# Lab 1 实验报告

PB20000296 郑滕飞

## 框架构建：

想实现算法的比较，需要按照要求建立统一的比较框架，这里除主函数外分为两个函数：gen_input 生成输入与 test_sort 测试算法。

生成输入函数如下：

```
void gen_input(void) {
    ofstream outfile;
    char filepath[20];
    const int size = (int)pow(2, 18);
    cout << "gen begin" << endl;
    for (int j = 1; j <= 5; j++) {
        sprintf_s(filepath, "../input/%d.txt", j);
        outfile.open(filepath, ios::out);
        for (int k = 0; k < size; k++)
            outfile << rand() << endl;
        outfile.close();
        cout << '.';
    }
    cout << endl << "gen end" << endl << endl;
}
```

由于单次输入可能存在极端的数据，一共生成了五份$2^{18}$个数的输入，每个输入数据是 rand()随机生成的整数，保存在 input 文件夹中的 1.txt 到 5.txt 中，之后在进行$2^3$排序时是取其前 8 个数，其他数量依此类推。

测试算法的第一部分如下：

```
void test_sort(void (*sort)(int* A, int n), const char* name) {
    char outpath[40];
    sprintf_s(outpath, "cd ../output && mkdir %s", name);
    system(outpath);

    LARGE_INTEGER start, ende;
    ifstream infile;
    ofstream outfile, time;
    char infilepath[20];
    int* A;

    sprintf_s(outpath, "../output/%s/time.txt", name);
    time.open(outpath, ios::out);

    cout << name << endl;
```

除了定义一些变量外，这部分在 output 文件夹中建立了算法名所对应的文件夹，并在其中建立了 time 文件。之后的时间输出即在 time 文件中。

由于数据规模较小时时间很短，需要微妙精度的计时，因此我在计时时选用了 Query

Performance 函数。每次排序的结果放在文件夹中对应的 2^a_b.txt 中，其中 2^a 为数据规模，b 为对应第几个输入中的。而每次的时间在 time.txt 与控制台中同时输出。

```
for (int i = 3; i <= 18; i += 3) {
    time << "2^" << i << ":\t";
    cout << "2^" << i << ":\t";
    int size = (int)pow(2, i);
    A = new int[size];
    for (int j = 1; j <= 5; j++) {
        sprintf_s(infilepath, "../input/%d.txt", j);
        infile.open(infilepath, ios::in);
        for (int k = 0; k < size; k++) infile >> A[k];
        infile.close();
        QueryPerformanceCounter(&start);
        sort(A, size);
        QueryPerformanceCounter(&ende);
        time << ende.QuadPart - start.QuadPart << '\t';
        cout << ende.QuadPart - start.QuadPart << '\t';
        sprintf_s(outpath, "../output/%s/2^%d_%d.txt", name, i, j);
        outfile.open(outpath, ios::out);
        for (int k = 0; k < size; k++) outfile << A[k] << endl;
        outfile.close();
    }
    time << endl;
    cout << endl;
    delete[]A;
}
cout << endl;
time.close();
```

**算法实现：**

接下来是对于四个算法的具体实现。此部分主要来自上课记录的老师 ppt，并对数组下标起始为 0 还是 1 作了一些调整。

在上方的测试排序中，函数指针要求输入数组与数据规模，其中数组下标默认为从 0 开始，到数据规模-1。

对于堆排序，按照 ppt 的写法构造了调整堆、建堆两个函数，并且以这两个函数实现了堆排序：

```
void max_heapify(int* A, int i, int hsize) {
    int largest;
    while (1) {
        int l = 2 * i + 1, r = 2 * i + 2;
        if (l < hsize && A[l] > A[i]) largest = l;
        else largest = i;
        if (r < hsize && A[r] > A[largest]) largest = r;
        if (largest == i) return;
        swap(A[i], A[largest]);
        i = largest;
    }
}
```

```cpp
void build_max_heap(int* A, int n) {
    for (int i = n / 2; i >= 0; i--) max_heapify(A, i, n);
}

void heapsort(int* A, int n) {
    build_max_heap(A, n);
    int hsize = n;
    for (int i = n - 1; i >= 1; i--) {
        swap(A[0], A[i]);
        hsize--;
        max_heapify(A, 0, hsize);
    }
}
```

