数据结构实践报告

09019106 牟倪

Ⅰ 模拟矩阵相乘

1.1 问题描述

模拟矩阵相乘，实现最初版本的存储管理器。

1.2 系统结构

- 为矩阵编写一个类。
- 用文件模拟磁盘，将矩阵元素映射到文件内。在文件首部放置行首指针的offset，之后对元素进行顺序存储。将文件存储在类里。
- 为每个文件维护一个缓存，缓存的宽度为w。每次去文件读矩阵元素时，都把该元素以及其后的w-1个元素放到缓存中。
- 在类里维护上次读文件时所读数据的坐标 (prex, prey)。接下来读数据 (x, y) 时，首先考察是否满足 prex=x 且 0<=y-prey<w（即该数据在上次缓存范围中），如果满足则直接读缓存，如果不满足则读文件、更新缓存相关信息。
- 写数据的情形和读数据类似。

1.3 功能模块设计

矩阵类 `Matrix`：

- 成员变量：
  - `disk`，是一个文件。
  - `cache`，数据类型为int指针。
  - `w`，cache长度。
  - `prex, prey`，上次读文件时所读数据的坐标。
  - `miss`，读文件次数。
- 成员函数：
  - 构造函数 `Matrix(string name, int n, int w, int* a)`：
    - 创建一个名为 name 的文件，`disk=fstream(name,ios::in|ios::out|ios::binary)`。
    - 将n*n的数组a作为矩阵数据，写入文件：
      - 我们期望，在读行 x 列 y 的数值时，可以先通过直接索引 x 找到行 x 的地址 offset，然后通过 offset+y 找到该数据。
      - 因此，首先写入n个行指针的offset，第x行的offset为(x+1)*n。
      - 接下来，依序写入矩阵数据。
    - 初始化cache长度，`this.w=w`。
    - 初始化缓存，`cache=new int[w]`。
    - 初始化读文件次数，`miss=0`。
    - 简单起见，直接`prex=prey=0`，读数据a[0][0]、把数据放到cache里、miss++。
  - 析构函数 `~Matrix()`：
    - 关闭文件、delete缓存空间。
  - 读数据 `int read(int x, int y)`：
    - 参数为行 x、列 y。
    - 我们考察：是否满足`prex==x && 0<=y-prey<w`；若满足，则说明数据在缓存中，直接返回`cache[y-prey]`。

        - 若不满足，则首先读文件，读到指向行 x 的指针offset，然后读 offset+y，读取数据；并读取其后的w-1个数值，存入cache里，更新`prex=x, prey=y`，`++miss`。
    - 写数据`void write(int x, int y, int value)`：
        - 参数为行 x、列 y、希望写入的值 value。
        - 我们考察：是否满足`prex==x && 0<=y-prey<w`；若满足，则说明数据在缓存中，直接修改`cache[y-prey]`。
        - 若不满足，则：
            - 首先将cache的数据写回文件；
            - 然后读文件，读到指向行 x 的指针offset，然后读 offset+y，读取数据；并读取其后的w-1个数值，存入cache里，更新`prex=x, prey=y`，`++miss`；
            - 最后在cache里修改数据。

