void Merge(int* data,int a,int b,int length,int n){
  int right;
  if(b+length-1 >= n-1)  right = n-b;
  else  right = length;
  int* temp = new  int[length+right];
  int i=0, j=0;
  while(i<=length-1 && j<=right-1){
    if(data[a+i] <= data[b+j]){
      temp[i+j] = data[a+i];i++;
   }
    else{
    temp[i+j] = data[b+j];
    j++;
   }
  }
  if(j == right){//a 中还有元素，且全都比 b 中的大,a[i]还未使用
   memcpy(temp + i + j, data + a + i, (length - i) * sizeof(int));
  }
 else if(i == length){
   memcpy(temp + i + j, data + b + j, (right - j)*sizeof(int));
 }
  memcpy(data+a, temp, (right + length) * sizeof(int));
  delete [] temp;
}
void MergeSort(int* data, int n){
  int step = 1;
  while(step < n){
    for(int i=0; i<=n-step-1; i+=2*step)
      Merge(data, i, i+step, step, n);
  //将 i 和 i+step 这两个有序序列进行合并
  //序列长度为 step
  //当 i 以后的长度小于或者等于 step 时，退出
    step*=2;//在按某一步长归并序列之后，步长加倍
```

```
    }
}
int    main(){
    int    n;
    cin>>n;
    int*    data = new    int[n];
    if(!data)    exit(1);
    int    k = n;
    while(k--){
        cin>>data[n-k-1];
    }
    clock_t    s = clock();
    MergeSort(data, n);
    clock_t    e = clock();
    k=n;
    while(k--){
        cout<<data[n-k-1]<<' ';
    }
    cout<<endl;
    cout<<"the algorithm used"<<e-s<<"miliseconds."<<endl;
    delete    data;
    return    0;
}
```

//递归

```
#include<iostream>
using namespace std;
void merge(int *data, int start, int mid, int end, int *result)
{
    int i, j, k;
    i = start;
    j = mid + 1;                //避免重复比较 data[mid]
    k = 0;
    while (i <= mid && j <= end)      //数组 data[start,mid]与数组(mid,end]均没有全部归入数组 result 中去
    {
        if (data[i] <= data[j])       //如果 data[i]小于等于 data[j]
            result[k++] = data[i++];   //则将 data[i]的值赋给 result[k]，之后 i,k 各加一，表示后移一位
        else
            result[k++] = data[j++];   //否则，将 data[j]的值赋给 result[k]，j,k 各加一
    }
    while (i <= mid)              //表示数组 data(mid,end]已经全部归入 result 数组中去了，而数组 data[start,mid]还有剩余
        result[k++] = data[i++];      //将数组 data[start,mid]剩下的值，逐一归入数组 result
```

```
    while (j <= end)                //表示数组 data[start,mid]已经全部归入到 result 数组中去了，而数组(mid,high]还有剩余
        result[k++] = data[j++];      //将数组 a[mid,high]剩下的值，逐一归入数组 result

