Fibonacci sequence

Fibonacci sequence

$$F_1 = F_2 = 1,\\ F_N = F_{n-1} + F_{N-2}$$

递归方法

```
int fib(int n) {
    return (n < 3? 1: fib(n-1)+fib(n-2));
}
```

迭代方法

```
int fib(int n) {
    int f_2;
    int f_1 = 1;
    int f_0 = 1;
    for(int i = 1;i < n;i ++) {
        f_2 = f_1 + f_0;
        f_0 = f_1;
        f_1 = f_2;
    }
    return f_0;
}
```

利用矩阵快速幂方法,复杂度$O(log_2^n)$

$$\begin{bmatrix} f(n) & f(n-1) \\ f(n-1) & f(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

```
const long long mod = pow(10,9) + 7;

struct matrix {
    long long f_2, f_10, f_11, f_0;
    matrix() {
        f_2 = 1;
        f_10 = f_11 = 1;
        f_0 = 0;
    }
};

matrix operator *(matrix a, matrix b) {
    matrix c;
    c.f_2 = a.f_2*b.f_2 + a.f_10 *b.f_11;

    c.f_10 = a.f_2*b.f_10 + a.f_10 * b.f_0;
    c.f_11 = a.f_11*b.f_2 + a.f_0 * b.f_11;
    c.f_0 = a.f_11*b.f_10 + a.f_0 * b.f_0;
    c.f_2 %= mod;
    c.f_10 %= mod;
    c.f_11 %= mod;
    c.f_0 %= mod;
    return c;
}

long long f_pow(long long n) {
    if (n == -1) {
        return 0;
    }
    matrix ans,base;
    ans.f_11 = 0;
    ans.f_10 = 0;
    ans.f_0 = 1;
    while (n != 0) {
        if (n & 1 != 0) {
            ans  = ans * base;
        }
        base = base * base;
        n >>= 1;
    }
    return ans.f_2;
}
```

【快速幂一般模板】

```
int poww(int a, int b) {
    int ans = 1, base = a;
    while (b != 0) {
        if (b & 1 != 0)
            ans *= base;
        base *= base;
        b >>= 1;
    }
    return ans;
}
```

Sum of Powers

$$\begin{aligned}\sum_{k=1}^{n} k^2 &= \frac{1}{6}n(n+1)(2n+1)\\ \sum k^3 &= \left(\sum k\right)^2 = \left(\frac{1}{2}n(n+1)\right)^2\end{aligned}$$

- Pretty useful in many random situations
- Memorize above!

快速幂

- Recursive computation of a^n:

$$a^n = \begin{cases} 1 & n = 0 \\ a & n = 1 \\ (a^{n/2})^2 & n \text{ is even} \\ a(a^{(n-1)/2})^2 & n \text{ is odd} \end{cases}$$

```
double pow(double a, int n) {
     if(n == 0) return 1;
     if(n == 1) return a;
     double t = pow(a, n/2);
     return t * t * pow(a, n%2);
}
```

- Running time: $O(\log n)$

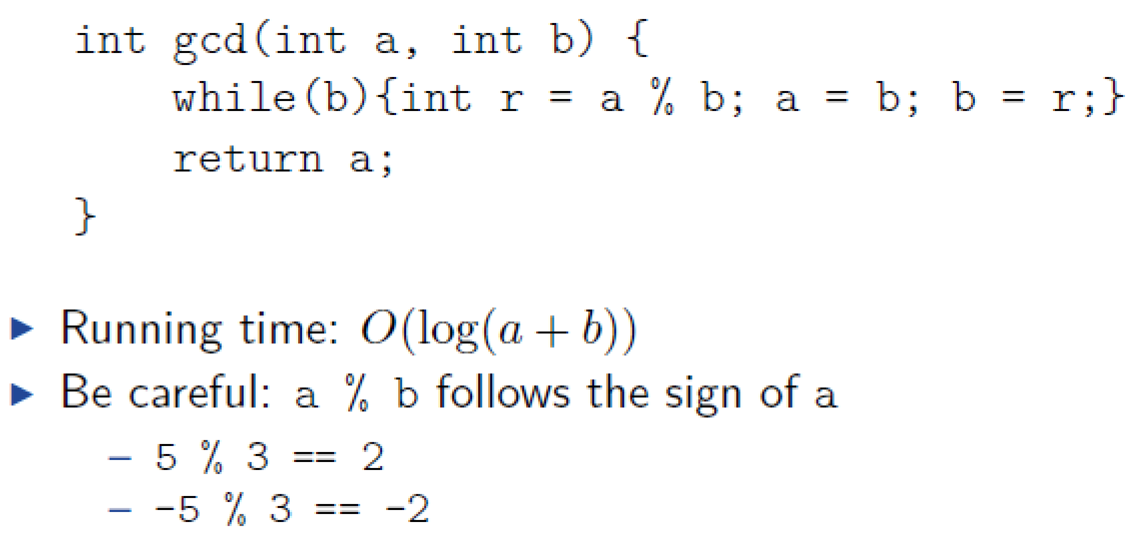
```
double pow(double a, int n) {
    double ret = 1;
    while(n) {
        if(n%2 == 1) ret *= a;
        a *= a; n /= 2;
    }
    return ret;
}
```