快速排序与归并排序的实现方式有些类似，不过一个是先调用 `partition`，一个是后调用 `merge`：

```cpp
int partition(int* A,int p, int r) {
    int x = A[r], i = p - 1;
    for (int j = p; j < r; j++) {
        if (A[j] <= x) {
            i++;
            swap(A[i], A[j]);
        }
    }
    swap(A[i + 1], A[r]);
    return i + 1;
}

void quicksort(int* A, int p, int r) {
    if (p < r) {
        int q = partition(A, p, r);
        quicksort(A, p, q - 1);
        quicksort(A, q + 1, r);
    }
}
```

```cpp
void merge(int* A, int p, int q, int r) {
    int n1 = q - p + 1, n2 = r - q;
    int* L = new int[n1 + 1];
    int* R = new int[n2 + 1];
    for (int i = 0; i < n1; i++) L[i] = A[p + i];
    for (int i = 0; i < n2; i++) R[i] = A[q + i + 1];
    L[n1] = R[n2] = INT_MAX;
    int i = 0, j = 0;
    for (int k = p; k <= r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i];
            i++;
        }
        else {
            A[k] = R[j];
            j++;
        }
    }
}

void mergesort(int* A, int p, int r) {
    if (p < r) {
        int q = (p + r) / 2;
        mergesort(A, p, q);
        mergesort(A, q + 1, r);
        merge(A, p, q, r);
    }
}
```

由于需要进行递归，都需要输入数组下标的起始与终止。为了达到输入数组与规模也能进行排序的效果，将 `sort(A,n)`重载为 `sort(A,0,n-1)`即可。

计数排序则是通过新建辅助数组达到效果：

```cpp
void countingsort(int* A, int n) {
    int* B = new int[RAND_MAX+1];
    int* C = new int[n];
    for (int i = 0; i <= RAND_MAX; i++) B[i] = 0;
    for (int i = 0; i < n; i++) B[A[i]]++;
    for (int i = 1; i <= RAND_MAX; i++) B[i] += B[i - 1];
    for (int i = n - 1; i >= 0; i--)
        C[--B[A[i]]] = A[i];
    for (int i = 0; i < n; i++) A[i] = C[i];
}
```

**结果统计：**

运行一次的控制台直接输出如下:

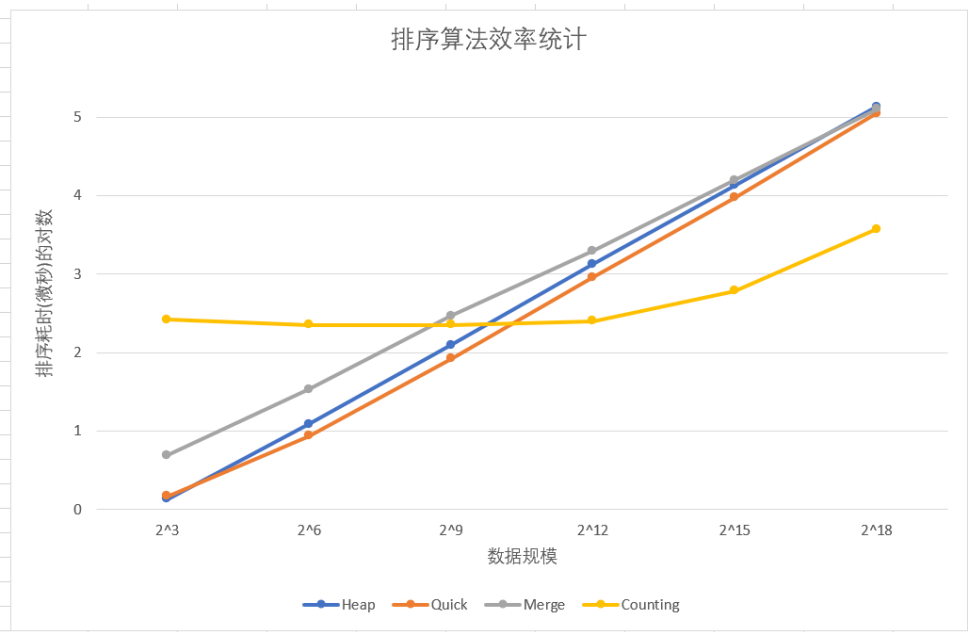
```
Microsoft Visual Studio 调试控制台
gen begin
.....
gen end

10000000 ticks per second
the following timings are all in ticks

HeapSort
2^3:    12      11      12      15      18
2^6:    137     148     105     108     110
2^9:    1243    1240    1254    1239    1240
2^12:   13275   13296   13583   13446   13366
2^15:   136684  133455  136862  133735  132608
2^18:   1347450 1319405 1335385 1361319 1360836

QuickSort
2^3:    17      11      25      14      7
2^6:    83      77      84      82      108
2^9:    880     813     759     861     857
2^12:   9410    9059    8754    9311    9089
2^15:   89427   97234   88880   98639   93884
2^18:   1048104 1050277 1138113 1085698 1264713

MergeSort
2^3:    31      78      36      61      41
2^6:    402     327     333     291     337
2^9:    2484    4053    3078    2504    2473
2^12:   21095   22350   19355   17449   18621
2^15:   150009  161459  180354  144983  149067
2^18:   1275131 1236694 1255743 1185102 1338090

CountingSort
2^3:    2010    2339    4174    2383    2309
2^6:    2784    2263    2093    1988    2176
2^9:    2381    2283    2119    2182    2348
2^12:   2514    2624    2572    2411    2411
2^15:   5980    6226    5938    5592    6736
2^18:   36213   37253   42441   35071   36531


D:\Desktop\exersize\src\x64\Debug\sorting.exe (进程 24800)已退出，代码为 0。
按任意键关闭此窗口. . .
```

