经典数据结构与算法

经典数据结构

向量 Vector

```
#define DEFAULT_CAPACITY X
using Data = int;

class Vector {
private:
    int _size;
    int _capacity;
    Data* _elem;
public:
    Data& operator[](int r) { return _elem[r]; }

    void expand(); // amortized: O(1)
    void insert(int r, Data const& e);
    void remove(int lo, int hi);

    void sort(); // O(nlogn)
    int find(Data& e, int lo, int hi) { // O(n)
        while ((lo < hi--) && (e != _elem[hi]));
        return hi;
    }
    int search(Data& e); // O(logn)
}; // S(n)
```

列表 List

```
struct Node {
    int data;
    Node* pred;
    Node* succ;
}

class List {
private:
    int _size;
    Node* head, tail;
public:
    List() {};
    void insert(int r, Node* e); // O(1)
    void remove(int r); // O(n)
}; // S(n)
```

栈 Stack

```
class Stack {
private:
    int size = 0;
    Node* nodes[MAXN];
public:
    void push(Node* e) { nodes[size++] = e; } // O(1)
    Node* pop() { return nodes[--size]; } // O(1)
    Node* top() { return nodes[size - 1]; } // O(1)
    bool empty() { return size == 0; }
}; // S(n)
```

队列 Queue

```
class Queue {
private:
    int size = 0;
    Node* head;
    Node* tail;
public:
    Queue() {
        head = new Node(); tail = new Node();
        head->succ = tail; tail->pred = head;
    }
    void enqueue(Node* e) { // O(1)
        size++;
        e->succ = tail;
        e->pred = tail->pred;
        tail->pred->succ = e;
        tail->pred = e;
    }
    Node* dequeue() { //O(1)
        size--;
        Node* ret = head->succ;
        head->succ = ret->succ;
        ret->succ->pred = head;
        return ret;
    }
    Node* top() { return head->succ; } // O(1)
    bool empty() { return size == 0; }
}; // S(n)
```

Steap (Stack + Heap)

```
class Steap {
private:
    Stack S, P;
public:
    Node* getMax() { return P.top(); } // O(1)
    void push(Node* e) { // O(1)
        P.push( max(e, P.top()) );
        S.push(e);
    }
    Node* pop() { // O(1)
        P.pop();
        return S.pop();
    }
}; // S(2n)
```

Queap (Queue + Heap)

```
class Queap {
private:
    Queue Q, P;
public:
    Node* getMax() { return P.top(); } // O(1)
    void enqueue(Node* e) { // O(n)
        P.enqueue(e);
        Q.enqueue(e);
        for (Node* x = P.tail; x && (x->data <= e); x = x->pred)
            x->data = e;
    }
    Node* dequeue() { // O(1)
        P.dequeue();
        return Q.dequeue();
    }
}; // S(2n)
```

BBST

1. AVL

```
struct Node {
    int data; int height;
    Node* parent;
    Node* lc; Node* rc;
    Node(int e = -1) : data(e), height(0) {};
    Node* succ();
};

class AVL {
```

```
private:
    int _size = 0;
    Node* _root;
    Node* _hot;
protected:
    bool isLChild(Node* x);
    Node* tallerChild(Node* x);
    bool avlBalanced(Node* x);
    void updateHeight(Node* x);
    Node* search(const int& e); // O(logn)
    void removeAt(Node* x, Node* hot);
    Node* connect34(Node* a, Node* b, Node* c, Node* t0, Node* t1, Node* t2, Node* t3);
    Node* rotateAt(Node* v);
    void processImbalance(Node* g);

public:
    AVL() : _root(nullptr), _hot(nullptr) {};
    void insert(const int& e) { // O(logn)
        Node* x = search(e);
        if (x) return;
        _size++;
        if (_hot == nullptr) _root = new Node(e);
        else {
            Node* node = new Node(e);
            (e < _hot->data) ? _hot->lc = node : _hot->rc = node;
            node->parent = _hot;
        }
        for (Node* g = _hot; g != nullptr; g = g->parent) {
            if (!avlBalanced(g)) {
                processImbalance(g);
                break;
            }
            else updateHeight(g);
        }
    }
    void remove(const int& e) { // O(logn)
        Node* x = search(e);
        removeAt(x, _hot);
        _size--;
        for (Node* g = _hot; g != nullptr; g = g->parent) {
            if (!avlBalanced(g)) {
                processImbalance(g);
            }
            updateHeight(g);
        }
    }
}
```

2. B树

```
const int M = 7;
const int T = 4;
```

```
class Node {
private:
    int* keys;
    Node** childrens;
    int num;
    bool leaf;
public:
    Node(bool isleaf) {
        keys = new DATA[M - 1];
        childrens = new Node* [M];
    }
    void insertNonFull(int v) {
        int i = num - 1;
        if (leaf) {
            while (i >= 0 && keys[i] > v) {
                keys[i + 1] = keys[i];
                i--;
            }
            keys[i + 1] = v;
            num++;
        }
        else {
            while (i >= 0 && keys[i] > v) i--;
            if (childrens[i + 1]->num == M - 1) {
                splitChild(i + 1, childrens[i + 1]);
                if (keys[i + 1] < v) i++;
            }
            childrens[i + 1]->insertNonFull(v);
        }
    }
    void splitChild(int i, Node* y);
    void traverse() {
        for (int i = 0; i < n; ++i) {
            if (leaf == false)
                childrens[i]->traverse();
            cout << " " << keys[i];
        }
    }
    Node* search(int v) { // O(logn)
        int i = 0;
        while (i < n && v > keys[i]) i++;
        if (i < n && keys[i] == v) return this;
        if (leaf) return NULL;
        return childrens[i]->search(v);
    }
    friend class BTree;
};

class BTree {
private:
    Node* root;
```

```
public:
    void traverse() { if (root) root->traverse(); }
    Node* search(int v) { return root ? root->search(v) : NULL; }
    void insert(int v) { // O(logn)
        if (!root) {
            root = new Node(true);
            root->keys[0] = v;
            root->num = 1;
        }
        else {
            if (root->num == M - 1) {
                Node* S = new Node(false);
                S->childrens[0] = root;
                S->splitChild(0, root);
                int i = 0;
                if (S->keys[0] < v) i++;
                S->childrens[i]->insertNonFull(v);
                root = S;
            }
            else root->insertNonFull(v);
        }
    }
    void delete(int v); // O(logn)
};
```

3. Red-Black Tree

```
class RedBlack { // Notation: [rotate, recolor, done]
protected:
    void solveDoubleRed(Node* x) {
        if (isRoot(*x)) {
            _root->color = BLACK;
            _root->height++;
            return;
        }
        Node* p = x->parent;
        if (isBlack(p)) return;
        Node* g = p->parent;
        Node* u = uncle(x);
        if (isBlack(u)) // RR-1: u->color == B [1~2, 2, √]
        else // RR-2: u->color == R [0, 3, x]
    }
    void solveDoubleBlack(Node* r) {
        Node* p = r ? r->parent : _hot;
        if (!p) return;
        Node* s = (r == p->lc) ? p->rc : p->lc;
        if (isBlack(s)) {
            Node* t = nullptr;
            if (isRed(s->rc)) t = s->rc;
            if (isRed(s->lc)) t = s->lc;
            if (t) // BB-1: s->c == R [1~2, 3, √]
            else {
```

```
                s->color = RED;
                s->height--;
                if (isRed(p)) // BB-2R: s->c == Bs, p == R [0, 2, √]
                else {// BB-2B: s->c == Bs, p == B [0, 1, x]
                    p->height--;
                    solveDoubleBlack(p);
                }
            }
        }
        else // BB-3: s == R [1, 2, x]
    }
    int updateHeight(Node* x) {
        return x->height = isBlack(x) + max(stature(x->lc), stature(x->rc));
    }
public:
    void insert(const int& e) { // O(logn)
        Node* x = search(e);
        if (x) return;
        x = new Node(e, _hot, NULL, NULL, 0);
        _size++;
        solveDoubleRed(x);
    }
    void remove(const int& e) { // O(logn)
        Node* x = search(e);
        if (!x) return;
        Node* r = removeAt(x, _hot);
        if (!(--_size)) return;
        if (!_hot) {
            _root->color = BLACK;
            updateHeight(_root);
            return;
        }
        if (blackHeightUpdated(*_hot))
        if (isRed(r)) {
            r->color = BLACK;
            r->height++;
            return;
        }
        solveDoubleBlack(r);
    }
};
```

