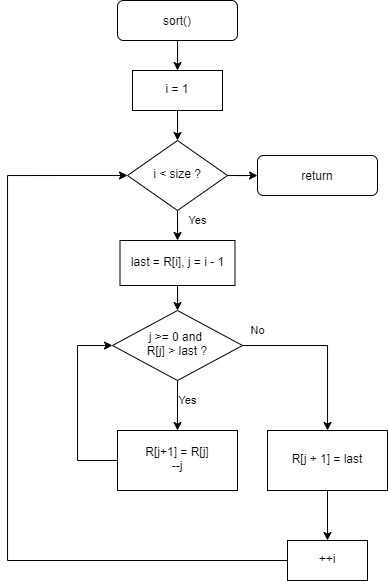
1. **实验要求**

计算插入排序、合并排序、快速排序的腾挪次数，并用黑盒、白盒测试算法的正确性。

1. **插入排序**
2. **算法思想**

以待排数组头部作为已排序序列，将其后的元素依次与相邻的前个元素进行比较，当大于(升序排序)时，则交换两个元素并继续比较，直至当前元素小于等于前个元素，或比较至数组头部。



P2大东

P1

1. **黑盒测试**



经多次不同大小随机乱序数组验证，结果无误。

1. **白盒测试**

采用OpenCppCoverage测试工具检查代码的覆盖率。

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 执行结果 | P1 | P2 |
| [1] | [1] | × | × |
| [1 2] | [1 2] | √ | × |
| [2 1] | [1 2] | √ | √ |

1. **源代码**

void sort() {

for (int i = 1; i < size; ++i)

{

T last = R[i];

int j = i - 1;

while ((j >= 0) && (R[j] > last)) {

R[j + 1] = R[j]; ++times; //腾挪

--j;

}

R[j + 1] = last; //腾挪

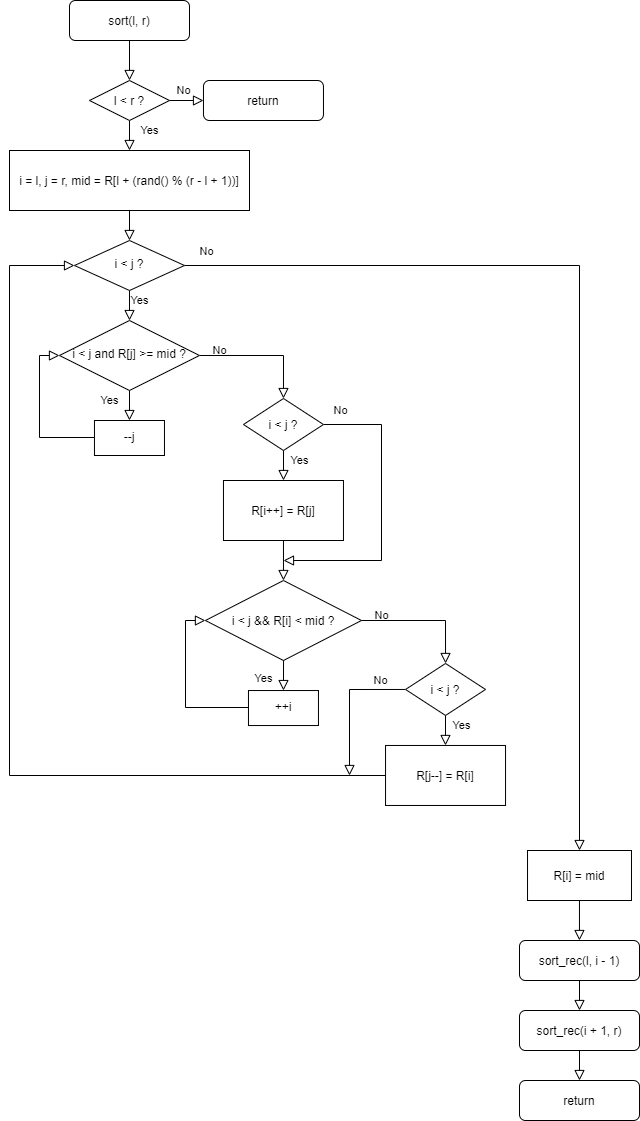
for (int i = 0; i < size; ++i)cout << R[i] << " ";

cout << endl;

}

}

1. **快速排序**
2. **算法思想**



P6大东

P5大东

P4大东

P3大东

P1大东

P2大东

以数组中某一元素值作为枢轴元素，前后两指针同时遍历，循环将左侧比枢轴元素大的值与右侧比枢轴元素小的值进行交换，直至两指针相遇，此时可将原数组分割为两个数组，并继续递归求解至数组大小为0或1。

1. **黑盒测试**



经多次不同大小随机乱序数组验证，结果无误。

1. **白盒测试**

采用OpenCppCoverage测试工具检查代码的覆盖率。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 测试用例 | 执行结果 | P1 | P2 | P3 | P4 | P5 | P6 |
| [3 2 1] | [1 2 3] | √ | √ | √ | × | × | × |
| [2 1 3] | [1 2 3] | √ | √ | √ | √ | × | × |
| [4 3 1 2] | [1 2 3 4] | √ | √ | √ | √ | × | √ |
| [3 5 4 1 2] | [1 2 3 4 5] | √ | √ | √ | √ | √ | √ |