由于系统的 CPU 时钟是每秒$10^7$次，下方的单位事实上是 0.1 微妙。将五个不同数据集上的结果进行平均，可得到最终数据:

| Average | Heap | Quick | Merge | Counting |
|---|---|---|---|---|
| **2^3** | 13.6 | 14.8 | 49.4 | 2643 |
| **2^6** | 121.6 | 86.8 | 338 | 2260.8 |
| **2^9** | 1243.2 | 834 | 2918.4 | 2262.6 |
| **2^12** | 13393.2 | 9124.6 | 19774 | 2506.4 |
| **2^15** | 134668.8 | 93612.8 | 157174.4 | 6094.4 |
| **2^18** | 1344879 | 1117381 | 1258152 | 37501.8 |

但是，由于下方坐标轴是指数增长的，如果直接以此作图，前期的区分极小，因此作图时先对数据取了以 10 为底的对数，并减 1，也即最终作图是通过所需微秒数的以 10 为底的对数:

| Average | Heap | Quick | Merge | Counting |
|---|---|---|---|---|
| 2^3 | 13.6 | 14.8 | 49.4 | 2643 |
| 2^6 | 121.6 | 86.8 | 338 | 2260.8 |
| 2^9 | 1243.2 | 834 | 2918.4 | 2262.6 |
| 2^12 | 13393.2 | 9124.6 | 19774 | 2506.4 |
| 2^15 | 134668.8 | 93612.8 | 157174.4 | 6094.4 |
| 2^18 | 1344879 | 1117381 | 1258152 | 37501.8 |




可以发现，计数排序由于是 n+k 量级，而 k 是定值，在数据规模小时效果显著不如其他三种，但数据规模很大时(也即 n 的比重比 k 大时)则会显著更优。另外三种排序则都是 n log n 量级，由于 rand()生成的随机数是均匀的，总体来说快速排序的性能好于另外两个。而在数量级不大时，堆排序的效果显著好于归并排序，但增大后优势逐渐消失，甚至被反超。

为此，我做了更多测试，并发现在 2^18 量级时，归并排序几乎稳定好于堆排序。然而，效率的代价是它并不是就地排序，在数据规模大时需要较多的额外空间。

按实验报告要求，最后附上 2^3 时对五组输入的排序结果，详见 output 文件夹：

| | | | | |
|---|---|---|---|---|
| 3283 | 1979 | 6107 | 110 | 2818 |
| 4440 | 4695 | 11673 | 2180 | 6279 |
| 5775 | 10031 | 13856 | 7519 | 8931 |
| 12081 | 11877 | 14335 | 11469 | 11265 |
| 15743 | 12131 | 15463 | 15323 | 11419 |
| 20491 | 15603 | 18487 | 21879 | 15183 |
| 22391 | 25532 | 30170 | 24843 | 23272 |
| 24754 | 27462 | 32339 | 26691 | 31199 |

# Lab 2 实验报告

PB20000296 郑滕飞

## 矩阵链乘：

### 1、代码实现

算法部分与书上基本相同：

```
long long min_matrix_times(vector<int> p) {
    const int size = p.size() - 1;
    vector<vector<long long>> m(size, vector<long long>(size));
    for (int l = 1; l < size; l++) for (int i = 0; i < size - l; i++) {
        int j = i + l;
        m[i][j] = INFTY;
        for (int k = i; k < j; k++) {
            long long q = m[i][k] + m[k+1][j] + (long long)p[i]*p[k+1]*p[j+1];
            if (q < m[i][j]) {
                m[i][j] = q;
                m[j][i] = k;
            }
        }
    }
    outfile << m[0][size-1] << endl;
    print_result(m, 0, size-1);
    outfile << endl;
    return m[0][size-1];
}
```

注意到 m 只利用了上三角部分，而 s 只需要不用对角线的下三角部分，可以直接用 m 的下三角部分存储 s 的转置，打印结果时也利用转置即可：

```
void print_result(vector<vector<long long>> s, int i, int j) {
    if (i == j) {
        outfile << 'A' << i;
        return;
    }
    outfile << '(';
    print_result(s, i, s[j][i]);
    print_result(s, s[j][i] + 1, j);
    outfile << ')';
}
```

