

**程序设计实践实验报告**

**实 验（三）**

题 目 排序算法实现与性能比较

专 业 计算机科学与技术

学　　 号 2021113679

班　　 级 2103101

学 生 吴嘉阳

指 导 教 师 吴锐

实 验 地 点 管理楼 G709

实 验 日 期 2022.8.29

**计算学部**

**目 录**

[第1章 实验基本信息 - 2 -](#_Toc112351242)

[1.1 实验目的 - 2 -](#_Toc112351243)

[1.2 实验环境与工具 - 2 -](#_Toc112351244)

[1.2.1 硬件环境 - 2 -](#_Toc112351245)

[1.2.2 软件环境 - 2 -](#_Toc112351246)

[1.2.3 开发工具 - 2 -](#_Toc112351247)

[第2章 数据生成和性能测试 - 3 -](#_Toc112351248)

[3.1 合理调用rands(),rand()生成数据（2分） - 3 -](#_Toc112351249)

[3.2 合理保障输入各个排序算法的数据一致性（2分） - 3 -](#_Toc112351250)

[3.3 获取、计算、输出各个排序算法的运行时间（2分） - 3 -](#_Toc112351251)

[第3章 排序算法 - 4 -](#_Toc112351252)

[2.1 平方级排序算法（6分） - 4 -](#_Toc112351253)

[2.1.1冒泡排序算法（2分） - 4 -](#_Toc112351254)

[2.1.2选择排序算法（2分） - 4 -](#_Toc112351255)

[2.1.2插入排序算法（2分） - 4 -](#_Toc112351256)

[2.2 nlogn排序算法（14分） - 4 -](#_Toc112351257)

[2.2.1归并排序算法（7分） - 4 -](#_Toc112351258)

[2.2.2快速排序算法（7分） - 4 -](#_Toc112351259)

[2.3 算法性能比较（3分） - 5 -](#_Toc112351260)

[2.3.1数量级10000上的性能比较（1分） - 5 -](#_Toc112351261)

[2.3.2数量级100000上的性能比较（1分） - 5 -](#_Toc112351262)

[2.3.3数量级1000000上的性能比较（1分） - 5 -](#_Toc112351263)

[第3章 堆排序与扩展研究 - 6 -](#_Toc112351264)

[3.1 堆排序算法（5分） - 6 -](#_Toc112351265)

[3.2扩展研究（5分） - 6 -](#_Toc112351266)

# 第1章 实验基本信息

## 1.1 实验目的

* 利用Makefile管理编译过程
* 熟练掌握典型排序算法
* 熟练掌握Linux下程序的调试
* 理解算法对性能的影响

## 1.2 实验环境与工具

### 1.2.1 硬件环境

 Intel 10750H x86\_64

### 1.2.2 软件环境

 WSL2 Ubuntu20.04 64位

### 1.2.3 开发工具

Gcc vim

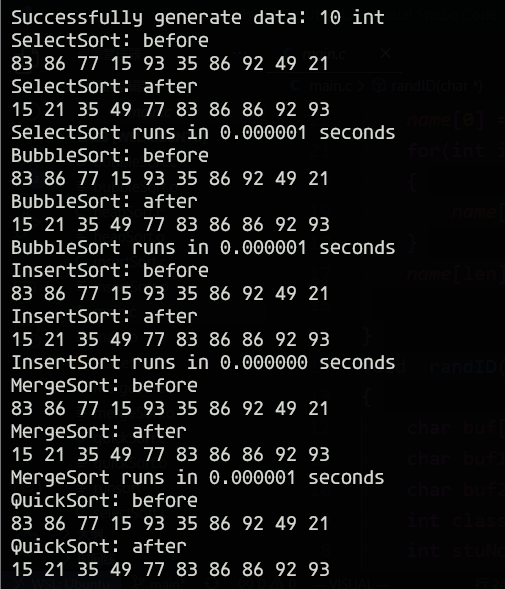
# 第2章 数据生成和性能测试

## 3.1 合理调用rands(),rand()生成数据（2分）

函数代码与说明：

1. **void**  generateData(ElemType \* A, **int** num)
2. {
3. **if**(dataType == **INT**)
4. {
5. **for**(**int** i=0; i<num; i++)
6. {
7. \*((**int**\*)(A+i)) = rand()%(10\*num);
8. }
9. }
10. **else** **if**(dataType == **FLOAT**)
11. {
12. **for**(**int** i=0; i<num; i++)
13. {
14. \*((**float**\*)(A+i)) = rand()%(10\*num);
15. }
16. }
17. **else** **if**(dataType == DOUBLE)
18. {
19. **for**(**int** i=0; i<num; i++)
20. {
21. \*((**double**\*)(A+i)) = rand()%(10\*num);
22. }
23. }
24. **else**
25. {
26. **for**(**int** i=0; i<num; i++)
27. {
28. randID(((STU\*)(A+i))->id);
29. randName( ((STU\*)(A+i))->name);
30. ((STU\*)(A+i))->total = 0;
31. **for**(**int** j=0; j<5; j++)
32. {
33. ((STU\*)(A+i))->score[j] = (rand()%10000)/((**double**) 100);
34. ((STU\*)(A+i))->total += ((STU\*)(A+i))->score[j];
35. }
36. }
37. }
38. }
40. **void**  randName(**char** \* name)
41. {
42. **char** ALPHA[26]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
43. **char** alpha[26]="abcdefghijklmnopqrstuvwxyz";
44. **int** len = 1+ rand()%18;
45. name[0] = ALPHA[rand()%26];
46. **for**(**int** i=1; i<len; i++)
47. {
48. name[i] = alpha[rand()%26];
49. }
50. name[len] = '\0';
52. }
53. **void**  randID(**char** \* id)
54. {
55. **char** buf[20] = "2021031";
56. **char** buf1[3];
57. **char** buf2[5];
58. **int** classNo = rand()%100;
59. **int** stuNo = rand()%10000;
60. sprintf(buf1, "%02d", classNo);
61. buf1[2] = '\0';
62. sprintf(buf2,  "%04d", stuNo);
63. buf2[4] = '\0';
64. strcat(buf, buf1);
65. strcat(buf, buf2);
66. strcpy(id, buf);
67. }