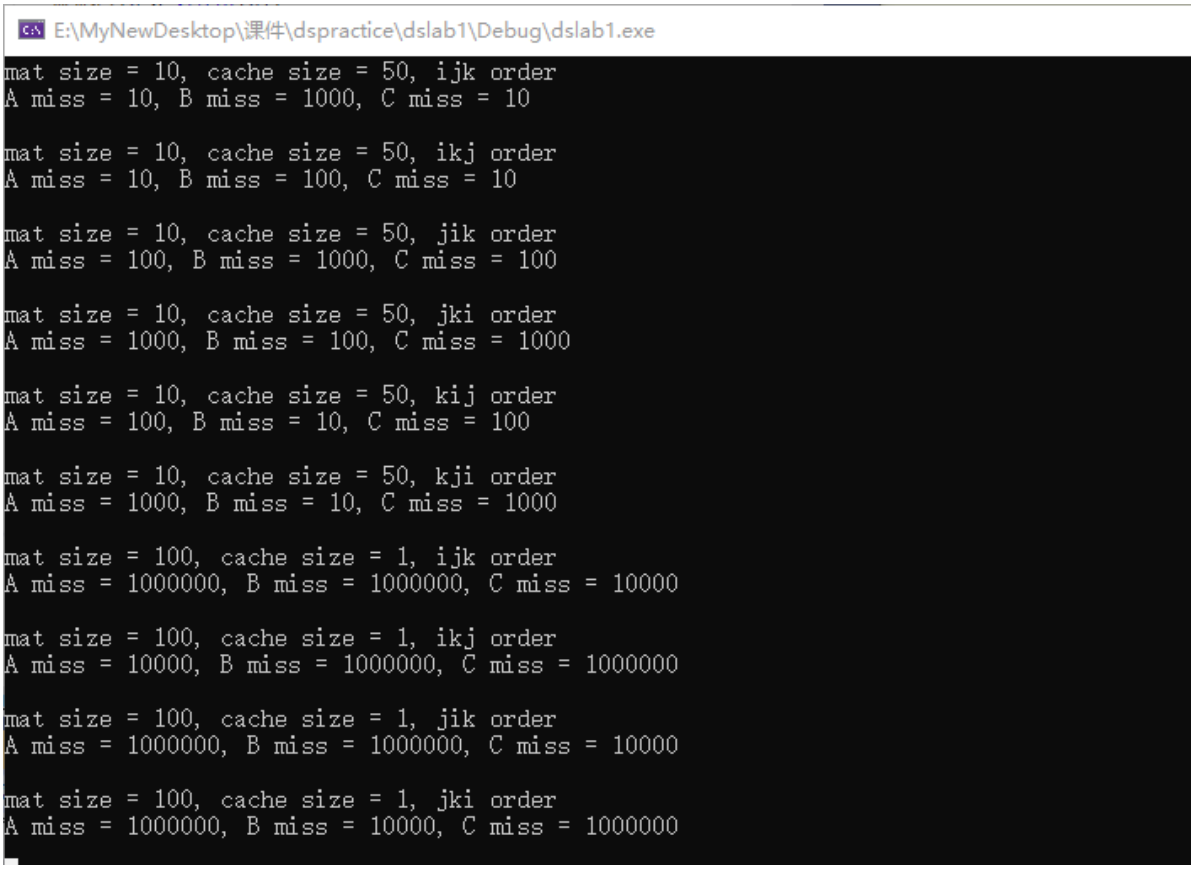
1.4 测试结果与分析

1.4.1 实验数据

测试数据生成 & 实验流程：

- 将大小为 10*10、100*100、1000*1000、10000*10000 的单位矩阵相乘。
- 同时，cache大小w分别设为 1、5、10、50。
- 同时，分别采用 ijk, ikj, jik, jki, kij, kji 6种循环顺序。
- 分别记录两个乘数矩阵与一个结果矩阵的cache miss次数。

控制台结果截图（部分）：



运行结果表格（部分）：

	order	ijk		ikj		jik		jki		kij		kji	
cache_size	mat_size	10	100	10	100	10	100	10	100	10	100	10	100
1	A_miss	1e3	1e6	1e2	1e4	1e3	1e6	1e3	1e6	1e3	1e4	1e3	1e6
	B_miss	1e3	1e6	1e3	1e6	1e3	1e6	1e2	1e4	1e2	1e6	1e2	1e4
	C_miss	1e2	1e4	1e3	1e6	1e2	1e4	1e3	1e6	1e3	1e6	1e3	1e6
50	A_miss	1e1	2e4	1e1	2e2	1e2	2e4	1e3	1e6	1e2	1e4	1e3	1e6
	B_miss	1e3	1e6	1e2	2e4	1e3	1e6	1e2	1e4	1e1	2e4	1e1	2e2
	C_miss	1e1	2e2	1e1	2e4	1e2	1e5	1e3	1e6	1e2	2e4	1e3	1e6

1.4.2 实验结论

观察实验数据，可以得到以下结论：（记n为矩阵尺寸，w为cache尺寸）

- 对ijk的遍历顺序，A miss=n^3/w，B miss=n^3，C miss=n^2/w。
- 对ikj的遍历顺序，A miss=n^3/w，B miss=n^3/w，C miss=n^2/w。
- 对jik的遍历顺序，A miss=n^3/w，B miss=n^3，C miss=n^2。
- 对jki的遍历顺序，A miss=n^3，B miss=n^3，C miss=n^2。
- 对kij的遍历顺序，A miss=n^3/w，B miss=n^3/w，C miss=n^2/w。
- 对kji的遍历顺序，A miss=n^3/w，B miss=n^3/w，C miss=n^2。