GCD

- $\gcd(a, b)$: greatest integer divides both a and b
- Used very frequently in number theoretical problems
- Some facts:
  - $\gcd(a, b) = \gcd(a, b - a)$
  - $\gcd(a, 0) = a$
  - $\gcd(a, b)$ is the smallest positive number in $\{ax + by \mid x, y \in \mathbf{Z}\}$

- Repeated use of $\gcd(a, b) = \gcd(a, b - a)$
- Example:

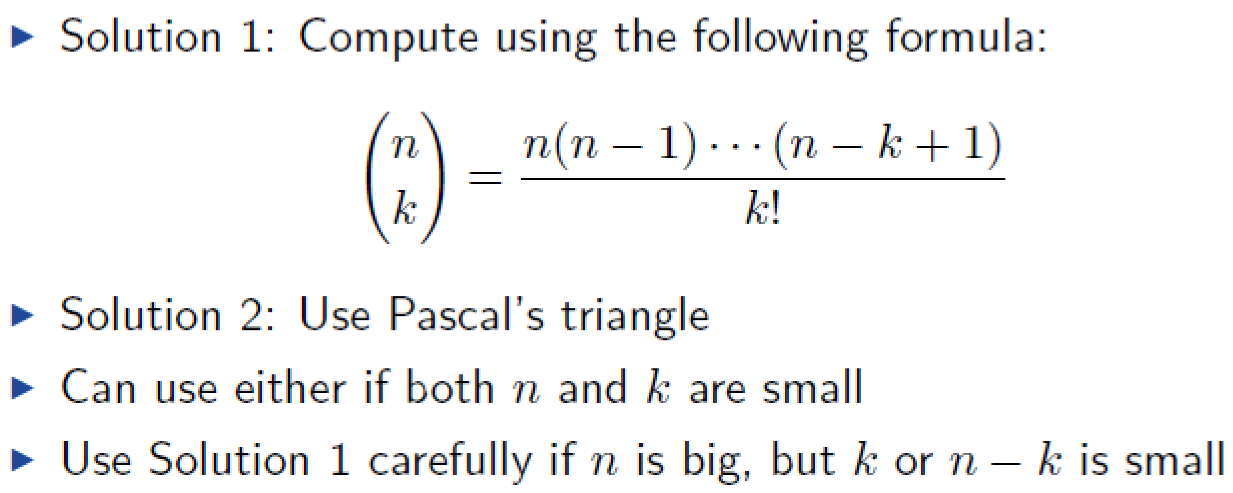
$$\begin{aligned}
\gcd(1989, 867) &= \gcd(1989 - 2 \times 867, 867) \\
&= \gcd(255, 867) \\
&= \gcd(255, 867 - 3 \times 255) \\
&= \gcd(255, 102) \\
&= \gcd(255 - 2 \times 102, 102) \\
&= \gcd(51, 102) \\
&= \gcd(51, 102 - 2 \times 51) \\
&= \gcd(51, 0) \\
&= 51
\end{aligned}$$



```
int gcd(int a, int b) {
    while(b){int r = a % b; a = b; b = r;}
    return a;
}
```

- Running time: $O(\log(a+b))$
- Be careful: `a % b` follows the sign of `a`
  - `5 % 3 == 2`
  - `-5 % 3 == -2`

Binomial Coefficients



- Solution 1: Compute using the following formula:

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k!}$$

- Solution 2: Use Pascal's triangle
- Can use either if both n and k are small
- Use Solution 1 carefully if n is big, but k or $n-k$ is small

In Pascal's triangle, each number is the sum of the two numbers directly above it.