4. Splay Tree

```
class Splay {
protected:
    Node* splay(Node* v) {
        if (!v) return nullptr;
        Node* p, g;
        while ((p = v->parent) && (g = p->parent)) {
            Node* gg = g->parent;
            if ( IsLChild( * v ) )
```

```
                if ( IsLChild( * p ) ) // zig-zig
                else // zig-zag
            else
                if ( IsRChild( * p ) ) // zag-zag
                else // zag-zig
            if ( !gg ) v->parent = nullptr;
            else ( g == gg->lc ) ? attachAsLC(v, gg) : attachAsRC(gg, v);
            updateHeight(g); updateHeight(p); updateHeight(v);
        }
    }
public:
    Node* search(const int& e) {
        Node* p = BST::search(e);
        _root = splay(p ? p : _hot);
        return _root;
    }
    void insert(const int& e) {
        if (!_root) {
            _size = 1;
            _root = new Node*(e);
            return;
        }
        Node* t = search(e);
        if (e == t->data) return;
        if (t->data < e) // _root->lc = t
        else // _root->rc = t
        _size++; updateHeightAbove(t);
    }
    void remove(const int& e) {
        if (!_root || (e != search(e)->data)) return;
        Node* L = _root->lc, R = _root->rc;
        release(_root);
        if (!R) {
            if (L) L->parent = NULL;
            _root = L;
        }
        else {
            _root = R;
            R->parent = NULL;
            search(e); // from R
            if (L) L->parent = _root;
            _root->lc = L;
        }
    }
};
```

跳转表 Skip List

```
struct QNode {
    int data;
```

```
    QNode* pred, succ, above, below;
    QNode* insert(int const& e, QNode* b = NULL);
};
struct QuadList {
    int _size;
    QNode* header, trailer;
    void remove(QNode* p);
    void insert(int const& e, QNode* p, QNode* b = NULL);
};
struct SkipList : public Dictionary<int, int>, public List<QuadList<Entry<int, int>>*> {
    QNode* search(int key); // expected-O(logn)
    int size();
    int height() { return List::size(); }
    void put(int key, int value); // expected-O(logn)
    int* get(int key);
    bool remove(int key); // expected-O(logn)
};
```

图 Graph

```
using VStatus = enum { UNDISCOVERED, DISCOVERED, VISITED };
struct Vertex {
    int data; int inDegree, outDegree;
    VStatus status;
    int dTime, fTime;
    int parent;
    int priority;
    Vertex(int const& d) : data(d), status(UNDISCOVERED), dTime(-1), fTime(-1), parent(-1),
priority(INT_MAX) {}
};

using EType = enum {UNDETERMINED, TREE, CROSS, FORWARD, BACKWARD};
struct Edge {
    int data;
    int weight;
    EType type;
    Edge(int const& d, int w) : data(d), weight(w), type(UNDETERMINED) {}
};

class GraphMatrix { // adjacency matrix
private:
    int n, e;
    Vector<Vertex> V;
    Vector<Vector<Edge>*> E;
public:
    void insertEdge(int const& edge, int w, int v, int u) {
        if (exists(v, u)) return;
        E[v][u] = new Edge(edge, w); e++;
        V[v].outDegree++;
        V[u].inDegree++;
```

```
    }
    void insertVertex(int const& vertex);
    void removeEdge(int v, int u);
    void removeVertex(int v);
}
```

并查集 DSU

> Disjoint Set Union

```
struct dsu {
    vector<int> parent;
    dsu(int size) : parent(size) { for (int i = 0; i < size; ++i) parent.put(i); }
    int find(int x) { return parent[x] == x ? x : find(parent[x]); }
    int find(int x) { return parent[x] == x ? x : parent[x] = find(parent[x]); }
    void unite(int x, int y) { parent[find(x)] = find(y); }
    void erase(int x) {
        --size[find(x)]; // vector<int> parent, size;
        parent[x] = x;
    }
};
```

时间复杂度：$O(\alpha(n))$，其中α表示Ackermann函数的反函数，单词操作平均运行时间接近于一个很小的常数

优先级队列（interface） Priority Queue

```
struct PQ {
    virtual void insert(int x) = 0;
    virtual int getMax() = 0;
    virtual int delMax() = 0;
};
```

完全二叉堆 Complete Heap

```
#define Parent(i) ( ((i) - 1) >> 1 ) // _root = 0
#define LChild(i) ( 1 + ((i) << 1) )
#define RChild(i) ( (1 + (i)) << 1)

class ComplHeap : public PQ, public Vector<int> {
protected:
    int percolateDown(int* A, int n, int i) { // O(logn)
        int j;
        while (i != (j = ProperParent(A, n, i))) {
            swap(A[i], A[j]);
            i = j;
```

```
        }
        return i;
    }
    int percolateUp(int* A, int i) { // O(logn)
        while (0 < i) {
            int j = Parent(i);
            if (A[i] < A[j]) break;
            swap(A[i], A[j]);
            i = j;
        }
        return i;
    }
    void heapify(int* A, int n) { // Floyd: O(n)
        for (int i = n / 2 - 1; 0 <= i; --i) percolateDown(A, n, i);
    }
public:
    ComplHeap(int* A, int n) { copyFrom(A, 0, n); heapify(_elem, n); }
    void insert(int x) { Vector::insert(x); percolateUp(_elem, _size - 1); }
    int getMax() { return _elem[0]; }
    void delMax() { _elem[0] = _elem[--_size]; percolateDown(_elem, _size, 0); }
}
```

锦标赛树 Tournament Tree

```
// Winner Tree
#define Parent(i) ( ((i) - 1) >> 1 ) // _root = 0
class TournamentTree {
private:
    vector<int> tree;
    int n;
public:
    TournamentTree(const vector<int>& elements) { // O(n)
        n = elements.size();
        int tree_size = 2 * n - 1;
        tree.resize(tree_size, INT_MAX);
        for (int i = 0; i < n; i++) tree[n - 1 + i] = elements[i];
        for (int i = n - 2; i >= 0; i--) tree[i] = min(tree[2 * i + 1], tree[2 * i + 2]);
    }
    int findMin() { return tree[0]; } // O(1)
    void update(int i, int new_val) { // O(logn)
        int index = n - 1 + i;
        tree[index] = new_val;
        while (index > 0) {
            index = Parent(index);
            tree[index] = min(tree[2 * index + 1], tree[2 * index + 2]);
        }
    }
};

// Loser Tree
```

```
#define Parent(x) = ((i) >> 1) // _root(winner) = 0
class TournamentTree {
private:
    vector<int> tree;
    int n ;
public:
    TournamentTree(const vector<int>& elements); // O(n)
    int findMin() { return tree[0]; } // O(1)
    void update(int i, int new_val) { // O(logn)
        int index = n + i;
        tree[index] = new_val;
        int cur_val = new_val;
        while (index > 0) {
            if (tree[Parent(index)] < cur_val) {
                swap(tree[Parent(index)], cur_val); // swap(&, &)
                index = Parent(index);
            }
            else index = Parent(index);
        }
        tree[0] = cur_val;
    }
};
```