进而可以得到以下结论：

- 如果i是k的外层循环，A miss=n^3/w，否则A miss=n^3。
- 如果k是j的外层循环，B miss=n^3/w，否则B miss=n^3。
- 如果i是j的外层循环，C miss=n^2/w，否则C miss=n^2。

1.5 实验总结

经过本次实验，我熟悉了C++语言的使用，为接下来的实验做铺垫。

II 用快速排序实现外排序

2.1 问题描述

使用快速排序，实现外部排序。

2.2 系统结构

- 将内存划分为input，small，large，middle group四块，我们使用四个数组来模拟。
- 每当input为空时，从磁盘中读入数据充满input区（使用input缓存，减少磁盘读写次数）。
- 然后，从input读入数据，充满middle区。在middle区维护一个双端优先队列（这里直接使用 C++ STL 的 multiset，基于红黑树，支持自动排序）。
- 接下来，
  - 当新读到的数据小于middle区最小值时，将数据写入small区。
  - 当新读到的数据大于middle区最大值时，将数据写入large区。
  - 否则，将middle区最小值弹出，写入small区，将新数据写入middle区。
- 当small和large区满的时候，写到新文件，文件需要进行递归编号。（此时，只能保证【small里所有元素 < middle里所有元素 < large里所有元素】，但small区和large区的元素是无序的，需要继续进行递归的排序）
- middle区的数据已经排好序了（双端优先队列），将middle区数据按照排好的顺序写回文件。
- 然后，对small区数据写入的文件、large区数据写入的文件，进行递归的外排序。
- 把所有文件合并，整合得到最终的答案文件。

2.3 功能模块设计

- 文件操作：
  - 在设计读出写入操作的函数中，`++disk_count`。
  - 实现函数`vector<int> readFile(string name, int offset, int n)`，读取数据。
  - 实现函数`vector<int> readFileAll(string name)`，读取文件所有数据。
  - 实现函数`void writeFile(string name, int offset, vector<int>data)`，向文件写入数据。
  - 实现函数`void newSpace(string name, int n)`，新建一个名为name的文件，开辟存放n个数据的空间。
- 外排序：
  - 实现函数`void externalSort(string name, int l, int r, int num)`。
  - 参数：文件名`name`，排序起始位置`l`，排序终止位置`r`，编号`num`。
  - 功能：对name文件中，从第 l 个数据开始到第 r 个数据进行外排序。
- 衡量外排序性能：维护全局变量`disk_count`，记录文件I/O次数。

2.4 测试结果与分析

2.4.1 实验数据

测试数据生成 & 实验流程：

- 随机生成1000以内的随机整数，作为需要外排序的实验数据。
- 同时，数据量分别设为 10、100、1000、10000、100000。
- 同时，分别采用不同的input区、small区、large区、middle区大小。
- 分别记录文件I/O次数`disk_count`。

运行结果截图（toy case）：

```
original data:
41 467 334 500 169 724 478 358 962 464 705 145 281 827 961 491 995 942 827 436

sorted data:
41 145 169 281 334 358 436 464 467 478 491 500 705 724 827 827 942 961 962 995

disk_count=9
```

运行结果表格（部分）：

data_scale	input_scale	small_scale	large_scale	middle_scale	disk_count			avg
1e1	1e2	1e2	1e2	1e3	8	8	8	8.00
1e2	1e2	1e2	1e2	1e3	8	8	8	8.00
1e3	1e2	1e2	1e2	1e3	17	17	17	17.00
1e4	1e2	1e2	1e2	1e3	731	725	725	727.00
1e5	1e2	1e2	1e2	1e3	39977	39943	39977	39965.67

data_scale	input_scale	small_scale	large_scale	middle_scale	disk_count			avg
1e4	1e2	1e2	1e2	1e3	731	731	731	731.00
1e4	5e1	1e2	1e2	1e3	1086	1077	1092	1085.00
1e4	1e2	5e1	1e2	1e3	950	932	941	941.00
1e4	1e2	1e2	5e1	1e3	784	785	784	784.33