    for (i = 0; i < k; i++)         //将归并后的数组的值逐一赋给数组 data[start,end]
        data[start + i] = result[i];   //注意，应从 data[start+i]开始赋值
}
void merge_sort(int *data, int start, int end, int *result)
{
    if (start < end)
    {
        int mid = start + (end-start) / 2;//避免溢出 int
        merge_sort(data, start, mid, result);                //对左边进行排序
        merge_sort(data, mid + 1, end, result);              //对右边进行排序
        merge(data, start, mid, end, result);                //把排序好的数据合并
    }
}
void amalgamation(int *data1, int *data2, int *result)
{
    for (int i = 0; i < 10; i++)
        result[i] = data1[i];
    for (int i = 0; i < 10; i++)
        result[i + 10] = data2[i];
}
int main()
{
    int data1[10] = { 1,7,6,4,9,14,19,100,55,10 };
    int data2[10] = { 2,6,8,99,45,63,102,556,10,41 };
    int *result = new int[20];
    int *result1 = new int[20];
    amalgamation(data1, data2, result);
    for (int i = 0; i < 20; ++i)
        cout << result[i] << "  ";
    cout << endl;
    merge_sort(result, 0, 19, result1);
    for (int i = 0; i < 20; ++i)
        cout << result[i] << "  ";
    delete[]result;
    delete[]result1;
    return 0;
}
```

桶排序

桶排序 (Bucket sort)或所谓的**箱排序**，是一个排序算法，工作的原理是将数组分到有限数量的桶子里。每个桶子再个别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排序）。桶排序是鸽巢排序的一种归纳结果。当要被排序的数组内的数值是均匀分配的时候，桶排序使用线性时间（Θ（*n*））。但桶排序并不是 比较排序，他不受到 O(n log n) 下限的影响。

中文名	桶排序	数据结构设计	链表可以采用很多种方式实现
要　求	数据的长度必须完全一样	性　质	平均情况下桶排序以线性时间运行
公　式	Data=rand()/10000+10000	原　理	桶排序利用函数的映射关系
		领　域	计算机算法

海量数据

一年的全国高考考生人数为500 万，分数使用标准分，最低100 ，最高900 ，没有小数，要求对这500 万元素的数组进行排序。

分析：对500W数据排序，如果基于比较的先进排序，平均比较次数为O(5000000*log5000000)≈1.112亿。但是我们发现，这些数据都有特殊的条件： 100=<score<=900。那么我们就可以考虑桶排序这样一个“投机取巧”的办法、让其在毫秒级别就完成500万排序。

方法：创建801(900-100)个桶。将每个考生的分数丢进f(score)=score-100的桶中。这个过程从头到尾遍历一遍数据只需要500W次。然后根据桶号大小依次将桶中数值输出，即可以得到一个有序的序列。而且可以很容易的得到100分有***人，501分有***人。

实际上，桶排序对数据的条件有特殊要求，如果上面的分数不是从100-900，而是从0-2亿，那么分配2亿个桶显然是不可能的。所以桶排序有其局限性，适合元素值集合并不大的情况。

典型

在一个文件中有10G个整数，乱序排列，要求找出中位数。内存限制为2G。只写出思路即可（内存限制为2G意思是可以使用2G空间来运行程序，而不考虑本机上其他软件内存占用情况。） 关于中位数：数据排序后，位置在最中间的数值。即将数据分成两部分，一部分大于该数值，一部分小于该数值。中位数的位置：当样本数为奇数时，中位数=(N+1)/2；当样本数为偶数时，中位数为N/2与1+N/2的均值（那么10G个数的中位数，就第5G大的数与第5G+1大的数的均值了）。

分析：既然要找中位数，很简单就是排序的想法。那么基于字节的桶排序是一个可行的方法。

递归版本/DFS

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/
class Solution {
public:
    int TreeDepth(TreeNode* pRoot)
    {
        if(pRoot==NULL)
            return 0;
        int left=TreeDepth(pRoot->left);
        int right=TreeDepth(pRoot->right);
        return (left>right?left:right)+1;
    }
};
```

非递归版本/BFS:

```
class Solution {
public:
    int TreeDepth(TreeNode* pRoot)
    {
        queue<TreeNode *>q;
        if(pRoot==NULL)
            return 0;
        q.push(pRoot);
        int depth=0;
        while(!q.empty())
        {
            int len=q.size();
            depth++;
            while(len--)
            {
                TreeNode * temp=q.front();
                q.pop();
                if(temp->left) q.push(temp->left);
                if(temp->right) q.push(temp->right);
            }
```

```
        }
        return depth;
    }
};
```