排序 SORT

插入排序

```
void insert_sort(int * arr, int len) {
    if (len > 1) {
        int first_unsorted = 1;
        while (first_unsorted < len) {
            int target = -1;
            for (int i = 0; i < first_unsorted; i++) {
                if (arr[i] > arr[first_unsorted]) {
                    target = i;
                    break;
                }
            }
            if (target != -1) {
                int element = arr[first_unsorted];
                for (int i = first_unsorted; i > target; i--) {
                    arr[i] = arr[i - 1];
                }
                arr[target] = element;
            }
            first_unsorted++;
        }
    }
}
```

归并排序

```
void merge(int * arr, int from, int mid, int end) {
    int len1 = mid - from + 1;
    int len2 = end - (mid+1) + 1;
    int * arr1 = new int[len1];
    int * arr2 = new int[len2];
    for (int i = 0; i < len1; i++) {
        arr1[i] = arr[from + i];
    }
    for (int i = 0; i < len2; i++) {
        arr2[i] = arr[mid + 1 + i];
    }
    int p = 0; int q = 0;
    int i = from;
    for (; i <= end && p < len1 && q < len2; i++) {
        arr[i] = arr1[p] < arr2[q] ? arr1[p++] : arr2[q++];
    }
    while (p < len1) {
        arr[i++] = arr1[p++];
    }
    while (q < len2) {
        arr[i++] = arr2[q++];
    }
}

void merge_sort(int * arr, int from, int end) {
    if (from < end) {
        int mid = (from + end) / 2;
        merge_sort(arr, from, mid);
        merge_sort(arr, mid + 1, end);
        merge(arr, from, mid, end);
    }
}
```

贪心算法

```
// 优先队列
// less是从大到小，greater是从小到大
// 可重载< > 进行更多判别方式的设定
// i.e. priority_queue<node,vector<node>,less<node> >q;
// 重载node的< 和 > ,bool operator < (const node & a,const node & b);
// 用greater的时候要添加头文件 #include <functional>;
priority_queue<int,vector<int>,less<int> >q;
priority_queue<int,vector<int>,greater<int> >q;

// 当元素为指针的时候，最好用这种方式priority_queue<node *,vector<node*>,PCmp >q;
// > 表示小的优先，< 表示大的优先
struct PCmp
{
    bool operator () (node const *x, node const *y)
    {
        return x->f > y->f;
    }
};
```

Huffman code

优先队列

有时候可能是char类型的锅orz

```
#include <iostream>
#include <string>
#include <vector>
#include <functional>
#include <queue>
#include <stack>
using namespace std;

//less是从大到小，greater是从小到大
//priority_queue<int,vector<int>,less<int> >q;
//priority_queue<int,vector<int>,greater<int> >q;


struct node
{
    int c;
    int f;
    int index;
    node * father;
    node(int c, int f) {
        this->c = c;

        this->f = f;
        father = NULL;
    }
};
bool operator < (const node & a,const node & b) {
    return a.f < b.f;
}

bool operator > (const node & a, const node & b) {
    return a.f > b.f;
}

struct PCmp
{
    bool operator () (node const *x, node const *y)
    {
        return x->f > y->f;
    }
};


int main() {
    int n;
    cin >> n;
    priority_queue<node *, vector<node*>, PCmp > q;
    queue<node *> q_record;
    for (int i = 0; i < n; i++) {
        int c;
        int f;
        cin >> c >> f;
        node * temp = new node(c, f);
        q.push(temp);
        q_record.push(temp);
    }
    while (q.size()>1) {
        node * a = q.top(); q.pop();
        node * b = q.top(); q.pop();
        node * ab = new node(-1, a->f + b->f);
        a->father = ab;
        a->index = 0;
        b->father = ab;
        b->index = 1;
        q.push(ab);
    }

    while (!q_record.empty()) {
        node * temp = q_record.front();
        cout << temp->c << ": ";
        q_record.pop();
        stack<int> s;
        while (temp->father) {
            s.push(temp->index);
            temp = temp->father;

        }
        while (!s.empty()) {
            cout << s.top();
            s.pop();
        }
        cout << endl;
    }
    system("pause");
}
```