左式堆 Left Heap

```
class LeftHeap : public PQ, public BinTree {
public:
    int getMax() { return _root->data; }
    void insert(int e) { // O(logn)
        _root = merge(_root, new Node(e, nullptr));
        _size++;
    }
    void delMax() { // O(logn)
        Node* lHeap = _root->lc;
        if (lHeap) lHeap->parent = nullptr;
        Node* rHeap = _root->rc;
        if (rHeap) rHeap->parent = nullptr;
        delete _root;
        _size--;
        _root = merge(lHeap, rHeap);
    }
    LeftHeap(LeftHeap& A, LeftHeap& B) {
        _root = merge(A._root, B._root);
        _size = A._size + B._size;
        A._root = B._root = nullptr;
        A._size = B._size = 0;
    }
};

Node* merge(Node* a, Node* b) { // O(logn)
```

```
    if (!a) return b;
    if (!b) return a;
    if (a->data < b->data) swap(a, b);
    (a->rc = merge(a->rc, b))->parent = a;
    if (!a->lc || a->lc->npl < a->rc->npl)
        swap(a->lc, a->rc);
    a->npl = a->rc ? 1 + a->rc->npl : 1;
    return a;
}

Node* merge(Node* a, Node* b) { // iterative version
    if (!a) return b;
    if (!b) return a;
    if (a->data < b->data) swap(a, b);
    for ( ; a->rc; a = a->rc) {
        if (a->rc->data < b->data) {
            b->parent = a;
            swap(a->rc, b);
        }
    }
    (a->rc = b)->parent = a;
    for ( ; a; b = a, a = a->parent) {
        if (!a->lc || a->lc->npl < a->rc->npl)
            swap(a->lc, a->rc);
        a->npl = a->rc ? a->rc->npl + 1 : 1;
    }
}
```

串 String

```
class String {
private:
    int n;
    char S[];
public:
    String substr(int i, int k) { return String(S, i, k); } // S[i, i + k)
    String prefix(int k) { return String(S, 0, k); } // S[0, k)
    String suffix(int k) { return String(S, n - k, k); } // S[n - k, n)
    int length();
    char charAt(int i);
    String concat(String s) { return String(S, s); }
    bool equal(String s);
    int indexOf(String s) { return search(s); }
};
// <string.h>: strlen(), strcpy(), strcat(), strcmp(), strstr()...
```

经典算法

快速幂

> 求a的n次方

```
int power(int a, int n) {
    int pow = 1, p = a; // O(1)
    while (0 < n) { // O(logn)
        if (n & 1) pow *= p;
        n >>= 1;
        p *= p;
    }
    return pow;
}
```

时间复杂度：$O(logn)$

countOnes

> 统计整数二进制展开中数位1的总数

```
int countOnes(unsigned int n) {
    int ones = 0;
    while (0 < n) {
        ones++;
        n &= n - 1;
    }
    return ones;
}
```

时间复杂度：$O(ones)$，正比于数位1的总数

总和最大区段

> 从整数序列$A[n]$中，求总和最大的区段的和（有多个时，短者优先）

```
int gs_LS(int A[], int n) {
    int gs = A[0], s = 0, i = n;
    while (0 < i--) {
        s += A[i];
        if (gs < s) gs = s;
        if (s <= 0) s = 0;
    }
    return gs;
}
```

时间复杂度：$O(n)$

斐波那契数列

> 斐波那契数列：1, 1, 2, 3, 5, 8... 求第n个数字（从1开始编号）

```
int fib(int n) {
    int g = 0, f = 1;
    while (0 < n--) {
        g = g + f;
        f = g - f;
    }
    return g;
}
```

时间复杂度：$O(n)$

最长公共子序列 LCS

> Longest Common Subsequence problem
> 求两序列（长度分别为n和m）最长公共子序列中的长度

```
unsigned int lcs(char const * A, int n, char const * B, int m) {
    if (n < m) { //make sure n >= m
        swap(A, B);
        swap(n, m);
    }
    unsigned int* lcs1 = new unsigned int[m + 1];
    unsigned int* lcs2 = new unsigned int[m + 1];
    memset(lcs1, 0x00, sizeof(unsigned int) * (m + 1));
    memset(lcs2, 0x00, sizeof(unsigned int) * (m + 1));
    for (int i = 0; i < n; swap(lcs1, lcs2), i++)
        for (int j = 0; j < m; j++)
            lcs2[j + 1] = (A[i] == B[j]) ? 1 + lcs1[j] : max(lcs2[j], lcs1[j + 1]);
    unsigned int solu = lcs1[m];
    delete[] lcs1;
    delete[] lcs2;
    return solu;
}
```

时间复杂度：$O(mn)$

就地循环移位

> 仅用$O(1)$辅助空间，将数组$A[0,n)$中的元素向左循环移动k个单位

- **迭代** 时间复杂度：$O(2n)$

```
int shift( int* A, int n, int s, int k ) { // O(n/GCD(n, k))
    int b = A[s];
    int i = s, j = (s + k) % n;
    int mov = 0;
    while ( s != j ) {
        A[i] = A[j];
        i = j;
        j = (j + k) % n;
        mov++;
    }
    A[i] = b;
    return mov + 1;
}

void shift1(int* A, int n, int k) { //O(g) = O(GCD(n, k))
    for (int s = 0, mov = 0; mov < n; s++)
        mov += shift(A, n, s, k);
}
```

- **倒置** 时间复杂度：$O(3n)$

```
void shift2(int* A, int n, int k) {
    reverse(A, k); // O(3k/2)
    reverse(A + k, n - k); // O(3(n-k)/2)
    reverse(A, n); // O(3n/2)
}
```

埃氏筛法 Eratosthenes

> 找出1~n中所有的素数

```
void Eratosthenes(int n, char* file) {
    Bitmap B(n);
    B.set(0); B.set(1);
    for (int i = 2; i < n; ++i) {
        if (!B.test(i))
            for (int j = i * i; j < n; j += i)
                B.set(j);
    }
    B.dump(file);
}
```

时间复杂度：$O(nlogn)$

二分查找 BS

> Binary Search
> 有序（可能有重）数列S中查找指定值e，返回不大于e的最后一个元素。

```
static Rank bs(T* S, T const& e, Rank lo, Rank hi) {
    while (lo < hi) {
        Rank mi = (lo + hi) >> 1;
        e < S[mi] ? hi = mi : lo = mi + 1;
    }
    return lo - 1;
}
```

时间复杂度：$O(logn)$

findKth

> 无序数列S中查找第k大的元素

```
Node nodes[MAXN];
bool cmp(Node a, Node b) { return a.data < b.data; } // NOT ≤ !
void findKth(int low, int high, bool(*cmp)(Node, Node), int k) { // [low, high], k = 1, 2,
...
    if (low == high) return;
    Node pivot = nodes[(low + high) >> 1];
    int i = low, j = high;
    i--; j++;
    while (i < j) {
        while (cmp(nodes[++i], pivot));
        while (cmp(pivot, nodes[--j]));
        if (i < j) swap(nodes[i], nodes[j]);
    }
    if (j == high) j--;
    i = j - low + 1;
    if (k <= i) return findKth(low, j, cmp, k);
    else return findKth(j + 1, high, cmp, k - i);
}

void quickSelect(Vector<int>& A, int k) { // expected-O(4n)
    for (int lo = 0, hi = A.size() - 1; lo < hi;) {
        int i = lo, j = hi;
        int pivot = A[lo];
        while (i < j) {
            while (i < j && pivot <= A[j]) j--; A[i] = A[j];
            while (i < j && A[i] <= pivot) i++; A[j] = A[i];
        } // assert: quit with i == j
        A[i] = pivot;
        if (k <= i) hi = i - 1;
        if (i <= k) lo = i + 1;
    }
```

```
}

void linearSelect(Vector<int>& A, int n, int k); // Q = 5
```

时间复杂度：$O(n)$

排序 Sort

> 将无序（可能有重）数列$S[lo, hi)$从小到大排序

1. 冒泡排序 BubbleSort

```
void bubbleSort(int lo, int hi) { // 跳跃版
    for (int last; lo < hi; hi = last)
        for (int i = (last = lo) + 1; i < hi; ++i)
            if (S[i - 1] > S[i])
                swap(S[i - 1], S[last = i]);
}
```

时间复杂度：最好$O(n)$，最坏$O(n^2)$

2. 归并排序 MergeSort

```
void mergeSort(int lo, int hi) {
    if (hi - lo < 2) return;
    int mi = (lo + hi) >> 1;
    mergeSort(lo, mi);
    mergeSort(mi, hi);
    merge(lo, mi, hi);
}

void merge(int lo, int mi, int hi) {
    int i = 0; int* A = S + lo;
    int j = 0, lb = mi - lo; int* B = new int[lb];
    for (int t = 0; t < lb; ++t)
        B[t] = A[t];
    int k = 0, lc = hi - mi; int* C = S + mi;

    while ((j < lb) && (k < lc))
        A[i++] = (B[j] <= C[k]) ? B[j++] : C[k++];
    while (j < lb)
        A[i++] = B[j++];
    delete [] B;
}
```