控制台输出部分如下：

```
QueryPerformanceFrequency(&start);
cout << start.QuadPart << " ticks per second" << endl;
cout << "the following timings are all in ticks" << endl << endl;
infile.open("../input/2_1_input.txt", ios::in);
outfile.open("../output/result.txt", ios::out);
time.open("../output/time.txt", ios::out);
int size;
for (int i = 0; i < 5; i ++) {
    infile >> size;
    vector<int> p(size + 1);
    cout << "size: " << size << endl;
    for (int t = 0; t <= size; t++) infile >> p[t];
    QueryPerformanceCounter(&start);
    cout << "result: " << min_matrix_times(p) << endl;
    QueryPerformanceCounter(&ende);
    time << ende.QuadPart - start.QuadPart << endl;
    cout << "time: " << ende.QuadPart - start.QuadPart << endl;
}
outfile.close();
time.close();
infile.close();
```

控制台输出大小，结果与花费的时间，由于时间较短，采用 query 进行微秒级别计时。

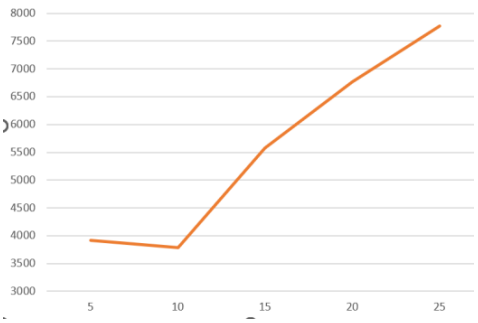
**2、结果展示**

控制台输出如下：

```
10000000 ticks per second
the following timings are all in ticks

size: 5
result: 154865959097238
time: 3919
size: 10
result: 42524697503391
time: 3784
size: 15
result: 5400945319618
time: 5578
size: 20
result: 319329979644400
time: 6772
size: 25
result: 574911761218280
time: 7777
```

n=5 时的结果为(A0(((A1A2)A3)A4))，其余具体结果见 output 文件夹。

时间与规模分布如图：




由于数据规模很小，只能基本看出上升趋势，进一步分析需要扩大规模，经测试与理论结果$n^3$量级基本一致。

## 最长公共子序列：

**1、代码实现**

利用书上思路可写出打印与计算的代码：

```
void print_result(vector<vector<char>> t, string a) {
    int m = t.size() - 1;
    int n = t[0].size() - 1;
    string d = "";
    while (m >= 0 && n >= 0) {
        if (t[m][n] == 'd') {
            d = a[m] + d;
            m--;
            n--;
        }
        else if (t[m][n] == 'u') m--;
        else n--;
    }
    cout << d << endl;
    outfile << d << endl;
}
```

```
int longest_common_subsequence(string a, string b) {
    int m = a.length(), n = b.length();
    vector<vector<int>> c(m+1, vector<int>(n+1));
    vector<vector<char>> t(m, vector<char>(n));
    for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) {
        if (a[i] == b[j]) {
            c[i+1][j+1] = c[i][j] + 1;
            t[i][j] = 'd';
        }
        else if (c[i][j+1] > c[i+1][j]) {
            c[i+1][j+1] = c[i][j+1];
            t[i][j] = 'u';
        }
        else {
            c[i+1][j+1] = c[i+1][j];
            t[i][j] = 'l';
        }
    }
    char outpath[30];
    sprintf_s(outpath, "../output/result_%d.txt", m);
    outfile.open(outpath, ios::out);
    outfile << c[m][n] << endl;
    print_result(t, a);
    outfile.close();
    return c[m][n];
}
```

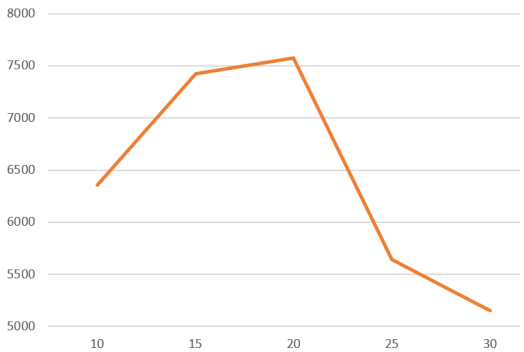
控制台输出部分与矩阵链乘基本相同。

**2、结果展示**

控制台输出如下:

```
10000000 ticks per second
the following timings are all in ticks

size: 10
DABAB
time: 6357
size: 15
ACCBDCCC
time: 7424
size: 20
CBACDADCBBDA
time: 7572
size: 25
CDDBBBCCDDBADD
time: 5642
size: 30
ADDBBCDBBCDDDCBD
time: 5179
```

时间与规模分布如图:



由于规模过小，运行时的随机性要素会比理论结果更加明显。自行加大规模后，可以发现结果量级与$n^2$基本一致，符合理论计算。

(但加大规模也不能太大，空间消耗事实上比时间更明显)
(所以到底为什么要在这种规模下画运行时间截图……)

# Lab 3 实验报告

PB20000296 郑滕飞

## 红黑树基本操作：

[这部分过多抄书内容，就不赘述了。]