Single-link Clustering

本质是最小生成树问题，用kruskal或者prim算法都可以，而kruskal算法又可归为并查集算法

所以并查集算法总结

快速查找（慢速并集）

$O(mn)$

```
#include<stdio.h>
#define N 1000
//quick-find
int main() {
    int i,p,q,t,id[N];
    for(i = 0;i < N;i ++) {
        id[i] = i; //初始化id数组，此时每个元素均不相通
    }
    while(scanf("%d %d",&p,&q) == 2) { //输入pq数对
        if(id[p] == id[q]) { //如果p-q的id相同，那么p-q连通，continue到下一次输入循环
            continue;
        }
        t = id[p];
        for(i = 0;i < N;i ++) { //在id数组中查找与id[p]相同的数字
            if(id[i] == t) { //如果相等，赋值为id[q]，这样id[p]与id[q]及所有与id[p]id[q]有联系的数组
都有相同的id了，即归并两个数组
                id[i] = id[q];
            }
        }
        printf("%d %d\n",p,q);
    }
}
```

quick-union 快速并集算法

图中表述的连通分量叫做树，每棵树只有一个指向它本身的对象，即树根，其余的对象均指向或者间接指向树根。比如一棵树的根是9，那么id[9] == 9，如果3->4，4->9，即id[3] == 4，id[4] == 9，3和4均不指向它本身，最终的指向都为9,3和4都在以9为根节点的书里面。

这样我们只需要找到树的一个节点，就可以根据这个节点找到树根，从而判断是否具有连通性了。

性质：若M>N，快速并集算法可能要运行多于M*N/2条指令来解决一个拥有N个对象、M个对的连通性问题。

```
#include<stdio.h>
#define N 1000
//quick-union
int main() {
    int i,j,p,q,id[N];
    for(i = 0;i < N;i ++) {
        id[i] = i;
    }
    /*
    p   q   0 1 2 3 4 5 6 7 8 9

    3   4   0 1 2 4 4 5 6 7 8 9
    4   9   0 1 2 4 9 5 6 7 8 9
    8   0   0 1 2 4 9 5 6 7 0 9
    2   3   0 1 9 4 9 5 6 7 0 9
    5   6   0 1 9 4 9 6 6 7 0 9
    2   9   0 1 9 4 9 6 6 7 0 9
    5   9   0 1 9 4 9 6 6 7 0 9

    */
    while(scanf("%d %d",&p,&q) == 2) {
        for(i = p;i != id[i];i = id[i]); //从节点p开始找，一直找到对应的树根并赋给i
        for(j = q;j != id[j];j = id[j]); //从节点q开始找，一直知道对应的树根并赋给j
        if(i == j) { //p和q的树根相同，进行下一次输入循环
            continue;
        }
        id[i] = j; //p和q的树根不相同，则把q的树根赋给p的树根，那么这两棵数就建立了联系变成了一棵树
        printf("%d %d\n",p,q);
    }
}
```

weighted quick-union 加权快速并集算法

加权快速算法是快速并集算法的修改版：不是任意连通第二棵树和第一棵树，而是比较两棵树节点数多少，把小的树的根节点连到大的树的根节点上。这样每个节点与树根的距离短，查找的运算效率也就高了。

性质：加权快速并集算法判断N个对象的其中两个是否连通，最多要跟踪$2 * lgN$个指针。

```
#include<stdio.h>
#define N 1000

//weighted quick-union
int main() {
    int i,j,p,q,id[N],size[N]; //定义一个数组size来记录树的大小
    for(i = 0;i < N;i ++) {
        id[i] = i;
        size[i] = 1; //初始所有的树的size为1
    }
    while(scanf("%d %d",&p,&q) == 2) {

        for(i = p;id[i] != i;i = id[i]);
        for(j = q;id[j] != j;j = id[j]);
        if(i == j) {
            continue;
        }
        if(size[i] < size[j]) { //如果i树的大小小于j树，那么把小的i树的根节点设为j，i树合并到j树，并且j树的size增大为size[i]+size[j]
            id[i] = j;size[j] += size[i];
        }
        else {
            id[j] = i;size[i] += size[j];
        }
        printf("%d %d\n",p,q);
    }
```