时间复杂度：$O(nlogn)$

3. 选择排序 SelectionSort

```
void selectionSort(Node* p, int n) { // [p, p + n)
```

```
    Node* head = p->pred, tail = p;
    for (int i = 0; i < n; ++i)
        tail = tail->succ;
    while (1 < n) {
        insert( remove( selectMax(head->succ, n) ), tail );
        tail = tail->pred; n--;
    }
}

Node* selectMax(Node* p, int n) {
    Node* max = p;
    for (Node* cur = p; 1 < n; n--)
        if ( (cur = cur->succ)->data >= max->data )
            max = cur;
    return max;
}
```

时间复杂度：$O(n^2)$，但元素移动操作显著少于冒泡排序

4. 插入排序 InsertionSort

```
void insertionSort(Node* p, int n) {
    for (int r = 0; r < n; ++r) {
        insert( search(p->data, r, p), p->data );
        p = p->succ;
        remove(p->pred);
    }
}
```

时间复杂度：最好$O(n)$，最坏$O(n^2)$

5. 桶排序 BucketSort

```
const int MAXN = 1000;

int n, w, a[MAXN]; // w = max(a[n])
vector<int> bucket[MAXN];

void bucketSort() {
    int size = w / n + 1;
    for (int i = 0; i < n; ++i) bucket[i].clear();
    for (int i = 1; i <= n; ++i) bucket[a[i] / size].push_back(a[i]);
    int p = 0;
    for (int i = 0; i < n; ++i) {
        insertionSort(bucket[i], bucket[i].size());
        for (int j = 0; j < bucket[i].size(); ++j)
            a[++p] = bucket[i][j];
    }
}
```

时间复杂度：平均$O(n + \frac{n^2}{k} + k) \approx (n)$，最坏$O(n^2)$（其中$n$为数组元素数量，$k$为桶数量）

6. 基数排序 RadixSort

```
int maxBit(int data[], int n) {
    int d = 1;
    int p = 10;
    for (int i = 0; i < n; ++i) {
        while (data[i] >= p) {
            p *= 10;
            ++d;
        }
    }
    return d;
}

void radixSort(int data[], int n) {
    int d = maxBit(data, n);
    int tmp[n];
    int count[10];
    int radix = 1;
    for (int i = 1; i <= d; ++i) {
        for (int j = 0; j < 10; ++j) count[j] = 0;
        for (int j = 0; j < n; ++j) {
            int k = (data[j] / radix) % 10;
            count[k]++;
        }
        for (int j = 1; j < 10; ++j) count[j] = count[j - 1] + count[j];
        for (int j = n - 1; j >= 0; --j) {
            int k = (data[j] / radix) % 10;
            tmp[count[k] - 1] = data[j];
            count[k]--;
        }
        for (int j = 0; j < n; ++j) data[j] = tmp[j];
        radix *= 10;
    }
}
```

时间复杂度：$O(n)$（若将位数d视作常数）

7. 堆排序 HeapSort

```
void heapSort(int lo, int hi) { // [lo, hi)
    int* A = _elem + lo;
    int n = hi - lo;
    heapify(A, n);
    while (0 < --n) {
        swap(A[0], A[n]);
        percolateDown(A, n, 0);
    }
}
```

时间复杂度：$O(nlogn)$

8. 快速排序 QuickSort

- 递归版本:

```
void quickSort(int lo, int hi) {
    if (hi - lo < 2) return;
    int mi = partition(lo, hi);
    quickSort(lo, mi);
    quickSort(mi + 1, hi);
}

int partition(int lo, int hi) { // LUG
    swap(_elem[lo], _elem[lo + rand() % (hi - lo)]);
    int pivot = _elem[lo];
    while (lo < hi) {
        do { hi--; } while (lo < hi && pivot <= _elem[hi]);
        if (lo < hi) _elem[lo] = _elem[hi];
        do { lo++; } while (lo < hi && _elem[lo] <= pivot);
        if (lo < hi) _elem[hi] = _elem[lo];
    } // assert: lo == hi || hi + 1
    _elem[hi] = pivot;
    return hi;
}

int partition(int lo, int hi) { // DUP
    swap(_elem[lo], _elem[lo + rand() % (hi - lo)]);
    int pivot = _elem[lo];
    while (lo < hi) {
        do { hi--; } while (lo < hi && pivot < _elem[hi]);
        if (lo < hi) _elem[lo] = _elem[hi];
        do { lo++; } while (lo < hi && _elem[lo] < pivot);
        if (lo < hi) _elem[hi] = _elem[lo];
    } // assert: lo == hi || hi + 1
    _elem[hi] = pivot;
    return hi;
}

int partition(int lo, int hi) { // LGU
    swap(_elem[lo], _elem[lo + rand() % (hi - lo)]);
    int pivot = _elem[lo];
    int mi = lo;
    for (int k = lo + 1; k < hi; ++k) {
        if (_elem[k] < pivot) swap(_elem[++mi], _elem[k]);
    }
    swap(_elem[lo], _elem[mi]);
    return mi;
}
```

时间 / 空间复杂度 / 递归深度：最好$O(nlogn)$，最坏$O(n)$

- 迭代 + 贪心 版本:

```
#define Put(K, s, t) { if (1 < (t) - (s)) { K.push(s); K.push(t); } }
#define Get(K, s, t) { t = K.pop(); s = K.pop(); }
void quickSort(int lo, int hi) {
    Stack<int> Task;
    Put(Task, lo, hi);
    while (!Task.empty()) {
        Get(Task, lo, hi);
        int mi = partition(lo, hi);
        if (mi - lo < hi - mi) { Put(Task, mi + 1, hi); Put(Task, lo, mi); }
        else { Put(Task, lo, mi); Put(Task, mi + 1, hi); }
    }
}
```

空间复杂度：辅助栈空间不超过$O(logn)$（递归深度可能超过）

统计序列逆序对

> 统计序列A中逆序对的个数

```
int mergeSort(int lo, int hi) {
    int count = 0;
    if (hi - lo < 2) return 0;
    int mi = (lo + hi) >> 1;
    count += mergeSort(lo, mi);
    count += mergeSort(mi, hi);
    count += merge(lo, mi, hi);
    return count;
}

int merge(int lo, int mi, int hi) {
    int cnt = 0;
    int i = 0; int* A = S + lo;
    int j = 0, lb = mi - lo; int* B = new int[lb];
    for (int i = 0; i < lb; ++i)
        B[i] = A[i];
    int k = 0, lc = hi - mi; int* C = S + mi;

    while ((j < lb) && (k < lc)) {
        if (B[j] > C[k]) {
            cnt += (mi - lo - j); // count inversions
            A[i++] = C[k++];
        }
        else A[i++] = B[j++];
    }
    while (j < lb)
        A[i++] = B[j++];
    delete [] B;
    return cnt;
}
```

时间复杂度：$O(nlogn)$

最大间隙问题 MaxGap

> 给定n个实数$x_1, x_2, \ldots, x_n$，求这些数在实轴上相邻2个数之间的最大差值。

```
int max(int n, double x[]) {
    int m = MIN_INT;
    for (int i = 0; i < n; i++)
        if (x[i] > x[m]) m = i;
    return m;
}

int min(int n, double x[]) {
    int m = MAX_INT;
    for (int i = 0; i < n; i++)
        if (x[i] < x[m]) m = i;
    return m;
}

double maxGap(int n, double x[]) {
    double maxx = x[max(n, x)];
    double minx = x[min(n, x)];
    int* cnt = new int[n];
    double* high = new double[n];
    double* low = new double[n];
    for (int i = 0; i < n; ++i) { // init
        cnt[i] = 0;
        high[i] = minx;
        low[i] = maxx;
    }
    for (int i = 0; i < n; ++i) {
        double avergap = (maxx - minx) / (n - 1);
        int index = (x[i] - minx) / avergap);
        if (index == n - 1) index = n - 2;
        cnt[index]++;
        if (x[i] > high[index]) high[index] = x[i];
        if (x[i] < low[index]) low[index] = x[i];
    }
    double ans = 0;
    double left = high[0];
    for (int i = 0; i < n; ++i) {
        if (cnt[i]) {
            double thisgap = low[i] - left;
            if (thisgap > ans) ans = thisgap;
            left = high[i];
        }
    }
    return ans;
}
```