第一个表格是数据量与访问磁盘数量的关系；

第二个表格是改变input区大小、small区大小、large区大小后，磁盘I/O次数的变化。

2.4.2 实验结论

- 当数据量小于内存容量时，可以直接用内排序完成排序，此时磁盘I/O次数为常数。
- 总磁盘I/O次数随数据量增加而增加，且增加速率较快，应该不是线性增长。
- 减小input区大小，对磁盘I/O次数影响较大。
- 减小small区大小，对磁盘I/O次数有一些影响；减小large区大小，对磁盘I/O次数影响较小。可能是因为，当快排读入新数据的大小在middle区范围内时，我们会将middle区最小值弹出写入small区，将新数据写入middle区，因此small区的读写次数比large区更多。
- 总结：虽然快排对于内排序的性能很好，但是对于外排序的性能不太理想。

2.5 实验总结

经过本次实验，我进一步熟悉了C++语言的文件处理，了解到递归是很有力的思想。

Ⅲ 用归并排序实现外排序

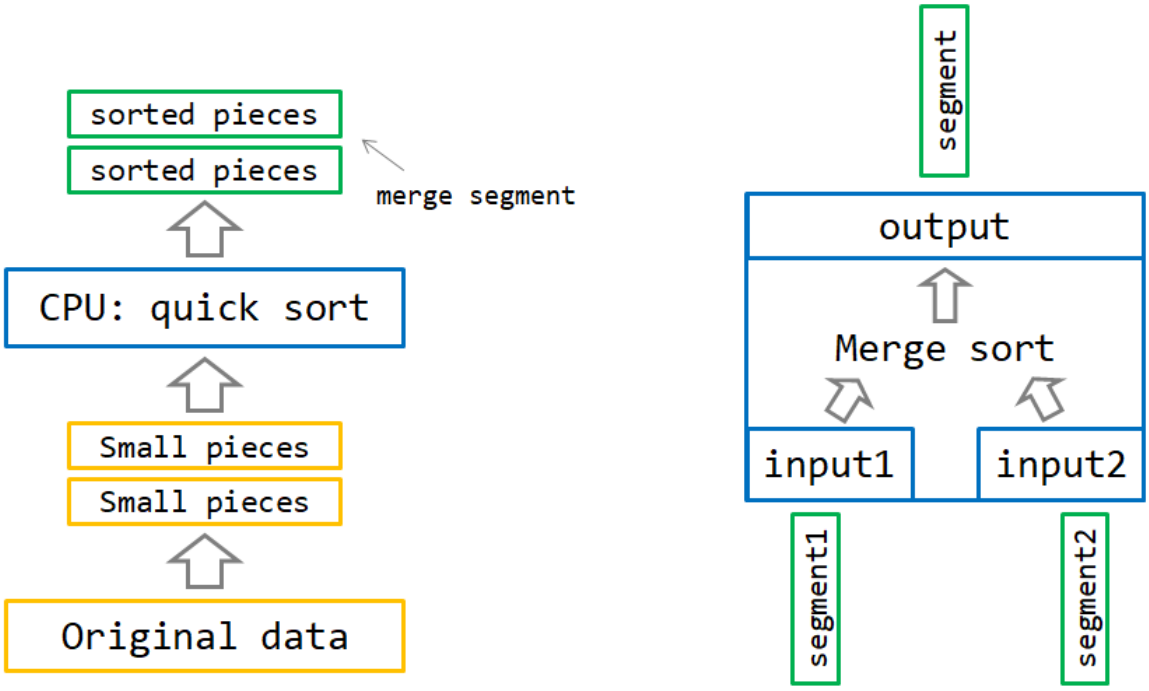
3.1 问题描述

使用二路归并排序，实现外部排序。

3.2 系统结构

- 生成初始归并段：
  - 每次从磁盘读入数据，直到充满内存区域，然后使用内排序方法（如快速排序）进行排序。
  - 将排序后的数据写入新文件，从而产生一系列初始归并段，并保存归并段文件名数据（names）。
- 归并：
  - 将内存划分为 input1，input2，output 3块。
  - 对归并段数组（names）进行递归的归并排序，首先对左右两部分进行归并排序（递归调用函数），得到两个有序文件（记为left_file，right_file），然后进行归并。注意，文件仍需递归编号。
  - 每当input1为空时，从left_file中读入数据，充满input1区。每当input2为空时，从right_file中读入数据，充满input2区。
  - 对input1和input2的数据进行归并，归并到output区。
  - 每当output区数据满，将数据写到新文件。