### 1、类型定义

结点类型的定义为：

```
struct node{
    int low;
    int high;
    int m;
    bool color;
    node* left;
    node* right;
    node* p;
};
```

low 与 high 表示区间端点，m 为子树中最大元素。

而红黑树类包含 root 与 nil 两个结点和一些成员函数，之后将进行解释：

```
class IntervalTree {
    private:
        node* nil;
        node* root;
        void print_tree(node* now, int depth);
        void left_rotate(node* x);
        void right_rotate(node* x);
        void insert_fixup(node* z);
        void transplant(node* u, node* v);
        void delete_fixup(node* x);

    public:
        void init(void);
        void traverse(void);
        node* minimum(node* x);
        node* search(int low, int high);
        void insert(int low, int high);
        node* delete_node(node* z);
        node* random_node(void);
};
```

### 2、初始化

初始化只需要设置 nil 结点，并将 root 置为 nil 即可：

```
void IntervalTree::init(void) {
    nil = new node;
    nil->color = BLACK;
    nil->p = nil->left = nil->right = nil;
    nil->low = nil->high = nil->m = -1;
    root = nil;
}
```

此后的输出当输出-1 时即代表不存在(本次实验中正常的数据均为非负整数)。

**3、左/右旋**

利用书上伪代码实现即可，如左旋：

```
node* y = x->right;
x->right = y->left;
if (y->left != nil) y->left->p = x;
y->p = x->p;
if (x->p == nil) root = y;
else if (x == x->p->left) x->p->left = y;
else x->p->right = y;
y->left = x;
x->p = y;
```

**4、插入**

先完成正常插入的第一部分：

```
node* z = new node;
z->low = low;
z->high = z->m = high;
node* y = nil;
node* x = root;
while (x != nil) {
    y = x;
    if (z->low < x->low) x = x->left;
    else x = x->right;
}
z->p = y;
if (y == nil) root = z;
else if (z->low < y->low) y->left = z;
else y->right = z;
z->left = z->right = nil;
z->color = RED;
```

再利用书上伪代码进行对应的 `fixup` 即可。

**5、删除**

删除需要先实现替换操作与找最小值操作：

```
void IntervalTree::transplant(node* u, node* v) {
    if (u->p == nil) root = v;
    else if (u == u->p->left) u->p->left = v;
    else u->p->right = v;
    v->p = u->p;
}
```

```
node* IntervalTree::minimum(node* x) {
    while (x->left != nil) x = x->left;
    return x;
}
```

接着利用书上伪代码操作进行删除与对应的 `fixup` 即可。

## 区间树：

### 1、插入时的调整

利用 14 章的定理可以发现，实际需要在插入、删除与左右旋中进行调整，而 fixup 的调整是被左右旋自动完成的。

根据左右旋的性质，事实上只有被旋转的结点需要重新计算，左右旋的代码相同：

```
y->m = x->m;
x->m = max(max(x->left->m, x->right->m), x->high);
```

其中是以 x 为中心左旋，y 代表旋转后在 x 位置的结点。

这里 x 与 y 都不可能是 nil，而 nil 的 m 属性固定为-1，比其他都小，因此可以直接计算。

插入只需要从插入结点向上维护：

```
for (node* t = z->p; t != nil; t = t->p)
    high = t->m = max(t->m, high);
```

这里 z 即为插入结点的位置，插入时 z 自身的 m 已经设为 high，于是对其上进行计算。

### 2、删除时的调整

除了 fixup 中的左右旋外，删除时的调整较为复杂：

```
node* t = x;
t->m = -1;
do {
    t->m = max(max(t->left->m, t->right->m), t->high);
    t = t->p;
} while (t != nil);
```

删除时 x 记录了需要维护的位置，但是其可能为 nil 结点，且改变了 m 的值，因此必须将 m 置为-1(即忽略其当前 m)后逐级向上调整。

### 3、查找

直接利用书上伪代码实现即可：

```
node* IntervalTree::search(int low, int high) {
    node* x = root;
    while (x != nil && (low > x->high || high < x->low))
        if (x->left != nil && x->left->m >= low)
            x = x->left;
        else x = x->right;
    return x;
}
```

### 4、遍历

如下图，利用中序序列逐级打出全部区间：

```cpp
void IntervalTree::print_tree(node* now, int depth) {
    if (now == nil) return;
    print_tree(now->left, depth + 1);
    for (int i = 0; i < depth; i++) cout << "---";
    cout << (now->color ? 'B' : 'R') << " [" << now->low << ", " << now->high << ']';
    cout << ' ' << now->m << endl;
    print_tree(now->right, depth + 1);
}

void IntervalTree::traverse(void) {
    print_tree(root, 0);
    cout << endl;
}
```

## 主程序与结果：

### 1、 随机结点

按照题目要求，需要实现随机寻找结点，此处采取随机游走策略：

```cpp
node* IntervalTree::random_node(void) {
    node* x = root;
    while (1) {
        if (x->left == x->right) return x
        int way = rand() % 3;
        if (x->left == nil) {
            if (way < 2) return x;
            x = x->right;
        }
        else if (x->right == nil) {
            if (way < 2) return x;
            x = x->left;
        }
        else {
            if (way == 0) return x;
            if (way == 1) return x->left;
            return x->right;
        }
    }
}
```