时间复杂度：$O(n)$

进制转换

> 以10进制转16进制为例，自高而低输出结果

```
void convert(Stack<char>& S, __int64 n, int base) {
    char digit[] = "0123456789ABCDEF";
    while (n > 0) {
        S.push( digit[n % base] );
        n /= base;
    }
    while (!S.empty())
        printf("%c", S.pop());
}
```

时间复杂度：$O(n)$，n为原整数位数

括号匹配

> 给定算式x，判断算式中的括号是否匹配。设只存在一种括号。

```
bool paren(const char x[], int lo, int hi) { //x[lo, hi)
    Stack<char> S;
    for (int i = lo; i < hi; ++i) {
        if ('(' == x[i]) S.push(x[i]);
        else if (!S.empty()) S.pop();
        else return false;
    }
    return S.empty();
}
```

时间复杂度：$O(n)$，n为算式字符长度

甄别禁形（非栈混洗）

> Stack Permutation
> 给定1~n的一个序列x，判断该序列是否为栈混洗

```
bool sp(const int x[], int lo, int hi) { // A[lo, hi)
    Stack<int> A, B, S;
    for (int i = hi; i > lo; --i)
        A.push(i);
    for (int i = lo; i < hi; ) {
        if (!S.empty() && x[i] == S.top()) {
            B.push(S.pop());
```

```
            ++i;
        }
        else if (!A.empty()) S.push(A.pop());
        else return false;
    }
    return true;
}
```

时间复杂度：$O(n)$

表达式求值

> 给定含运算符和数字的表达式，求解其值

1. 中缀表达式求值 Infix Notation

```
bool isDigit(char s);
void readNumber(char* S, Stack<double> opnd);
double calc(char op, double opnd);
double calc(double opnd1, char op, double opnd2);
const char pri[N_OPTR][N_OPTR] = {
    /*--   +  */    '>', '>', '<', '<', '<', '>', '>',
    /* |   -  */    '>', '>', '<', '<', '<', '>', '>',
    /* T   *  */    '>', '>', '>', '<', '<', '>', '>',
    /* o   ^  */    '>', '>', '>', '>', '<', '>', '>',
    /* p   (  */    '<', '<', '<', '<', '<', '=', ' ',
    /* |   )  */    ' ', ' ', ' ', ' ', ' ', ' ', ' ',
    /*--  \0  */    '<', '<', '<', '<', '<', ' ', '='
    //               +    -    *    ^    (    )   \0
    //               |---------Current Optr--------|
};
char priority(char top, char cur);

double evaluate(char* S) {
    Stack<double> opnd; // operand stack
    Stack<char> optr; // operator stack
    optr.push('\0');
    while (!optr.empty()) {
        if (isDigit(*S))
            readNumber(S, opnd);
        else
            switch(priority(optr.top(), *S)) {
                case '<':
                    optr.push(*S);
                    S++;
                    break;
                case '=':
                    optr.pop();
                    S++;
                    break;
```

```
                case '>':
                    char op = optr.pop();
                    if ('!' == op)
                        opnd.push(calc(op, opnd.pop()));
                    else {
                        double opnd2 = opnd.pop();
                        double opnd1 = opnd.pop();
                        opnd.push(calc(opnd1, op, opnd2));
                    }
                    break;
            }
    }
    return opnd.pop();
}
```

时间复杂度：$O(n)$

2. 后缀表达式求值 RPN

> Reverse Polish Notation

```
double evaluate(char* S) {
    Stack<double> opnd; // operand stack
    Stack<char> optr; // operator stack
    optr.push('\0');
    while (!optr.empty()) {
        if (isDigit(*S))
            readNumber(S, opnd);
        else {
            if ('!' == *S)
                opnd.push(calc(*S, opnd.pop()));
            else {
                double opnd2 = opnd.pop();
                double opnd1 = opnd.pop();
                opnd.push(calc(opnd1, op, opnd2));
            }
        }
    }
    return opnd.pop();
}
```

时间复杂度：$O(n)$

3. 中缀表达式转后缀表达式求值

直方图内最大矩形

> $H[0,n)$为非负整数的直方图，求直方图内最大的矩形面积（若有多种情况同解，则取最左边的解）

```
int maxRect(int H[], int n) {
```

```
    Stack<int> SR;
    int maxRect = 0;
    for (int t = 0; t <= n; t++) {
        while (!SR.empty() && (t == n || H[SR.top()] > H[t])) {
            int r = SR.pop();
            int s = SR.empty() ? 0 : SR.top() + 1;
            maxRect = max(maxRect, H[r] * (t - s));
        }
        if (t < n) SR.push(t);
    }
    return maxRect;
}
```

时间复杂度：$O(n)$

二叉树的遍历

时间复杂度：$O(n)$

1. 前序遍历

递归写法：

```
void travPre(Node* x, VST& visit) {
    if (!x) return;
    visit(x->data);
    travPre(x->lc, visit);
    travPre(x->rc, visit);
}
```

迭代写法：

```
void travPre(Node* x, VST& visit) {
    Stack<Node*> S;
    while(true) {
        while(x) {
            visit(x->data);
            S.push(x->rc);
            x = x->lc;
        }
        if (S.empty()) break;
        x = S.pop();
    }
}
```

2. 中序遍历

递归写法：

```
void travIn(Node* x, VST& visit) {
```

```
    if (!x) return;
    travIn(x->lc, visit);
    visit(x->data);
    travIn(x->rc, visit);
}
```

迭代写法:

```
void travIn(Node* x, VST& visit) {
    Stack<Node*> S;
    while(true) {
        while(x) {
            S.push(x);
            x = x->lc;
        }
        if (S.empty()) break;
        x = S.pop();
        visit(x->data);
        x = x->rc;
    }
}
```

3. 后序遍历

递归写法:

```
void travPost(Node* x, VST& visit) {
    if (!x) return;
    travPost(x->lc, visit);
    travPost(x->rc, visit);
    visit(x->data);
}
```

迭代写法:

```
void travPost(Node* x, VST& visit) {
    Stack<Node*> S;
    if (x) S.push(x);
    while (!S.empty()) {
        if (S.top() != x->parent) {
            while (x = S.top()) {
                if (HasLChild(*x)) {
                    if (HasRChild(*x))
                        S.push(x->rc);
                    S.push(x->lc);
                }
                else S.push(x->rc);
            }
            S.pop(); // null
        }
        x = S.pop();
```

```
        visit(x->data);
    }
}
```

4. 层次遍历

```
void travLevel(Node* x, VST& visit) {
    Queue<Node*> Q;
    Q.enqueue(x);
    while (!Q.empty()) {
        x = Q.dequeue();
        visit(x->data);
        if (HasLChild(*x)) Q.enqueue(x->lc);
        if (HasRChild(*x)) Q.enqueue(x->rc);
    }
}
```

Huffman算法：最优带权编码树

> 已知待编码的字符集$ch[]$及其出现频率$freq[]$，求最优二进制编码规则使得编码后的总字符最短，输出字符集中各字符的编码。

1. 优先级队列

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

#define MAXN 100

class Node { // must use class instead of struct
public:
    char data;
    int freq;
    Node* left;
    Node* right;
    Node(char ch, int fr) : data(ch), freq(fr) {};
};

class Compare {
public:
    bool operator()(Node* a, Node* b) {
        return a->freq > b->freq;
    }
};

Node* generateTree(priority_queue<Node*, vector<Node*>, Compare> pq) {
    while (pq.size() != 1) {
        Node* left = pq.top();
        pq.pop();
```