二叉搜索树：

【 思路：

1.二叉搜索树具有一个很好的特点。以当前结点为根节点的左边结点的值都是小于根节点的值，右边结点的值都大于根节点的值。

2.根据这个特点，如果给的两个节点的值都小于根节点，那么它们的最低公共祖先就一定在它左子树。

3.如果给的两个节点的值都大于根节点，那么它们的最低公共祖先就一定在它右子树。

4.如果一个结点的值大于根节点的值，一个结点的值小于根节点的值，那么这个根节点就是它的最低公共祖先。 】

代码:

```
1.  //求两个节点的最低公共祖先(递归解法)
2.  BSTreeNode* GetCommenParent_R(BSTreeNode* root, BSTreeNode* bstn1, BSTreeNod
    e* bstn2)
3.  {
4.      if (root == NULL || bstn1 == NULL || bstn2 == NULL)
5.          return NULL;
6.
7.      if ((bstn1->_data < root->_data) && (bstn2->_data< root->_data))
8.      {
9.          GetCommenParent_R(root->_left, bstn1, bstn2);
10.     }
11.     else if ((bstn1->_data>root->_data) && (bstn2->_data>root->_data))
12.     {
13.         GetCommenParent_R(root->_right, bstn1, bstn2);
14.     }
15.     else
16.         return root;
17. }
```

普通二叉树：

【思路：

如果一个结点为根，另一个结点无论在什么地方它们的最低公共祖先一定为根结点。

如果一个结点在左树，另一个结点在右树，那么它的最低公共祖先一定是根节点。

如果两个结点都在左树，以子问题在左树查找。

如果两个结点都在右树，以子问题在右树查找。】

代码：

```
1.  //找一个结点
2.  struct TreeNode* GetNode(struct TreeNode* root, struct TreeNode* cur){
3.      if(root==NULL){
4.          return NULL;
5.      }
6.      if(root->val==cur->val){
7.          return root;
```

```
    }

    struct TreeNode* ret=GetNode(root->left,cur);
    if(ret){
        return ret;
    }
    return GetNode(root->right,cur);
}
struct TreeNode* lowestCommonAncestor(struct TreeNode* root, struct TreeNode* p, struct TreeNode* q) {
    if(root->val==p->val||root->val==q->val){
        return root;
    }
    bool pleft,pright,qleft,qright;

    if(GetNode(root->left,p)){
        pleft=true;
        pright=false;
    }
    else{
        pleft=false;
        pright=true;
    }
    if(GetNode(root->left,q)){
        qleft=true;
        qright=false;
    }
    else{
        qleft=false;
        qright=true;
    }
    //一个在左，一个在右，返回根
    if((pleft&&qright)||(pright&&qleft)){
        return root;
    }
    //两个都在左
    if(pleft&&qleft){
        return lowestCommonAncestor(root->left,p,q);
    }
    //两个都在右
    if(pright&&qright){
        return lowestCommonAncestor(root->right,p,q);
    }

```

```
51.     return NULL;
52. }
```

```cpp
#include<iostream>
#include<cstring>
using namespace std;
void dfs_pre(char*in , char*pos , int len){
    if(len<=0)return;
    int p = strchr(in,pos[len-1])-in;
    cout<<pos[len-1];
    dfs_pre(in,pos,p);
    dfs_pre(in+p+1,pos+p,len-p-1);
}
void dfs_pos(char *in, char*pre , int len ){
    if(len<=0)return;
    int p = strchr(in,pre[0])-in;
    dfs_pos(in,pre+1,p);
    dfs_pos(in+p+1,pre+p+1,len-p-1);
    cout<<pre[0];
}
int main(){
    char pre[]="ABDGHCEFI",in[]="GDHBAECIF",pos[]="GHDBEIFCA";
    dfs_pos(in,pre,strlen(in));
    cout<<endl;
    dfs_pre(in,pos,strlen(in));
    cout<<endl;
    return 0;
}
```

```cpp
class Solution {
public:
    int Fibonacci(int n) {
        int a=1,b=1,res=0;
        if(n==0){
            return 0;
        }else if(n<3){
            return 1;
        }
        for(int i=3;i<=n;i++){
           res=a+b;
           a=b;
           b=res;
        }
        return res;
    }
};
//思路1：迭代:for(){}
//思路2：递归:f(n+2)=f(n+1)+f(n)
```