从根节点开始，若无孩子则返回，否则对任何存在的孩子有 1/3 概率前往，剩余概率是直接返回。

### 2、随机区间

除了随机寻找结点以外，还要求找到符合要求的随机区间，这里做法如下：

```cpp
ofstream infile;
infile.open(path, ios::out);
int a[50] = {0}, count = 0;
srand(time(0));
```

首先，随机种子并且建立初始全为 0 的 int 数组，和用于统计个数的 count 变量。

```
while (1) {
    int temp_start = rand() % 50;
    if (temp_start >= 24 && temp_start <= 30) continue;
    if (a[temp_start] == 1) continue;
    a[temp_start] = 1;
    infile << temp_start << ' ';
    if (temp_start < 24) {
        int end = temp_start + 1 + rand() % (24 - temp_start);
        infile << end << endl;
    }
    else {
        int end = temp_start + 1 + rand() % (50 - temp_start);
        infile << end << endl;
    }
    if (++count == 30) break;
}
```

接着就是计算过程，将 a 的位置作为查找表确定起点是否用过、是否符合要求，接着在范围内随机终点。

**3、结果展示**

下方为建立红黑树后的输出，可发现黑高度为 3，符合要求，更多结果见 output 文件夹：

```
------------R [0, 12] 12
---------B [2, 20] 20
------R [3, 17] 24
---------B [4, 24] 24
---B [6, 16] 24
------------R [10, 20] 20
---------B [10, 20] 21
------------R [11, 21] 21
------R [12, 24] 24
------------R [13, 16] 16
---------B [14, 22] 22
B [15, 17] 50
---------------R [16, 19] 19
------------B [17, 21] 21
---------R [18, 24] 24
------------B [19, 22] 22
------B [20, 24] 48
---------------R [22, 24] 24
------------B [31, 39] 39
---------R [33, 44] 48
------------B [34, 48] 48
---R [35, 48] 50
---------------R [36, 44] 44
------------B [37, 47] 47
---------R [38, 45] 50
------------B [39, 40] 50
---------------R [41, 50] 50
------B [43, 48] 50
------------B [45, 48] 48
---------R [46, 47] 50
------------B [47, 50] 50
```

# Lab 4 实验报告

PB20000296 郑滕飞

## 基本框架：

### 1、输入生成

如下图，按照顶点数与边数生成即可，这里假设不存在自环，边不会指向自己。

```
void gen_input(int vnum, int edgenum, const char* path) {
    ofstream outfile;
    outfile.open(path, ios::out);
    for (int i = 0; i < vnum; i++)
        for (int j = 0; j < edgenum; j++) {
            outfile << i << ' ';
            int r = rand() % vnum;
            while (r == i) r = rand() % vnum;
            outfile << r << ' ';
            outfile << rand() % 61 - 10 << endl;
    }
    cout << '.';
}
```

```
gen_input(27, 2, "input/input11.txt");
gen_input(27, 1, "input/input12.txt");
gen_input(81, 2, "input/input21.txt");
gen_input(81, 2, "input/input22.txt");
gen_input(243, 3, "input/input31.txt");
gen_input(243, 2, "input/input32.txt");
gen_input(729, 4, "input/input41.txt");
gen_input(729, 3, "input/input42.txt");
cout << "generate done" << endl << endl;
```

### 2、实现算法

```
class Graph {
    private:
        int edge_num;
        vector<node*> vertex;
        void add_edge(int u, int v, int length); // 添加边
        int delete_edge(int u, int v); // 删除边
        int get_edge_length(int u, int v); // 获取边的长度
        int find_negative_circle(void); // 寻找负环
        vector<int> get_renew_h(void); // 计算更新权值的h
        void dijkstra(int v, vector<int> h); // 对带h更新的权值使用Dijsktra算法

    public:
        void init(int vnum); // 初始化顶点
        int get_vertex_num(void) { return vertex.size(); }; // 顶点数
        int get_edge_num(void) { return edge_num; }; // 边数
        void read_file(const char* path); // 读取文件并生成图
        void johnson(const char* path); // Johnson算法
};
```

上图为自建的 Graph 类中实现的函数。由此可以进行算法测试：

```
void test(int vnum, const char* in_file, const char* out_file) {
    Graph G;
    time_file << vnum << '\t';
    cout << "input vertex num: " << vnum << endl;
    G.init(vnum);
    G.read_file(in_file);
    cout << "negative circle cleaned" << endl;
    cout << "edge num: " << G.get_edge_num() << endl;
    time_file << G.get_edge_num() << " \t";
    QueryPerformanceCounter(&start);
    G.johnson(out_file);
    QueryPerformanceCounter(&ende);
    time_file << ende.QuadPart - start.QuadPart << endl;
    cout << "time: " << ende.QuadPart - start.QuadPart << endl;
    cout << endl;
}
```