```
        Node* right = pq.top();
        pq.pop();
        Node* node = new Node('^', left->freq + right->freq);
        node->left = left;
        node->right = right;
        pq.push(node);
    }
    return pq.top();
}

void printCodes(Node* root, int arr[], int top) {
    if (root->left) {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }
    if (root->right) {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }
    if (!root->left && !root->right) {
        cout << root->data << " ";
        for (int i = 0; i < top; i++)
            cout << arr[i];
        cout << endl;
    }
}

void HuffmanCodes(char data[], int freq[], int size) {
    priority_queue<Node*, vector<Node*>, Compare> pq;
    for (int i = 0; i < size; ++i) {
        Node* newNode = new Node(data[i], freq[i]);
        pq.push(newNode);
    }
    Node* root = generateTree(pq);
    int arr[MAXN], top = 0;
    printCodes(root, arr, top);
}

int main() {
    char data[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    int size = sizeof(data) / sizeof(data[0]);
    HuffmanCodes(data, freq, size);
    return 0;
}
```

时间复杂度：初始化优先级队列$O(n)$ + 建树$O(nlogn)$ = $O(nlogn)$

2. 栈+有序队列

```
#include <iostream>
#include <cstdlib>
```

```
using namespace std;

#define MAXN 100

struct Node {
    char data;
    int freq;
    Node* left;
    Node* right;
    Node* pred;
    Node* succ;
    Node(char ch, int fr) : data(ch), freq(fr) {};
};

class Stack {
public:
    int size = 0;
    Node* nodes[MAXN];
    void push(Node* e) { nodes[size++] = e; }
    Node* pop() { return nodes[--size]; }
    Node* top() { return nodes[size - 1]; }
    bool empty() { return size == 0; }
};

class Queue {
public:
    int size = 0;
    Node* head;
    Node* tail;
    Queue() {
        head = new Node('^', 0);
        tail = new Node('^', 0);
        head->succ = tail;
        tail->pred = head;
    }
    void push(Node* e) {
        size++;
        e->succ = tail;
        e->pred = tail->pred;
        tail->pred->succ = e;
        tail->pred = e;
    }
    Node* pop() {
        size--;
        Node* ret = head->succ;
        head->succ = ret->succ;
        ret->succ->pred = head;
        return ret;
    }
    Node* top() { return head->succ; }
    bool empty() { return size == 0; }
};
```

```
Node* buildTree(Stack& stack, Queue& queue) {
    if (stack.size == 1) {
        Node* root = new Node('^', stack.top()->freq);
        root->left = stack.pop();
        return root;
    }
    Node* node1 = stack.pop();
    Node* node2 = stack.pop();
    while (true) {
        Node* newNode = new Node('^', node1->freq + node2->freq);
        newNode->left = node1;
        newNode->right = node2;
        queue.push(newNode);
        if (stack.empty() && queue.size == 1) return newNode;
        if (stack.empty()) {
            node1 = queue.pop();
            node2 = queue.pop();
        }
        else {
            node1 = (stack.top()->freq <= queue.top()->freq) ? stack.pop() : queue.pop();
            if (!stack.empty() && !queue.empty())
                node2 = (stack.top()->freq <= queue.top()->freq) ? stack.pop() :
queue.pop();
            else
                node2 = (stack.empty()) ? queue.pop() : stack.pop();
        }
    }
}

int cmp(const void* a, const void* b) {
    return (*(Node**)b)->freq - (*(Node**)a)->freq;
}

void printCodes(Node* root, int arr[], int top) {
    if (root->left) {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }
    if (root->right) {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }
    if (!root->left && !root->right) {
        cout << root->data << " ";
        for (int i = 0; i < top; i++)
            cout << arr[i];
        cout << endl;
    }
}

void HuffmanCodes(char data[], int freq[], int size) {
```

```
    Node* vec[MAXN];
    for (int i = 0; i < size; ++i) {
        Node* newNode = new Node(data[i], freq[i]);
        vec[i] = newNode;
    }
    qsort(vec, size, sizeof(Node*), cmp);
    Stack S;
    Queue Q;
    for (int i = 0; i < size; ++i)
        S.push(vec[i]);
    Node* root = buildTree(S, Q);

    int arr[MAXN], top = 0;
    printCodes(root, arr, top);
}

int main() {
    char data[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    int size = sizeof(data) / sizeof(data[0]);
    HuffmanCodes(data, freq, size);
    return 0;
}
```

时间复杂度：排序建栈$O(nlogn)$ + 队列建树$O(n)$ = $O(nlogn)$

KD-tree

1. 1D Range Query

> 给定在x轴上的点集$P = p_1, p_2, \ldots, p_n$。对于任意给定的区间$I = (x_1, x_2]$，求 ① 点集$P$落在区间$I$中的点个数，并 ② 遍历所有$I \cap P$中的点

2. Planar Range Query

> 给定在平面上的点集$P = p_1, p_2, \ldots, p_n$。对于任意给定的二维区间$R = (x_1, x_2] \times (y_1, y_2]$，求 ① 点集$P$落在区间$R$中的点个数，并 ② 遍历所有$R \cap P$中的点
>
> *下例出自DSA PA3-5 temperature*

```
Node nodes[MAXN];
Node tree[MAXN * 4];

void buildTree(int low, int high, int depth, int p) {
    if (low == high) {
        tree[p] = nodes[low];
        return;
    }
    if (depth % 2) findKth(low, high, cmp_x, ((low + high) / 2));
    else findKth(low, high, cmp_y, ((low + high) / 2));
    buildTree(low, (low + high) / 2, depth + 1, 2 * p);
```

```
    buildTree((low + high) / 2 + 1, high, depth + 1, 2 * p + 1);
    tree[p].lx = min(tree[2 * p].lx, tree[2 * p + 1].lx);
    tree[p].ly = min(tree[2 * p].ly, tree[2 * p + 1].ly);
    tree[p].rx = max(tree[2 * p].rx, tree[2 * p + 1].rx);
    tree[p].ry = max(tree[2 * p].ry, tree[2 * p + 1].ry);
    tree[p].low = min(tree[2 * p].low, tree[2 * p + 1].low);
    tree[p].high = max(tree[2 * p].high, tree[2 * p + 1].high);
}
// buildTree(0, n - 1, 0, 1)  /* p = 1 */

struct Answer { int low; int high; bool valid; }
answer { INT_MAX, -1, false};

void inquiry(int x1, int x2, int y1, int y2, int p) {
    if (tree[p].low >= answer.low && tree[p].high <= answer.high)
        return; // 剪枝
    if (x1 <= tree[p].lx && x2 >= segment[p].rx
        && y1 <= tree[p].ly && y2 >= tree[p].ry) {
        answer = {min(tree[p].low, answer.low), max(tree[p].high, answer.high), true}
        return;
    }
    if (x1 > tree[p].rx || x2 < tree[p].lx || y1 > tree[p].ry || y2 < tree[p].ly)
        return;
    inquiry(x1, x2, y1, y2, 2 * p);
    inquiry(x1, x2, y1, y2, 2 * p + 1);
}
// inquiry(x1, x2, y1, y2, 1)
```

时间复杂度：建树$O(nlogn)$ + 单次查询$O(r+\sqrt{n})$

Interval Tree

> Given a set of intervals in general position on the **x**-axis: $S=\{s_i=[x_i,x_i'|1\le i\le n]\}$ and a query point q_x
> Find all intervals that contain q_x: $\{s_i=[x_i,x_i']|x_i\le q_x\le x_i'\}$

```
class Interval {
public:
    int label;
    int left;
    int right;
    Interval* parent;
    Interval(int data);
    Interval(int data, int l, int r);
    Interval(int data, int l, int r, Interval parent);
    friend ostream& operator<<(ostream& output, Interval interval);
};

ostream& operator<<(ostream& output, Interval interval) {
    output << "(" << interval.label << "):  [" << interval.left << ", " << interval.right <<
"]\n";
```