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) :
            val(x), next(NULL) {
    }
};*/
class Solution {
public:
    ListNode* ReverseList(ListNode* pHead)
    {
        // 反转指针
        ListNode* pNode=pHead;  // 当前节点
        ListNode* pPrev=nullptr;// 当前节点的上一个节点
        ListNode* pNext=nullptr;// 当前节点的下一个节点
        ListNode* pReverseHead=nullptr;//新链表的头指针

        // 反转链表
        while(pNode!=nullptr)
        {
            pNext=pNode->next; // 建立链接

            if(pNext==NULL)    // 判断 pNode 是否是最后一个节点
                pReverseHead=pNode;

            pNode->next=pPrev; // 指针反转
            pPrev=pNode;
            pNode=pNext;
        }
        return pReverseHead;
    }
};
```

- **全局排序**，O(n*lg(n))
- **局部排序**，只排序 TopK 个数，O(n*k)
- **堆**，TopK 个数也不排序了，O(n*lg(k))
- ~~分治法，每个分支“都要”递归，例如：快速排序，O(n*lg(n))~~
- ~~减治法，“只要”递归一个分支，例如：三分查找 O(lg(n))，随机选择 O(n)~~
- **随机选择**+partition
- **比特位图计数**

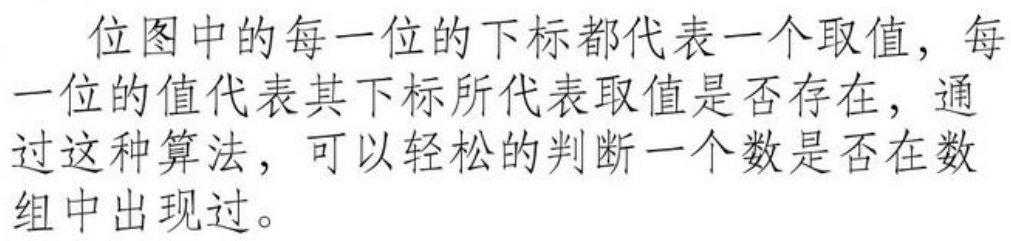
随机选择并排序/基于快速排序：

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

int findK(int nums[],int k,int start,int end) {
    int low=start;
    int high=end;
    int temp=nums[low];//枢纽点
    while(low<high) {
        while(low<high&&nums[high]<=temp) {
            high--;
        }
        nums[low]=nums[high];
        while(low<high&&nums[low]>=temp) {
            low++;
        }
        nums[high]=nums[low];
    }
    nums[high]=temp;

    if(high==k-1) {
        return temp;
    } else if(high>k-1) {
        return findK(nums,k,start,high-1);
    } else {
        return findK(nums,k,high+1,end);
    }

}

int findKthLargest(int nums[],int k,int len) {
    return findK(nums,k,0,len);
}
```

```
33.
34. int main() {
35.     int nums[]= {97,76,99,102,3,5,888};
36.     cout<<findKthLargest(nums,6,7)<<endl;
37.     return 0;
38. }
```

1. 先排序（快排、归并），再去重
   排序：

   去重：
   1. 由于数组已经完成排序，因此设定第一个指针 i，遍历数组，每遇到 nums[i] != nums[i - 1],就说明遇到了新的不同数字，记录之；
   2. 设定第二个指针 k，有两个用途：
      记录不同数字的数量；
      每遇到新的不同数字，作为修改数组元素的 index。
2. 位图算法/比特图