weighted quick-union with path compression by halving 对分路径压缩加权快速并集算法

对分路径压缩是通过让每一个链接在通向树上部的路径跳到下一个节点来实现压缩。

```
#include<stdio.h>
#define N 1000

//weighted quick-union with path compression by halving
int main() {
    int i,j,p,q,id[N],size[N];
    for(i = 0;i < N;i ++) {
        id[i] = i;
        size[i] = 1;
    }
    while(scanf("%d %d",&p,&q) == 2) {
        for(i = p;id[i] != i;i = id[i]) {
            id[i] = id[id[i]]; //找到上一个节点的路径，对分路径长度
        }
        for(j = q;id[j] != j;j = id[j]) {
            id[j] = id[id[j]];
        }
        if(i == j) {
            continue;
        }
        if(size[i] < size[j]) {
            id[i] = j;size[j] += size[i];
        }
        else {
            id[j] = i;size[i] += size[j];
        }
        printf("%d %d\n",p,q);
    }
}
```

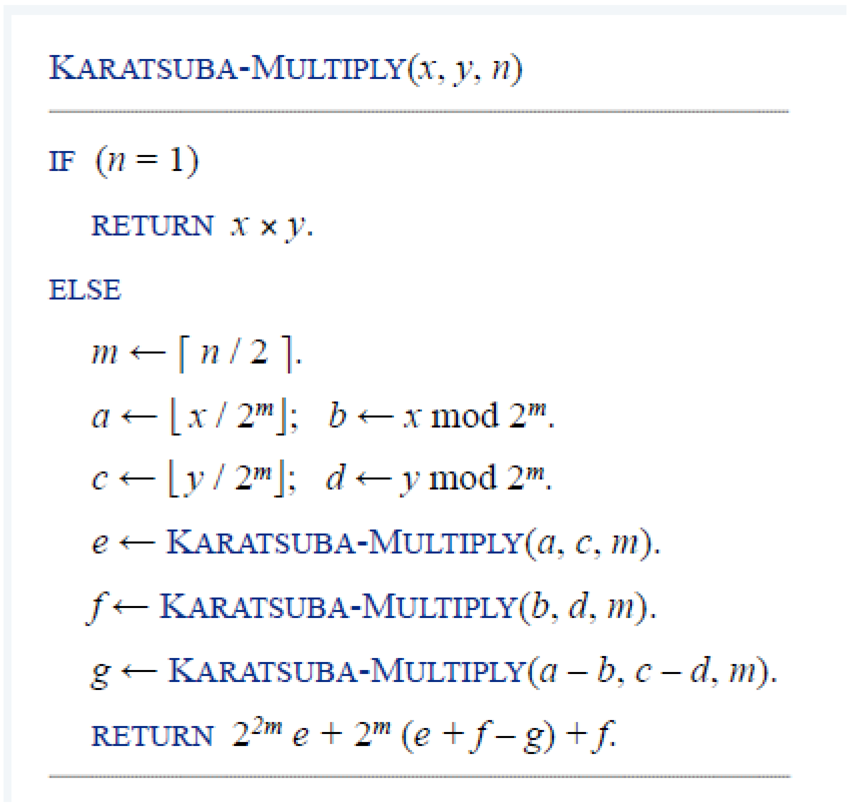
保留两位精度

```
#include <iomanip>
    double ans;
    cout << fixed << setprecision(2) << ans;
// 或者
    printf("%.2f",ans);
```

大数求模

```
//str是大数，求str%k
int ans = 0;
for (int i = 0; i < str.length(); i++) {
    ans = ((ans * 10)% k  + (str[i]-'0')% k ) % k;
}
```

Karatsuba multiply



```
KARATSUBA-MULTIPLY($x, y, n$)

IF ($n = 1$)
    RETURN $x \times y$.
ELSE
    $m \leftarrow \lceil n / 2 \rceil$.
    $a \leftarrow \lfloor x / 2^m \rfloor$;  $b \leftarrow x \bmod 2^m$.
    $c \leftarrow \lfloor y / 2^m \rfloor$;  $d \leftarrow y \bmod 2^m$.
    $e \leftarrow$ KARATSUBA-MULTIPLY($a, c, m$).
    $f \leftarrow$ KARATSUBA-MULTIPLY($b, d, m$).
    $g \leftarrow$ KARATSUBA-MULTIPLY($a - b, c - d, m$).
    RETURN $2^{2m} e + 2^m (e + f - g) + f$.
```