```
    return output;
}

bool LSortFunc(const Interval& i1, const Interval& i2) {
    return i1.left < i2.left;
}

bool RSortFunc(const Interval& i1, const Interval& i2) {
    return i1.right > i2.right;
}

class IntervalTree {
public:
    int xMed; // middle point
    vector<Interval> sMed; // sMed = mL + mR
    vector<Interval> mL; // left points sorted vector
    vector<Interval> mR;  // right points sorted vector
    vector<Interval> lMed; // left
    vector<Interval> rMed; // right
    IntervalTree* lTree;
    IntervalTree* rTree;
    void findKth(int low, int high, int k, vector<int>& S);
    int findMedium(vector<Interval>& source) {
        vector<int> S;
        for (auto i = source.begin(); i < source.end(); ++i) {
            S.push_back(i->left);
            S.push_back(i->right);
        }
        findKth(0, source.size() * 2 - 1, source.size(), S);
        return S[source.size()];
    }
    void buildTree(vector<Interval>& source) {
        xMed = findMedium(source);
        int cnt = 0;
        for (auto i = source.begin(); i < source.end(); ++i) {
            if (xMed < i->left) rMed.push_back(*i);
            else if (xMed > i->right) lMed.push_back(*i);
            else {
                sMed.push_back(*i);
                if (i->left != xMed) {
                    Interval* lInterval = new Interval(sMed[cnt].label, i->left, xMed,
sMed[cnt]);
                    mL.push_back(*lInterval);
                }
                if (i->right != xMed) {
                    Interval* rInterval = new Interval(sMed[cnt].label, xMed, i->right,
sMed[cnt]);
                    mR.push_back(*rInterval);
                }
                cnt++;
            }
        }
```

```
        sort(mL.begin(), mL.end(), LSortFunc);
        sort(mR.begin(), mR.end(), RSortFunc);
        if (lMed.size() > 0) {
            lTree = new IntervalTree();
            lTree->buildTree(lMed);
        }
        else lTree = nullptr;
        if (rMed.size() > 0) {
            rTree = new IntervalTree();
            rTree->buildTree(rMed);
        }
        else rTree = nullptr;
    }
    void piercingQuery(int x) {
        if (x < xMed) {
            for (auto i = mL.begin(); i < mL.end(); ++i) {
                if (x < i->left) break;
                cout << *(i->parent);
            }
            if (lTree != nullptr) lTree->piercingQuery(x);
        }
        else if ( x > xMed) {
            for (auto i = mR.begin(); i < mR.end(); ++i) {
                if (x > i->right) break;
                cout << *(i->parent);
            }
            if (rTree != nullptr) rTree->piercingQuery(x);
        }
        else {
            for (auto i = sMed.begin(); i < sMed.end(); ++ i) {
                cout << *i;
            }
        }
    }
};
```

时间复杂度：建树$O(nlogn)$ + 穿刺查询$O(r+logn)$

Segment Tree

> 一般情况

```
void build(int s, int t, int p) {
    if (s == t) {
      d[p] = a[s];
      return;
    }
    int m = s + ((t - s) >> 1);
    build(s, m, p * 2), build(m + 1, t, p * 2 + 1);
    d[p] = d[p * 2] + d[(p * 2) + 1];
```

```
}
int inquiry(int l, int r, int s, int t, int p) {
    if (l <= s && t <= r) return d[p];
    if (lazy[p]) {
        d[p * 2] += lazy[p] * (m - s + 1);
        d[p * 2 + 1] += lazy[p] * (t - m);
        lazy[p * 2] += lazy[p];
        lazy[p * 2 + 1] += lazy[p];
        lazy[p] = 0;
    }
    int m = s + ((t - s) >> 1), sum = 0;
    if (l <= m) sum += inquiry(l, r, s, m, p * 2);
    if (r > m) sum += inquiry(l, r, m + 1, t, p * 2 + 1);
    return sum;
}
// lazy mark
void rangeUpdate(int l, int r, int c, int s, int t, int p) {
    if (l <= s && t <= r) {
        d[p] += (t - s + 1) * c, lazy[p] += c;
        return;
    }
    int m = s + ((t - s) >> 1);
    if (lazy[p] && s != t) {
        d[p * 2] += lazy[p] * (m - s + 1);
        d[p * 2 + 1] += lazy[p] * (t - m);
        lazy[p * 2] += lazy[p];
        lazy[p * 2 + 1] += lazy[p];
        lazy[p] = 0;
    }
    if (l <= m) rangeUpdate(l, r, c, s, m, p * 2);
    if (r > m) rangeUpdate(l, r, c, m + 1, t, p * 2 + 1);
    d[p] = d[p * 2] + d[p * 2 + 1];
}
```

> *下例出自DSA PA 3-4-2 Kidd*
>
> 长大后的黑羽快斗也成为了一位魔术师，他开始了扑克牌练习。
>
> 他首先把一叠扑克以正面朝上的姿态顺次在桌上排成一行，之后的每一次挥一挥衣袖，都会反转一连串的扑克，改变它们的正反朝向。为了检验自己的心算能力，快斗会随时问自己：目前从第i张扑克牌到第j张扑克牌（包含第i和第j张扑克牌），一共被反转过几次？

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

const int MAXM;

int n, m;
int query_len = 0;
int map_len = 0;

struct Operations {
    char label[2]; // H: rangeIncread, Q: inquiry
```

```
    int left;
    int right;
} operations[MAXM];

struct Query {
    int point;
    int map_label;
} query[MAXM * 2];

struct Map {
    int left;
    int right;
} map[MAXM * 4];

struct SegmentTree {
    int sum;
    int lazy;
} segmentTree[MAXM * 16];

int comp(const void* e1, const void* e2)
    return ((Query*)e1)->point - ((Query*)e2)->point;

void initialize() {
    cin >> n >> m;
    readRanges();
    qsort(query, 2 * m, sizeof(Query), comp);
    query_deduplicate();
    init_map(); // 1,5,6,9 => [1,1], [2,4], [5,5], [6,6], [7,8], [9,9], [10,10]
    init_segmentTree(); // sum = lazy = 0
}

int rev_map (int point); // point => map_label

void pushDown(int s, int t, int p) {
    int mid = (s + t) >> 1;
    if (segmentTree[p].lazy != 0 && s != t) {
        segmentTree[p * 2].sum += segmentTree[p].lazy * (map[mid].right - map[s].left + 1);
        segmentTree[p * 2 + 1].sum += segmentTree[p].lazy * (map[t].right - map[mid +
1].left + 1);
        segmentTree[p * 2].lazy += segmentTree[p].lazy;
        segmentTree[p * 2 + 1].lazy += segmentTree[p].lazy;
        segmentTree[p].lazy = 0;
    }
}

int inquiry(int l, int r, int s, int t, int p) {
    // [l, r]: target range; [s, t]: current range; p: curret label
    if (l <= s && t <= r) return segmentTree[p].sum;
    pushDown(s, t, p);
    int mid = (s + t) >> 1;
    int ans = 0;
    if (l <= mid) ans += inquiry(l, r, s, mid, p * 2);
```

```
    if (r >= mid + 1) ans += inquiry(l, r, mid + 1, t, p * 2 + 1);
    return ans;
}

void rangeIncrease(int l, int r, int s, int t, int p) {
    // [l, r]: target range; [s, t]: current range; p: curret label
    if (l <= s && t <= r) {
        segmentTree[p].sum += map[t].right - map[s].left + 1;
        segmentTree[p].lazy += 1;
        return;
    }
    pushDown(s, t, p);
    int mid = (s + t) >> 1;
    if (l <= mid) rangeIncrease(l, r, s, mid, p * 2);
    if (r >= mid + 1) rangeIncrease(l, r, mid + 1, t, p * 2 + 1);
    segmentTree[p].sum = segmentTree[p * 2].sum + segmentTree[p * 2 + 1].sum;
}
```

时间复杂度：$O(mlog(m))$（m为离散化叶节点数量）

搜索图

1. 广度优先搜索 BFS

```
void bfs(int v, int& clock) {
    Queue<int> Q;
    status(v) = DISCOVERED;
    Q.enqueue(v);
    while (!Q.empty()) {
        int v = Q.dequeue();
        dTime(v) = ++clock;
        for (int u = firstNbr(v); -1 < u; u = nextNbr(v, u))
            if (UNDICOVERED == status(u)) {
                status(u) = DISCOVERED;
                Q.enqueue(u);
                type(v, u) = TREE;
                parent(u) = v;
            }
            else type(v, u) = CROSS;
        status(v) = VISITIED;
    }
}
```