假设 array1 有足够的空间了，于是我们不需要额外构造一个数组，并且可以从后面不断地比较元素进行合并。

- 比较 array2 与 array1 中最后面的那个元素，把最大的插入第 m+n 位
- 改变数组的索引，再次进行上面的比较，把最大的元素插入到 array1 中的第 m+n-1 位。
- 循环一直到结束。**循环结束条件：当 index1 或 index2 有一个小于 0 时**，此时就可以结束循环了。如果 index2 小于 0，说明目的达到了。如果 index1 小于 0，就把 array2 中剩下的前面的元素都复制到 array1 中去就行。

功能代码：
输入一次 m>n 的情况
输入一次 m<n 的情况

特殊输入情况：

- 当 array1 为空，array2 不为空时，将 array2 的所有元素添加到 array1 中即可
- 当 array1 不为空，array2 为空时，就是上面的循环结束条件，直接返回 array1.
- 当 array1 跟 array2 都为空时，返回空。

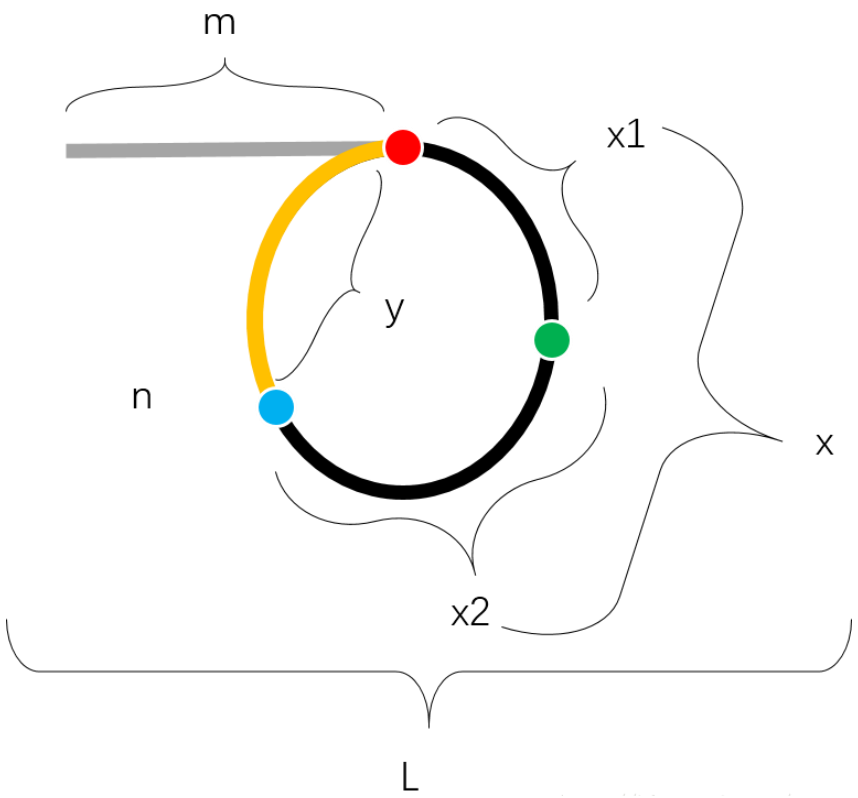
```
#include<iostream>
using namespace std;

int merge(int numa[],int m,int numb[],int n) {
    int index1=m-1;
    int index2=n-1;
    while(index2>=0){
        if(index1<0){
            for(int i=0;i<=index2;i++){
                    numa[i]=numb[i];
            }
            break;
        }
        if(numa[index1]>=numb[index2]){
            numa[index1+index2+1]=numa[index1];
            index1--;
        }else{
            numa[index1+index2+1]=numb[index2];
            index2--;
        }
    }
}
```

```
int main() {
    int numa[]={1,3,7,0,0,0};
    int numb[]={2,4,5};
    merge(numa,3,numb,3);
    for(int i=0;i<6;i++){
        cout<<numa[i]<<endl;
    }
    return 0;
}
```