测试过程中，按顶点数进行初始化，然后读取文件进行图的初始化。在清理负边之后，运行 Johnsson 算法，输出结果并计时。

## 图的表示：

### 1、初始化

```
void Graph::init(int vnum) {
    vector<node*> v(vnum);
    vertex = v;
    for (int i = 0; i < vnum; i++) {
        vertex[i] = new node;
        vertex[i]->serial = -1;
        vertex[i]->edge_length = 0;
        vertex[i]->next = nullptr;
    }
    edge_num = 0;
}
```

利用邻接链表表示图，链表中每个结点有对应边的另一个顶点与边长两个数据域。对于表示头的第一个结点，序号为-1，长度初始化为 0，不过之后的过程事实上不会用到。

### 2、边的基本操作

关于边的操作都是基本的链表操作，如删除：

```
int Graph::delete_edge(int u, int v) {
    node* now = vertex[u];
    node* pre = now;
    while (now->next != nullptr) {
        now = now->next;
        if (now->serial == v) {
            pre->next = now->next;
            delete now;
            edge_num--;
            return 0;
        }
        pre = pre->next;
    }
    return 1;
}
```

注意过程中需要对应维护边的数量。

**3、读取文件**

读取文件事实上是从文件中读取边并加入，若有负环则删除：

```
while (in_file >> u >> v >> w) {
    add_edge(u, v, w);
    if (find_negative_circle()) {
        delete_edge(u ,v);
        cout << '(' << u << ',' << v << "); ";
    }
}
```

直到读取不成功后停止。

## Bellman-Ford：

Bellman-Ford 算法有两处使用，也即寻找负权环与构造 h。注意到，添加一个源点到其他所有点有权为 0 的边事实上就是 d[i]全部初始化为 0 后对应更新，且寻找负环在添加源点后更容易进行，即可写出算法：

```
int Graph::find_negative_circle(void) {
    const int v = get_vertex_num();
    vector<int> d(v);
    for (int i = 0; i < v - 1; i++) for (int j = 0; j < v; j++) {
        node* now = vertex[j];
        while (now->next != nullptr) {
            now = now->next;
            if (d[now->serial] > d[j] + now->edge_length)
                d[now->serial] = d[j] + now->edge_length;
        }
    }
    for (int j = 0; j < v; j++) {
        node* now = vertex[j];
        while (now->next != nullptr) {
            now = now->next;
            if (d[now->serial] > d[j] + now->edge_length)
                return 1;
        }
    }
    return 0;
}
```

只要循环 v-1 次后还能进一步松弛，就代表存在负环，删除这时加入的边就删除了负环。

重复进行这个函数，直到返回值为 0，即删掉了所有负环，而构造 h 的函数即是此函数的上半部分，最终的 d 就是 h。

## Dijkstra：

**1、优先队列**

为了实现 Dijkstra 算法，需要先实现优先队列。这里的优先队列较为复杂，因为需要

同时维护顶点序号对应堆中位置与堆中位置对应顶点序号两个互为逆置换的置换：

```
class NumHeap {
    private:
        vector<int> d;
        vector<int> heap;
        vector<int> pos;
        int heap_size;

    public:
        void init(int n);
        int extract_min(void);
        void decrease(int serial, int after);
        int get_heap_size(void) { return heap_size; };
        int get_d(int v) { return d[v]; };
        int get_pos(int v) { return pos[v]; };
};
```

其中，heap 存的是堆中序号下的顶点序号，pos 为顶点序号对应的堆中序号，去掉最小值的过程即为(依据 d 排序，每次交换同时维护两个映射)：

```
int NumHeap::extract_min(void) {
    int ret = heap[0];
    swap(pos[heap[0]], pos[heap[--heap_size]]);
    swap(heap[0], heap[heap_size]);
    int now = 0, mini = 0;
    while (1) {
        int l = 2 * now + 1, r = 2 * now + 1;
        if (l < heap_size && d[heap[l]] < d[heap[mini]]) mini = l;
        if (r < heap_size && d[heap[r]] < d[heap[mini]]) mini = r;
        if (mini == now) return ret;
        swap(pos[heap[now]], pos[heap[mini]]);
        swap(heap[now], heap[mini]);
        now = mini;
    }
    return ret;
}
```

减值同理：

```
void NumHeap::decrease(int serial, int after) {
    d[heap[serial]] = after;
    while (1) {
        int pre = (serial - 1) / 2;
        if (d[heap[pre]] <= d[heap[serial]]) return;
        swap(pos[heap[pre]], pos[heap[serial]]);
        swap(heap[pre], heap[serial]);
        serial = pre;
    }
}
```