时间复杂度：$O(n + e)$

- 应用1：根据给定顶点计算连通分量和可达分量

```
void bfs_all(int s) {
    reset();
```

```
    int clock = 0;
    int v = s;
    do {
        if (UNDISCOVERED == status(v)) bfs(v, clock);
    } while (s != (v = ((v + 1) % n)));
}
```

- 应用2：无向图中的最短距离 = bfs树的搜索通路

2. 深度优先搜索 DFS

```
void dfs(int v, int& clock) {
    dTime(v) == ++clock;
    status(v) = DISCOVERED;
    for (int u = firstNbr(v); -1 < u; u = nextNbr(v, u)) {
        switch (status(u)) {
            case UNDISCOVERED:
                typr(v, u) = TREE;
                parent(u) = v;
                dfs(u, clock);
                break;
            case DISCOVERED:
                type(v, u) = BACKWARD;
                break;
            default: // VISITED
                type(v, u) = dTime(v) < dTime(u) ? FORWARD : CROSS;
        }
        status(v) = VISITED;
        fTime(v) = ++clock;
    }
}
```

时间复杂度：$O(n+e)$

- 应用1：根据给定顶点计算连通分量和可达分量

```
void dfs_all(int s) {
    reset();
    int clock = 0;
    int v = s;
    do {
        if (UNDISCOVERED == status(v)) dfs(v, clock);
    } while (s != (v = ((v + 1) % n)));
}
```

- 应用2：有向图环路检测 = 存在BACKWARD的边

3. 优先级搜索 PFS

```
void pfs(int v, void(*prioUpdater(Graph*, int, int))) {
    priority(v) = 0;
```

```
    status(v) = VISITED;
    parent(v) = -1;
    while (true) {
        for (int u = firstNbr(v); -1 < u; u = nextNbr(v, u))
            prioUpdater(this, v, u);
        for (int shortest = INT_MAX, u = 0; u < n; ++u) {
            if (UNDISCOVERED == status(u) && shortest > priority(u)) {
                shortest = priority(u);
                v = u;
            }
        }
        if (VISITED == status(v)) break;
        status(v) = VISITED;
        type(parent(v), v) = TREE;
    }
}
```

时间复杂度：更新优先级$O(n+e)$ + 选出优先级最高者$O(n^2)$（邻接表）

- 采用优先级队列 / 完全二叉堆：更新优先级$O(elogn)$ + 选出优先级最高者$O(nlogn)$
- 采用多叉堆：更新优先级$O(n \cdot d \cdot log_d n)$ + 选出优先级最高者$O(elog_d n)$，取$d \approx \frac{e}{n}+2$，总体性能达到$O(e \cdot log_{(\frac{e}{n}+2)} n)$最优

拓扑排序 TSort

> TSort: Topological Sort；有向无环图：Directed Acyclic Graph
> 任给有向图G（不一定是DAG），尝试将所有顶点排成一个**线性序列**，使其次序与原图**相容**（每一顶点都不会通过边指向**前驱**顶点）。若原图存在回路（即并非DAG），检查并报告。否则，给出一个相容的线性序列。

顺序输出零入度顶点：

```
Queue<int> TSort() {
    Stack<int> S;
    Queue<int> Q;
    S.push_back(find_zero_indegree());
    while (!S.empty()) {
        int v;
        Q.enqueue(v = S.pop());
        for (int u = firstNbr(v); -1 < u; u = nextNbr(v, u)) {
            if (u.inDegree < 2) S.push(u);
            remove(v);
        }
    }
    return Graph::size() ? -1 : Q;
}
```

时间复杂度：$O(n+e)$

逆序输出零出度顶点：

```
bool TSort(int v, int& clock, Stack<int>* S) {
    dTime(v) = ++clock;
    status(v) = DISCOVERED;
    for (int u = firstNbr(v); -1 < u; u = nextNbr(v, u)) {
        switch (status(u)) {
            case UNDICOVERED:
                parent(u) = v;
                type(v, u) = TREE;
                if (!TSort(u, clock, S)) return false;
                break;
            case DISCOVERED:
                type(v, u) = BACKWARD;
                return false;
            default: // VISITED
                type(v, u) = dTime(v) < dTime(u) ? FORWARD : CROSS;
                break;
        }
        status(v) = VISITED;
        S->push(vertex(v));
    }
}
```

时间复杂度：$O(n+e)$

最短路径

1. Dijkstra算法

> 给定顶点s，计算s到其余各个顶点的最短路径及长度
> 假定：各边权重均为正

```
// include pfs
struct DijkPu {
    virtual void operator()(Graph* g, int v, int u) {
        if (UNDISCOVERED != g->status(u)) return;
        if (g->priority(u) > g->priority(v) + g->weight(v, u)) {
            g->priority(u) = g->priority(v) + g->weight(v, u);
            g->parent(u) = v;
        }
    }
}; // g->pfs(0, DijkPu());
```

时间复杂度：$O(n^2)$（通过其他数据结构可以优化到$O(elogn), O(eloge)$）

2. Floyd-Warshall算法

> 给定带权网络G，计算器中所有点对之间的最短距离

```
void fw(int n, int A[][]) {
    for (int k = 0; k < n; ++k)
```

```
        for (int u = 0; u < n; ++u)
            for (int v = 0; v < n; ++v)
                A[u][v] = min(A[u][v], A[u][k] + A[k][v]);
}
```

时间复杂度：$O(n^3)$

最小支撑树 MST

> Minimum Spanning Tree
> 求联通网络$N=(V;E)$的子图$T=(V;F)$，使得T的总权重$w(T)=\sum_{e\in F}w(e)$最小

1. Prim算法

```
// include pfs
struct PrimPu {
    virtual void operator()(Graph* g, int v, int u) {
        if (UNDISCOVERED != g->status(u)) return;
        if (g->priority(u) > g->weight(v, u)) {
            g->priority(u) = g->weight(v, u);
            g->parent(u) = v;
        }
    }
}; // g->pfs(0, PrimPu());
```

时间复杂度：$O((n+e)logn$（优先级队列算法），$O(n^2)$（普通PFS）

2. Kruskal算法

```
int parent[MAXN];

// before: qsort(edges, m, sizeof(edge), cmp);
void kruskal() {
    int sumWeight = 0;
    int u, v;
    ufSet();
    for (int i = 0; i < m; ++i) {
        int u = edge[i].u;
        int v = edge[i].v;
        if (find(u) != find(v)) {
            sumWeight += edges[i].w;
            parent[u] = v;
        }
    }
}

void ufSet() { for (int i = 0; i < n; ++i) parent[i] = i; }
int find(int x) { return parent[x] == x ? x : parent[x] = find(parent[x]); }
```

时间复杂度：$O(eloge+elogn)$

串匹配

1. KMP算法

> Knuth-Morris-Pratt

```
int match(char* P, char* T) { // O(2n)
    int* next = buildNext(P);
    int n = (int) strlen(T), i = 0;
    int m = (int) strlen(P), j = 0;
    while (j < m && i < n) {
        if (j < 0 || T[i] == P[j]) { i++; j++; }
        else j = next[j];
    }
    delete [] next;
    return i - j;
}

int* buildNext(char* P) {
    int m = strlen(P), j = 0;
    int* N  = new int[m];
    int t = N[0] = -1;
    while (j < m - 1) {
        if (t < 0 || P[j] == P[t]) {
            ++j; ++t;
            N[j] = (P[j] != P[t]) ? t : N[t];
        }
        else t = N[t];
    }
    return N;
}
```

时间复杂度：$O(n)$

2. BM算法

> Boyer-Moore
> BM算法 = BC + GS

```
int* buildBC(char* P) { // Bad_Character: O(s + m)
    int* bc = new int[256];
    for (int j = 0; j < 256; ++j) bc[j] = -1;
    for (int m = strlen(P), j = 0; j < m; ++j) bc[P[j]] = j;
    return bc;
}

int* buildGS(char* P); // Good Suffix

int match(char* P, char* T);
```

时间复杂度：Best Case: $O(\frac{n}{m})$，Worst Case: $O(n+m)$

3. Karp-Rabin算法