算法的描述图如下：



3.3 功能模块设计

- 文件操作：
  - 在设计读出写入操作的函数中，`++disk_count`。
  - 实现函数`vector<int> readFile(string name, int offset, int n)`，读取数据。
  - 实现函数`vector<int> readFileAll(string name)`，读取文件所有数据。
  - 实现函数`void writeFile(string name, int offset, vector<int>data)`，向文件写入数据。
  - 实现函数`void newSpace(string name, int n)`，新建一个名为name的文件，开辟存放n个数据的空间。
- 生成初始归并段：
  - 实现函数`vector<string> generateRes(string name, int l, int r)`。
  - 参数：原始数据（未经排序）的文件名`name`，排序起始位置`l`，排序终止位置`r`。
  - 功能：对磁盘的数据进行分批的内排序，生成初始归并段，并将初始归并段的文件名以`vector<string>`的形式返回。
- 外排序：
  - 实现函数`string externalSort(vector<string> names, int l, int r, int num, int& outer_count)`；
  - 参数：归并段文件名数组`names`，排序起始位置`l`，排序终止位置`r`，编号`num`。将该区域排序后的数据个数，以引用的方式传给`outer_count`。
  - 功能：对`names`初始归并段数组中，从第`l`个段到第`r`个段的所有数据，进行外部排序。
- 衡量外排序性能：维护全局变量`disk_count`，记录文件I/O次数。

3.4 测试结果与分析

3.4.1 实验数据

测试数据生成 & 实验流程：

- 随机生成5000以内的随机整数，作为需要外排序的实验数据。
- 同时，数据量分别设为 10、100、1000、10000、100000。
- 同时，分别采用不同的input区、output区大小。

- 同时，分别采用不同的内存大小。
- 分别记录文件I/O次数`disk_count`。

运行结果截图（toy case）：

```
original data:
41 467 334 500 169 724 478 358 962 464 705 145 281 827 961 491 995 942 827 436

sorted data:
41 145 169 281 334 358 464 467 478 491 500 705 724 827 827 942 961 962 995 995

disk_count=4
```

运行结果表格（部分）：

data_scale	input_scale	output_scale	mem_scale	disk_count			avg
1e1	1e2	1e2	4e2	3	3	3	3.00
1e2	1e2	1e2	4e2	3	3	3	3.00
1e3	1e2	1e2	4e2	47	47	47	47.00
1e4	1e2	1e2	4e2	995	995	995	995.00
1e5	1e2	1e2	4e2	16453	16453	16453	16453.00

data_scale	input_scale	output_scale	mem_scale	disk_count			avg
1e4	1e2	1e2	4e1	4489	4489	4489	4489.00
1e4	1e2	1e2	4e2	995	995	995	995.00
1e4	1e2	1e2	4e3	407	407	407	407.00

data_scale	input_scale	output_scale	mem_scale	disk_count			avg
1e4	1e2	1e2	4e2	995	995	995	995.00
1e4	2e2	1e2	4e2	759	759	759	759.00
1e4	1e2	2e2	4e2	759	759	759	759.00

3.4.2 实验结论

- 当数据量小于内存容量时，可以直接用内排序完成排序，此时磁盘I/O次数为常数。
- 总磁盘I/O次数随数据量增加而增加，且增加速率较快，应该不是线性增长。
- data_scale=1e5时，二路归并外排序的磁盘I/O次数，明显小于快排（实验二）的磁盘I/O次数。
- 总磁盘I/O次数随内存容量减小而增加，且增加速率较快，应该不是线性增长。
- 减小input区和output大小，磁盘I/O次数增大，且减小input区和output大小的影响是对称的。
- 虽然实验数据是随机数，但每次排序结果都相同，不知道什么原因。可能是因为，随机数范围太小，数列每次的分布都比较相近。
- 总结：二路归并外排序的性能比较理想。