注意这里算的是二进制字符串相乘

```
// FOLLOWING TWO FUNCTIONS ARE COPIED FROM http://goo.gl/q0OhZ
// Helper method: given two unequal sized bit strings, converts them to
// same length by adding leading 0s in the smaller string. Returns the
// the new length

int makeEqualLength(string &str1, string &str2)
```

```
{
    int len1 = str1.size();
    int len2 = str2.size();
    if (len1 < len2)
    {
        for (int i = 0 ; i < len2 - len1 ; i++)
            str1 = '0' + str1;
        return len2;
    }
    else if (len1 > len2)
    {
        for (int i = 0 ; i < len1 - len2 ; i++)
            str2 = '0' + str2;
    }
    return len1; // If len1 >= len2
}

// The main function that adds two bit sequences and returns the addition
string addBitStrings( string first, string second )
{
    string result;  // To store the sum bits

    // make the lengths same before adding
    int length = makeEqualLength(first, second);
    int carry = 0;  // Initialize carry

    // Add all bits one by one
    for (int i = length-1 ; i >= 0 ; i--)
    {
        int firstBit = first.at(i) - '0';
        int secondBit = second.at(i) - '0';

        // boolean expression for sum of 3 bits
        int sum = (firstBit ^ secondBit ^ carry)+'0';

        result = (char)sum + result;

        // boolean expression for 3-bit addition
        carry = (firstBit&secondBit) | (secondBit&carry) | (firstBit&carry);
    }

    // if overflow, then add a leading 1
    if (carry)  result = '1' + result;

    return result;
}

// A utility function to multiply single bits of strings a and b
int multiplyiSingleBit(string a, string b)
{  return (a[0] - '0')*(b[0] - '0');  }

// The main function that multiplies two bit strings X and Y and returns

// result as long integer
```

```
long int multiply(string X, string Y)
{
    // Find the maximum of lengths of x and Y and make length
    // of smaller string same as that of larger string
    int n = makeEqualLength(X, Y);

    // Base cases
    if (n == 0) return 0;
    if (n == 1) return multiplyiSingleBit(X, Y);

    int fh = n/2;   // First half of string, floor(n/2)
    int sh = (n-fh); // Second half of string, ceil(n/2)

    // Find the first half and second half of first string.
    // Refer http://goo.gl/lLmgn for substr method
    string Xl = X.substr(0, fh);
    string Xr = X.substr(fh, sh);

    // Find the first half and second half of second string
    string Yl = Y.substr(0, fh);
    string Yr = Y.substr(fh, sh);

    // Recursively calculate the three products of inputs of size n/2
    long int P1 = multiply(Xl, Yl);
    long int P2 = multiply(Xr, Yr);
    long int P3 = multiply(addBitStrings(Xl, Xr), addBitStrings(Yl, Yr));

    // Combine the three products to get the final result.
    return P1*(1<<(2*sh)) + (P3 - P1 - P2)*(1<<sh) + P2;
}
```

Strassen‘s Method

Strassen's method

- Key idea: multiply 2-by-2 blocks with only 7 multiplications. (plus 11 additions and 7 subtractions)

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_1 + P_5 - P_3 - P_7 \end{aligned}$$

$$\begin{aligned} P_1 &\leftarrow A_{11} \times (B_{12} - B_{22}) \\ P_2 &\leftarrow (A_{11} + A_{12}) \times B_{22} \\ P_3 &\leftarrow (A_{21} + A_{22}) \times B_{11} \\ P_4 &\leftarrow A_{22} \times (B_{21} - B_{11}) \\ P_5 &\leftarrow (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &\leftarrow (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &\leftarrow (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