差速法:

```
1.  ListNode* EntryNodeOfLoop(ListNode* pHead) {
2.      if(pHead == null|| pHead.next == null|| pHead.next.next == null)
3.          return null;
4.      ListNode* fast = pHead->next->next;
5.      ListNode* slow = pHead->next;
6.      while(fast! = slow) {
7.          if(fast->next != null && fast->next->next != null) {
8.              fast = fast->next->next;
9.              slow = slow->next;
10.         } else {
11.             return null;
12.         }
13.     }
14.     fast = pHead;
15.     while(fast != slow) {
16.         fast = fast->next;
17.         slow = slow->next;
18.     }
19.     return slow;
20. }
```

证明:



上图是一个简单有环单链表的模型图，总长度为 L，无环部分长度（图中灰色部分）为 m，环的长度（图中黑色加橙色部分）为 n，红点为环的入口节点。

现在有两个指针，一个每次移动一个节点，称为慢指针，另一个的速度是前者的两倍，称为快指针。两个指针同时从链表头节点出发，它们必在环中相遇。图中绿色节点为慢指针刚到达环入口节点时快指针的位置，此时距离慢指针的距离（图中红绿两点间黑色部分）为 x1，蓝色点表示快慢指针在环中相遇的位置，此时距离环的入口节点距离（图中红蓝两点间黑色部分）为 x，沿着运动方向距离入口节点距离（图中橙色部分）还有 y。

上面是对图的说明，下面开始证明一个结论：

$$m = k * n + y. \text{ (k 为正整数)}$$

当快慢指针相遇时，假设慢指针走了 t 步，那么快指针一定走了 2t 步。

当慢指针进入以后，快指针一定会在慢指针走完一圈之前追上它。因为最坏的情况是慢指针进入环时快指针刚好落后它一整圈，这样慢指针走完一圈快指针刚好追上慢指针。

快慢指针相遇时，慢指针走过的距离是：

$$t = m + x$$

当慢指针刚进入环入口时，快指针走过的的距离是：

$$t1 = m + k * n + x1. \text{ (k 为正整数)}$$

当快慢指针相遇时，快指针又走过了：

$$t2 = n + x2$$

所以快慢指针相遇时，快指针一共走过了：

$$t1 + t2 = m + k * n + x1 + n + x2 = m + (k + 1) * n + x = m + k * n + x. \text{ (k 为正整数)}$$

而相遇时快指针走过的距离是慢指针的两倍。

$$2 * t = t1 + t2$$

$$2(m + x) = m + k * n + x$$

$$m = k * n - x$$

$$m = (k - 1) * n + (n - x)$$

$$m = k * n + y. \quad \text{(k 为正整数)}$$

这个结论说明，从头节点到环入口的距离等于快慢指针相遇处继续走到环入口（图中橙色部分）的距离加上环长度的整倍数。如果有一个人从链表头节点开始每次移动一个节点地往后走，另一个人从快慢指针相遇处（图中蓝色点）以同样的速度往前走，结果就是，两人相遇在环的入口节点处。

动态规划

题目描述

给定一个数组 array[1, 4, -5, 9, 8, 3, -6]，在这个数字中有多个子数组，子数组和最大的应该是：[9, 8, 3]，输出 20，再比如数组为[1, -2, 3, 10, -4, 7, 2, -5]，和最大的子数组为[3, 10, -4, 7, 2]，输出 18。

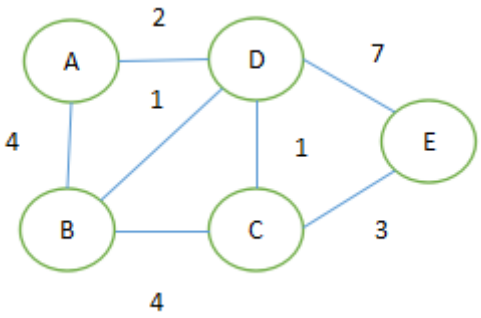
思路分析

1、状态方程 ： max(dp[i]) = getMax(max(dp[i -1]) + arr[i] ,arr[i])