3.5 实验总结

经过本次实验，我进一步熟悉了C++语言的文件处理，了解到递归是很有力的思想，感叹归并排序的思想很精简。

IV 改进归并排序的外排序

4.1 问题描述

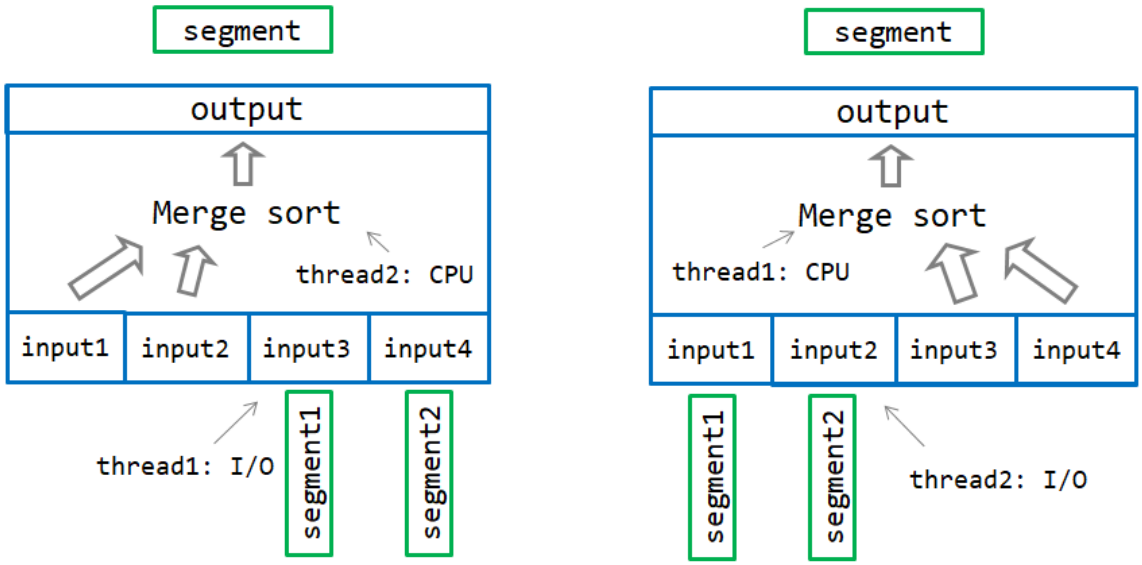
对二路归并的外部排序进行优化：

- 对给定长度的归并段序列，基于Huffman树，构造最佳归并顺序。
- 使用多线程技术，同时读写磁盘、在内存中进行归并。

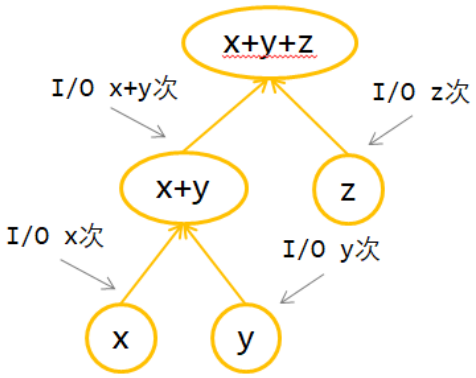
4.2 系统结构

- 生成初始归并段：
  - 根据给定的归并段参数，生成特定个数的50000以内的随机数，然后使用内排序方法（如快速排序）进行排序。
  - 将排序后的数据写入新文件，从而产生一系列初始归并段。
- 构造最佳归并顺序：
  - 根据给定的归并段参数，基于Huffman树进行构造：
  - 维护一个multiset，每次拿走长度最短的两个归并段进行归并，并将新的归并段放入multiset中。
- 归并：
  - 将内存划分为 input1，input2，input3，input4，output 共5块。
  - 根据最佳归并顺序，将文件逐个进行归并。
- 多线程：
  - 首先，我们往input1，input2中读入数据；
  - 接下来，我们同时进行以下两件事：
    - 往input3，input4中读入数据；
    - 对input1，input2进行归并，结果输出到output中，如果output满就写回到新归并段。
  - 再接下来，我们同时进行以下两件事：
    - 往input1，input2中读入数据；
    - 对input3，input4进行归并，结果输出到output中，如果output满就写回到新归并段。
  - 直到两个归并段中，某一个归并段被读完。