1. **源代码**

void sort\_rec(int l, int r) {

if (l < r) {

int i = l, j = r, mid = R[l + (rand() % (r - l + 1))];

while (i < j) {

while (i < j && R[j] >= mid) {

--j;

}

if (i < j) {

R[i++] = R[j]; ++times; //腾挪

}

while (i < j && R[i] < mid) {

++i;

}

if (i < j) {

R[j--] = R[i]; ++times; //腾挪

}

}

R[i] = mid;

sort\_rec(l, i - 1);

sort\_rec(i + 1, r);

}

}

1. **归并排序(递归)**
2. **算法思想**

将原数组递归拆分至大小为1的数组，两两按序合并，直至重新拼接为原数组。

流程图略。

1. **黑盒测试**



经多次不同大小随机乱序数组验证，结果无误。

1. **白盒测试**

采用OpenCppCoverage测试工具检查代码的覆盖率。

|  |  |  |
| --- | --- | --- |
| 测试用例 | 执行结果 | 覆盖率 |
| [1] | [1] |  |
| [2 1] | [1 2] |  |
| [2 1 3] | [1 2 3] |  |

1. **源代码**

void merge\_rec(int l, int m, int r) {

int i = l, j = m + 1, n = 0, len = r - l;

while (i <= m && j <= r) {

if (R[i] > R[j]) {

tmp[n++] = R[j++]; ++times; //腾挪

num += m - i + 1;

}

else {

tmp[n++] = R[i++]; ++times; //腾挪

}

}

if (i > m) {

while (j <= r) {

tmp[n++] = R[j++]; ++times; //腾挪

}

}

else {

while (i <= m) {

tmp[n++] = R[i++]; ++times; //腾挪

}

}

for (int k = 0; k <= len; ++k) {

R[l + k] = tmp[k]; ++times; //腾挪

}

}

void sort\_rec(int l, int r) {

if (l < r) {

int m = (l + r) / 2;

sort\_rec(l, m);

sort\_rec(m + 1, r);

merge\_rec(l, m, r);

}

}

1. **归并排序(迭代)**
2. **算法思想**

从步长为1开始，通过步长计算得到两个待合并的子区间，并进行合并，后步长加倍，模拟递归版归并排序中的合并顺序继续执行。相比于递归方式，省去了递归所占用的时间与栈空间资源。

流程图略。

1. **黑盒测试**



经多次不同大小随机乱序数组验证，结果无误。

1. **白盒测试**

采用OpenCppCoverage测试工具检查代码的覆盖率。

|  |  |  |
| --- | --- | --- |
| 测试用例 | 执行结果 | 覆盖率 |
| [1 2] | [1 2] |  |
| [3 1 2] | [1 2 3] |  |

1. **源代码**

void sort(int l, int r) {

int l1, r1, l2, r2;

for (int step = 1; step < size; step = step \* 2) {

for (l1 = 0; l1 < size - step; l1 = r2) {

l2 = r1 = l1 + step;

r2 = r1 + step;

if (r2 > size) r2 = size;

int next = 0;

while (l1 < r1 && l2 < r2) {

if (R[l1] < R[l2]) {

tmp[next++] = R[l1++]; ++times; //腾挪

}

else {

tmp[next++] = R[l2++]; ++times; //腾挪

}

}

while (l1 < r1) {

R[--l2] = R[--r1]; ++times; //腾挪

}

while (next > 0) {

R[--l2] = tmp[--next]; ++times; //腾挪

}

}

}

}

1. **总结**

本次实验比较了三种(四种)排序算法，通过构造流程图、黑盒与白盒测试，加深了我对这几种排序方法的理解和感悟。

插入排序由于其算法在理论上时间复杂度上的劣势为O(n^2)，使得其腾挪次数与比较次数都极其不理想。

快速排序，渐进时间复杂度为O(nlogn)。如果采用随机选择策略，其时间复杂度在渐进意义上可以较优，腾挪次数相对归并排序少，但比较次数相对归并排序较多。但是快速排序不存在迭代方式，当数组过大，递归层次过深时不易使用。这在STL提供的sort()中也可以体现，当递归层次过深时会改用堆排序。

归并排序最大的特点是稳定性，时间复杂度为O(nlogn)，其不受元素具体内容的限制，但需要额外的辅助空间，比较次数明显少于快速排序，腾挪次数却明显多于快速排序。可以改造为迭代方式，相较于快速排序，对栈空间的需求更低。

常见的基于比较的排序算法除了以上三种还有选择排序、希尔排序、堆排序等。

选择排序时间复杂度同插入排序为O(n^2)，但选择排序的腾挪次数为O(n)，例如对于重写效率受限的设备较为友好。

希尔排序则改进了插入排序，在插入排序的基础上通过分割出子序列从而增加比较间隔的方式减少了临近元素连续比较的次数，因而效率会优于插入排序，同时这样的算法形式容易改造为多线程，可以进一步提升效率。

堆排序则是在数据结构的基础上，通过构造二叉堆的方式使得查找插入位置的时间复杂度达到O(logn)，因而算法复杂度能够到达O(nlogn)，但应对对于较为有序的数组时退化严重至O(n^2)，更适用于随机的大数组排序。