Pf.
$$\begin{aligned} C_{12} &= P_1 + P_2 \\ &= A_{11} \times (B_{12} - B_{22}) + (A_{11} + A_{12}) \times B_{22} \\ &= A_{11} \times B_{12} + A_{12} \times B_{22}. \ \checkmark \end{aligned}$$

multiset

可能用到的头文件

```
#include <set>
#include <iterator>
#include <xfunctional>
```

样例

```
#include <iostream>
#include <set>
#include <iterator>

using namespace std;

int main()
{
    // empty multiset container
    multiset <int, greater <int> > gquiz1;

    // insert elements in random order
    gquiz1.insert(40);
    gquiz1.insert(30);

    gquiz1.insert(60);
    gquiz1.insert(20);
    gquiz1.insert(50);
    gquiz1.insert(50); // 50 will be added again to the multiset unlike set
    gquiz1.insert(10);

    // printing multiset gquiz1
    multiset <int, greater <int> > :: iterator itr;
    cout << "\nThe multiset gquiz1 is : ";
    for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr)
    {
        cout << '\t' << *itr;
    }
    cout << endl;

    // assigning the elements from gquiz1 to gquiz2
    multiset <int> gquiz2(gquiz1.begin(), gquiz1.end());

    // print all elements of the multiset gquiz2
    cout << "\nThe multiset gquiz2 after assign from gquiz1 is : ";
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
    {
        cout << '\t' << *itr;
    }
    cout << endl;

    // remove all elements up to element with value 30 in gquiz2
    cout << "\ngquiz2 after removal of elements less than 30 : ";
    gquiz2.erase(gquiz2.begin(), gquiz2.find(30));
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
    {
        cout << '\t' << *itr;
    }

    // remove all elements with value 50 in gquiz2
    int num;
    num = gquiz2.erase(50);
    cout << "\ngquiz2.erase(50) : ";
    cout << num << " removed \t" ;
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
    {
        cout << '\t' << *itr;
    }

    cout << endl;

    //lower bound and upper bound for multiset gquiz1
    cout << "gquiz1.lower_bound(40) : "
         << *gquiz1.lower_bound(40) << endl;
    cout << "gquiz1.upper_bound(40) : "
         << *gquiz1.upper_bound(40) << endl;

    //lower bound and upper bound for multiset gquiz2

    cout << "gquiz2.lower_bound(40) : "
```

```
         << *gquiz2.lower_bound(40) << endl;
    cout << "gquiz2.upper_bound(40) : "
         << *gquiz2.upper_bound(40) << endl;

         return 0;

}
```

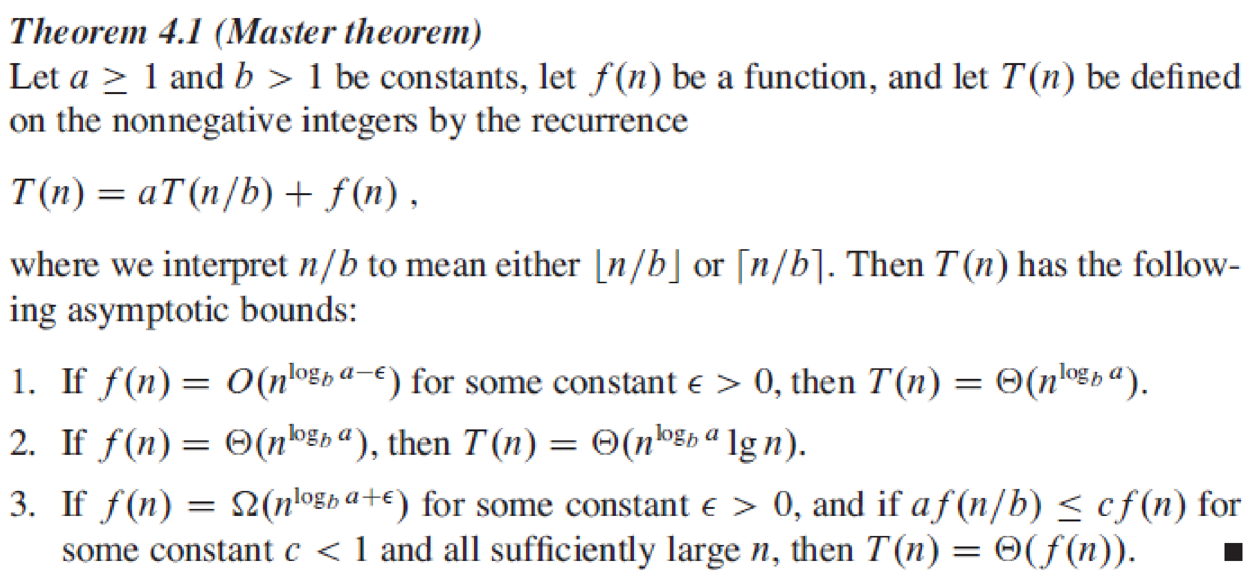
输出

```
The multiset gquiz1 is :  60    50  50  40  30  20  10

The multiset gquiz2 after assign from gquiz1 is :  10    20 30  40  50  50  60

gquiz2 after removal of elements less than 30 :  30 40  50  50  60
gquiz2.erase(50) : 2 removed        30  40  60
gquiz1.lower_bound(40) : 40
gquiz1.upper_bound(40) : 30
gquiz2.lower_bound(40) : 40
gquiz2.upper_bound(40) : 60
```