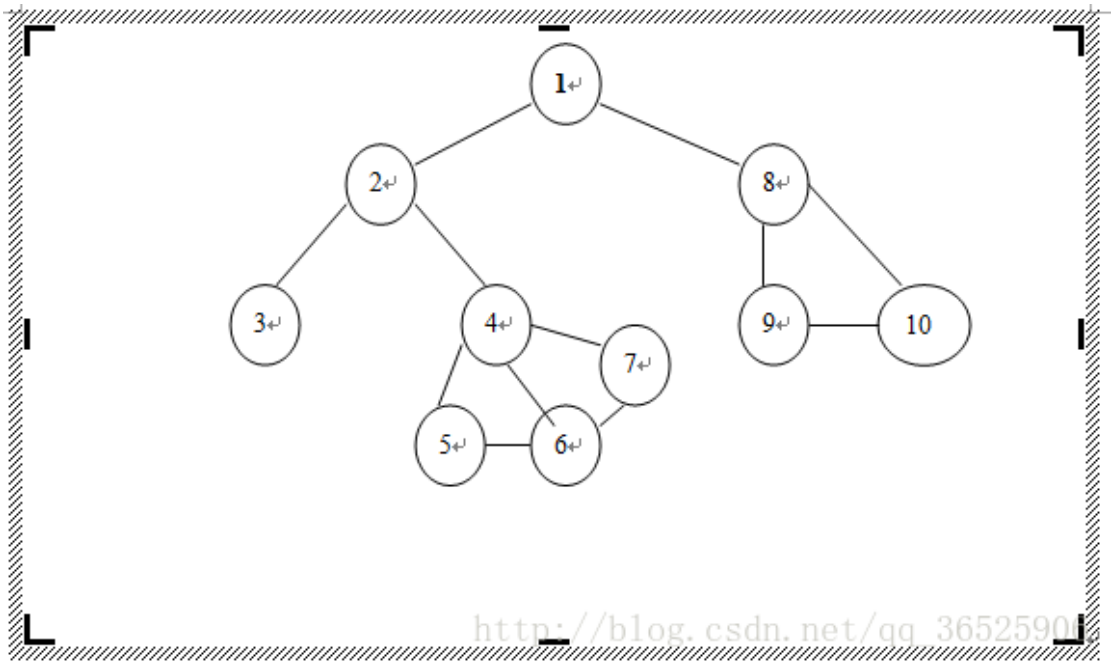
2、上面式子的意义是：我们从头开始遍历数组，遍历到数组元素 arr[i] 时，连续的最大的和 可能为 max(dp[i -1]) + arr[i] ， 也可能为 arr[i] ， 做比较即可得出哪个更大，取最大值。时间复杂度为 n。

代码实现

```
#include<iostream>
using namespace std;
int GetMax(int a, int b) { //得到两个数的最大值
    return a >b ? a : b;
}

int GetMaxAddOfArray(int* arr, int sz) {
    if (arr == NULL || sz <= 0)
        return 0;

    int Sum = arr[0];   //临时最大值
    int MAX = arr[0];   //比较之后的最大值

    for (int i = 1; i < sz; i++) {
        Sum = GetMax(Sum + arr[i], arr[i]);   //状态方程
        if (Sum >= MAX)
            MAX = Sum;
    }
    return MAX;
}

int main() {
    int array[] = { 2, 3, -6, 4, 6, 2, -2, 5, -9 };
    int sz = sizeof(array) / sizeof(array[0]);
    int MAX = GetMaxAddOfArray(array, sz);
    cout << MAX << endl;
    return 0;
}
```




```cpp
#include<iostream>
using namespace std;
int GetMax(int a, int b) { //得到两个数的最大值
    return a >b ? a : b;
}

int GetMaxAddOfArray(int* arr, int sz) {
    if (arr == NULL || sz <= 0)
        return 0;

    int Sum = arr[0];   //临时最大值
    int MAX = arr[0];   //比较之后的最大值

    for (int i = 1; i < sz; i++) {
        Sum = GetMax(Sum + arr[i], arr[i]);   //状态方程
        if (Sum >= MAX)
            MAX = Sum;
    }
    return MAX;
}

int main() {
    int array[] = { 2, 3, -6, 4, 6, 2, -2, 5, -9 };
    int sz = sizeof(array) / sizeof(array[0]);
    int MAX = GetMaxAddOfArray(array, sz);
    cout << MAX << endl;
    return 0;
}
```