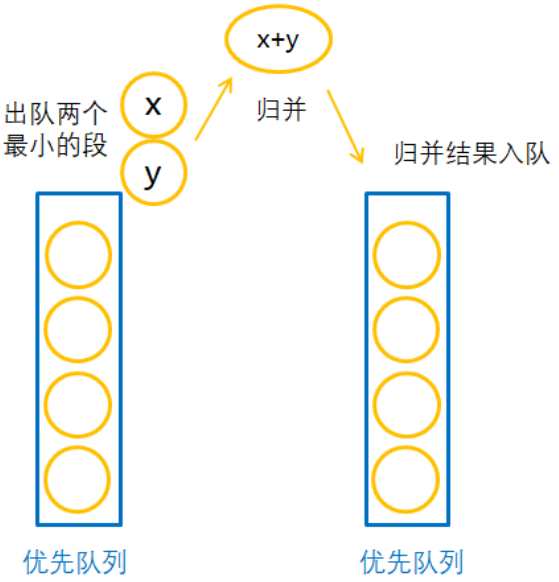
多线程算法的描述图如下：



Huffman树原理的描述图如下：



I/O次数（cost）与段长度成正比，
因此可以应用Huffman树



4.3 功能模块设计

- 文件操作：
  - 在设计读出写入操作的函数中，`++disk_count`。
  - 实现函数`vector<int> readFile(string name, int offset, int n)`，读取数据。
  - 实现函数`vector<int> readFileAll(string name)`，读取文件所有数据。
  - 实现函数`void writeFile(string name, int offset, vector<int>data)`，向文件写入数据。
  - 实现函数`void newSpace(string name, int n)`，新建一个名为name的文件，开辟存放n个数据的空间。
- 生成初始归并段：
  - 用全局变量`vector<pair<int, string>> merge_segments`来记录初始的归并段参数，`int`为长度，`string`为文件名。
  - 实现函数`void generateRes()`，用来生成初始归并段。
- 构造最佳归并顺序：
  - 用全局变量`vector<pair<pair<string, string>, pair<int, int>>> plan`来记录最佳归并顺序。`plan`表示一系列归并动作的`vector`，两个`string`分别是需要被归并的两个段的文件名，两个`int`分别是两个段的数据个数。归并结果的文件名约定为两个文件名的拼接。
  - 实现函数`void generateMergePlan(vector<pair<int, string>> merge_segments)`。
- 外排序：
  - 实现函数`void solve()`，按照最佳归并顺序进行归并。
  - 实现函数`void externalSort(string name1, int count1, string name2, int count2)`；
    - 参数：归并段文件`name1, name2`，相应的归并段大小`count1, count2`。
- 多线程：
  - 实现函数`void readDataIntoMem(string name1, string name2, int cur1, int cur2, bool input34)`。
    - 参数：归并段文件`name1, name2`，光标位置`cur1, cur2`，是否读进 input3 input4 的bool变量`input34`，为false则读进 input1 input2 。
  - 实现函数`void mergeSort(string name, int curout, bool input34)`。

    - 参数：归并后写入的文件`name`，归并后写入文件的光标`curout`，是否归并 input3 input4 的bool变量`input34`，为false则归并 input1 input2 。
- 衡量外排序性能：
    - 维护全局变量`disk_count`，记录文件I/O次数。
    - 对外排序进行计时。

4.4 测试结果与分析

4.4.1 实验数据

测试数据生成 & 实验流程：

- 随机生成50000以内的随机整数，作为需要外排序的实验数据。
- 同时，数据量分别设为 500，5000，50000。
- 分别记录文件I/O次数`disk_count`和归并耗时。

运行结果截图（toy case）：

```
best merge plan:
merge R1.dat 100, R2.dat 200
merge R1R2.dat 300, R3.dat 300
merge R4.dat 400, R5.dat 500
merge R1R2R3.dat 600, R6.dat 600
merge R4R5.dat 900, R1R2R3R6.dat 1200


disk_count=264

Total time=846ms
```

运行结果表格（部分）：

data_scale	disk_count		time (ms)							
	original	inproved	original			avg	inproved			avg
5e2	17	31	35	46	46	42	98	90	89	92
5e3	321	610	886	722	786	798	1876	1778	1662	1772
5e4	5483	6480	14384	16435	17638	16152	16592	17168	16626	16795

4.4.2 实验结论

- 当数据量较小时，多线程程序比单线程慢大概一倍，可能是因为，在多线程实现时，为了减小编程复杂性，书写了较多冗余代码。
- 当数据量达到5e4的时候，多线程和单线程的表现齐平。考虑到冗余代码的问题，我们可以做乐观的假设：当冗余代码被尽量缩减、两个程序在小数据下表现齐平时，多线程程序在大数据量下会比单线程快一倍。我恰好使用了两个线程，因此多线程加速是有效的。
- 总结：多线程进行了有效的加速。

4.5 实验总结

经过本次实验，我进一步熟悉了C++语言的文件处理，了解到递归是很有力的思想，感叹归并排序的思想很精简。