算法分析主定理 Master theorem



Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

最近点对 Closest Pair

时间复杂度 $o(nlognlogn)$

```
#include <stdio.h>
#include <float.h>

#include <stdlib.h>
```

```
#include <math.h>

// A structure to represent a Point in 2D plane
struct Point
{
    int x, y;
};

/* Following two functions are needed for library function qsort().
Refer: http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */

// Needed to sort array of points according to X coordinate
int compareX(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->x - p2->x);
}
// Needed to sort array of points according to Y coordinate
int compareY(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->y - p2->y);
}

// A utility function to find the distance between two points
float dist(Point p1, Point p2)
{
    return sqrt((p1.x - p2.x)*(p1.x - p2.x) +
        (p1.y - p2.y)*(p1.y - p2.y)
    );
}

// A Brute Force method to return the smallest distance between two points
// in P[] of size n
double bruteForce(Point P[], int n)
{
    double min = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i + 1; j < n; ++j)
            if (dist(P[i], P[j]) < min)
                min = dist(P[i], P[j]);
    return min;
}

// A utility function to find minimum of two float values
double min(double x, double y)
{
    return (x < y) ? x : y;
}


// A utility function to find the distance beween the closest points of

// strip of given size. All points in strip[] are sorted accordint to
// y coordinate. They all have an upper bound on minimum distance as d.
// Note that this method seems to be a O(n^2) method, but it's a O(n)
// method as the inner loop runs at most 6 times
double stripClosest(Point strip[], int size, float d)
{
    double min = d;  // Initialize the minimum distance as d

    qsort(strip, size, sizeof(Point), compareY);

    // Pick all points one by one and try the next points till the difference
    // between y coordinates is smaller than d.
    // This is a proven fact that this loop runs at most 6 times
    for (int i = 0; i < size; ++i)
        for (int j = i + 1; j < size && (strip[j].y - strip[i].y) < min; ++j)
            if (dist(strip[i], strip[j]) < min)
                min = dist(strip[i], strip[j]);

    return min;
}

// A recursive function to find the smallest distance. The array P contains
// all points sorted according to x coordinate
double closestUtil(Point P[], int n)
{
    // If there are 2 or 3 points, then use brute force
    if (n <= 3)
        return bruteForce(P, n);

    // Find the middle point
    int mid = n / 2;
    Point midPoint = P[mid];

    // Consider the vertical line passing through the middle point
    // calculate the smallest distance dl on left of middle point and
    // dr on right side
    double dl = closestUtil(P, mid);
    double dr = closestUtil(P + mid, n - mid);

    // Find the smaller of two distances
    double d = min(dl, dr);

    // Build an array strip[] that contains points close (closer than d)
    // to the line passing through the middle point
    Point *strip = new Point[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(P[i].x - midPoint.x) < d)
            strip[j] = P[i], j++;

    // Find the closest points in strip.  Return the minimum of d and closest
    // distance is strip[]
    return min(d, stripClosest(strip, j, d));

}
```

```
// The main functin that finds the smallest distance
// This method mainly uses closestUtil()
double closest(Point P[], int n)
{
    qsort(P, n, sizeof(Point), compareX);

    // Use recursive function closestUtil() to find the smallest distance
    return closestUtil(P, n);
}
// Driver program to test above functions
int main()
{
    Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
    int n = sizeof(P) / sizeof(P[0]);
    printf("The smallest distance is %f ", closest(P, n));
    return 0;
}
```