DFS：

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

int a[11][11];
bool visited[11];

void store_graph()  //邻接矩阵存储图
{
    int i,j;

    for(i=1;i<=10;i++)
        for(j=1;j<=10;j++)
            cin>>a[i][j];
}

void dfs_graph()    //深度遍历图
{
    void dfs(int v);

    memset(visited,false,sizeof(visited));

    for(int i=1;i<=10;i++)  //遍历每个顶点是为了防止图不连通时无法访问每个顶点
        if(visited[i]==false)
            dfs(i);
}

```

```
void dfs(int v)  //深度遍历顶点
{
    int Adj(int x);

    cout<<v<<" ";  //访问顶点 v
    visited[v]=true;

    int adj=Adj(v);
    while(adj!=0)
    {
        if(visited[adj]==false)
            dfs(adj);       //递归调用是实现深度遍历的关键所在

        adj=Adj(v);
    }
}

int Adj(int x)   //求邻接点
{
    for(int i=1;i<=10;i++)
        if(a[x][i]==1 && visited[i]==false)
            return i;

    return 0;
}

int main()
{
    cout<<"初始化图:"<<endl;
    store_graph();

    cout<<"dfs 遍历结果:"<<endl;
    dfs_graph();

    return 0;
}
```

BFS：

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

int a[11][11];
bool visited[11];
```

```
7.
8.  void store_graph()
9.  {
10.     for(int i=1;i<=10;i++)
11.         for(int j=1;j<=10;j++)
12.             cin>>a[i][j];
13. }
14.
15. void bfs_graph()
16. {
17.     void bfs(int v);
18.
19.     memset(visited,false,sizeof(visited));
20.
21.     for(int i=1;i<=10;i++)
22.         if(visited[i]==false)
23.             bfs(i);
24. }
25.
26. void bfs(int v)
27. {
28.     int Adj(int x);
29.
30.     queue<int> myqueue;
31.     int adj,temp;
32.
33.     cout<<v<<" ";
34.     visited[v]=true;
35.     myqueue.push(v);
36.
37.     while(!myqueue.empty())    //队列非空表示还有顶点未遍历到
38.     {
39.         temp=myqueue.front();  //获得队列头元素
40.         myqueue.pop();         //头元素出对
41.
42.         adj=Adj(temp);
43.         while(adj!=0)
44.         {
45.             if(visited[adj]==false)
46.             {
47.                 cout<<adj<<" ";
48.                 visited[adj]=true;
49.                 myqueue.push(adj);   //进对
50.             }
```

```
51.
52.             adj=Adj(temp);
53.         }
54.     }
55. }
56.
57. int Adj(int x)
58. {
59.     for(int i=1;i<=10;i++)
60.         if(a[x][i]==1 && visited[i]==false)
61.             return i;
62.
63.     return 0;
64. }
65.
66. int main()
67. {
68.     cout<<"初始化图:"<<endl;
69.     store_graph();
70.
71.     cout<<"bfs 遍历结果:"<<endl;
72.     bfs_graph();
73.
74.     return 0;
75. }
```