**2、主算法**

这里的 Dijkstra 主算法要求对某个 v 打印所有的最短路径与路径长度，具体过程利用上方的优先队列操作进行，先进行计算，并维护 pi：

```
vector<int> pi(vnum);
for (int i = 0; i < vnum; i++) pi[i] = -1;

NumHeap H;
H.init(vnum);
H.decrease(H.get_pos(v), 0);
while (H.get_heap_size() > 0) {
    int u = H.extract_min();
    node* now = vertex[u];
    while (now->next != nullptr) {
        now = now->next;
        int w = now->serial;
        int new_length = H.get_d(u) + now->edge_length + h[u] - h[w];
        if (H.get_d(w) > new_length) {
            H.decrease(H.get_pos(w), new_length);
            pi[w] = u;
        }
    }
}
```

再进行打印，打印时注意实际的最短路径长度与上方算出的 d 存在差值：

```
for (int i = 0; i < vnum; i++) {
    cout << v << '-' << i << ": ";
    if (pi[i] == -1) cout << "none" << endl;
    else {
        int j = i;
        while (pi[j] != -1) {
            cout << j << "<-";
            j = pi[j];
        }
        cout << j << ' ';
        cout << H.get_d(i) - h[v] + h[i] << endl;
    }
}
```

### 3、整合

最终版的 Johnson 算法如下：

```
void Graph::johnson(const char* path) {
    vector<int> h = get_renew_h();
    FILE* stream;
    freopen_s(&stream, path, "w", stdout);
    for (int i = 0; i < get_vertex_num(); i++)
        dijkstra(i, h);
    fclose(stdout);
    freopen_s(&stream, "CON", "w", stdout);
}
```

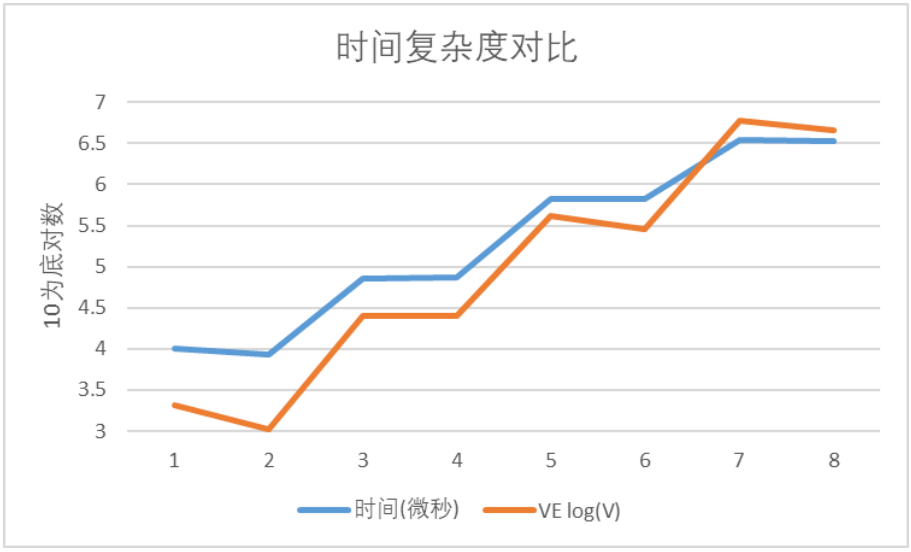
计算出 h 后，对每个顶点调用 Dijkstra 算法，并将输出定向到对应的文件中。

## 结果展示：

具体的输入与输出见 input 与 output 文件夹中的文件，控制台输出输入的顶点数、边数(删除负环前)、删除的负环边与时间，删除后的顶点数、边数与时间在 time.txt 中，以下为控制台输出示例：

```
input vertex num: 81
negative circle cleaned
edge num: 162
time: 732071

input vertex num: 243
(201,162); (218,8); (219,189); (232,149); (235,161); negative circle cleaned
edge num: 724
time: 6736036

input vertex num: 243
negative circle cleaned
edge num: 486
time: 6616400

input vertex num: 729
(440,36); (444,395); (528,369); (537,307); (552,58); (591,412); (616,142); (616,387); (622,310); (625,536); (645,243); (
650,140); (653,260); (655,416); (662,319); (665,241); (665,283); (667,536); (669,64); (678,437); (679,18); (683,645); (6
93,516); (706,453); (708,394); (710,642); (719,19); (720,197); (720,119); (721,481); (721,45); (726,487); (726,395); (72
7,6); negative circle cleaned
edge num: 2882
time: 35283529

input vertex num: 729
(545,34); (685,190); (706,199); (728,528); negative circle cleaned
edge num: 2183
time: 33863739
```

根据 time.txt 绘制出的时间复杂度图如下：




由于规模差异，统一采取对数坐标轴进行绘制，可以发现实际复杂度与理论复杂度是接近的，这里下方的 1 到 8 分别代表 11、12、21、22、31、32、41、42。

**总结：**

恭喜算法实验完结！【写完一个月后在提交 ddl 那天发现 bug 吓死】