运行结果截图：

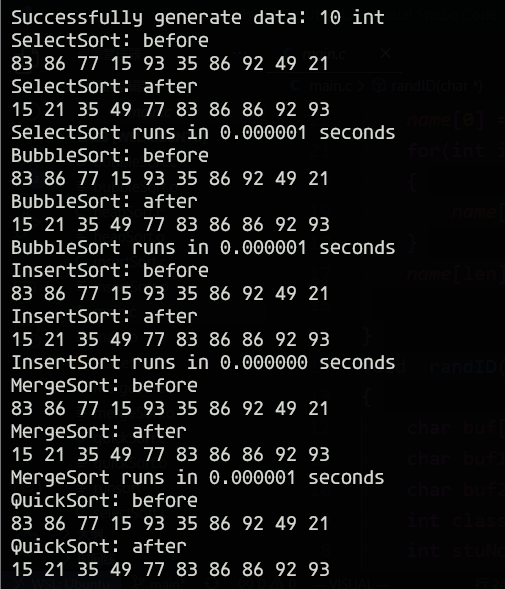


## 3.2 合理保障输入各个排序算法的数据一致性（2分）

函数代码与说明：

1. **void** output(ElemType A[],**int** n, **char** \* info)
2. {
3. printf("%s\n", info);
4. **switch**(dataType)
5. {
6. **case** **INT**:
7. **for**(**int** i=0; i<n; i++)
8. {
10. printf("%d ", \*((**int**\*)(A+i)));
11. }
12. printf("\n");
13. **break**;
15. **case** **FLOAT**:
16. **for**(**int** i=0; i<n; i++)
17. {
19. printf("%f ", \*((**float**\*)(A+i)));
20. }
21. printf("\n");
22. **break**;
23. **case** DOUBLE:
24. **for**(**int** i=0; i<n; i++)
25. {
27. printf("%f ", \*((**double**\*)(A+i)));
28. }
29. printf("\n");
30. **break**;
31. **case** STUDENT:
32. **for**(**int** i=0; i<n; i++)
33. {
35. printf("%s %-20s ", ((STU\*)(A+i))->id, ((STU\*)(A+i))->name);
36. **for**(**int** j=0; j<5; j++)
37. {
38. printf("%f ", ((STU\*)(A+i))->score[j]);
39. }
40. printf("%f \n", ((STU\*)(A+i))->total);
41. }
42. **break**;
43. **default**:
44. **break**;
45. }
46. }

运行结果截图：



## 3.3 获取、计算、输出各个排序算法的运行时间（2分）

函数代码与说明：

1. **int** main(**int** argc, **char** \*\* argv)
2. {
3. **size\_t** t1, t2;
4. **int** size;
5. ElemType \*orig, \*toSort;
6. **int** (\*cmpare)(ElemType \* a, ElemType \*b);
7. cmpare = &cmp;
8. **char** type[10] = {0};
10. **if**(argc<2)
11. **return** 1;
12. sscanf(argv[1], "%d", &size);
13. **if**(size<=0){
14. printf("Size must be a positive number");
15. **return** 0;
16. }
18. orig = (ElemType\*) malloc(**sizeof**(ElemType)\*size);
19. toSort = (ElemType\*) malloc(**sizeof**(ElemType)\*size);
20. generateData(orig, size);
22. **if**(dataType == **INT**) {
23. strcpy(type, "int");
24. } **else** **if** (dataType == **FLOAT**) {
25. strcpy(type, "float");
26. } **else** **if** (dataType == STUDENT) {
27. strcpy(type, "student");
28. } **else** {
29. strcpy(type, "double");
30. }
32. printf("Successfully generate data: %d %s\n", size, type);
34. memcpy(toSort, orig, size\***sizeof**(ElemType));
35. #ifdef Debug
36. output(toSort, size,"SelectSort: before");
37. #endif
38. t1 = clock();
39. SelectionSort(toSort, size, cmpare);
40. t2 = clock();
41. #ifdef Debug
42. output(toSort, size, "SelectSort: after");
43. #endif
44. printf("SelectSort runs in %f seconds\n", (t2-t1)/((**double**)CLOCKS\_PER\_SEC));


48. memcpy(toSort, orig, size\***sizeof**(ElemType));
49. #ifdef Debug
50. output(toSort, size,"BubbleSort: before");
51. #endif
52. t1 = clock();
53. BubbleSort(toSort, size, cmpare);
54. t2 = clock();
55. #ifdef Debug
56. output(toSort, size, "BubbleSort: after");
57. #endif
58. printf("BubbleSort runs in %f seconds\n", (t2-t1)/((**double**)CLOCKS\_PER\_SEC));

61. memcpy(toSort, orig, size\***sizeof**(ElemType));
62. #ifdef Debug
63. output(toSort, size,"InsertSort: before");
64. #endif
65. t1 = clock();
66. InsertSort(toSort, size, cmpare);
67. t2 = clock();
68. #ifdef Debug
69. output(toSort, size, "InsertSort: after");
70. #endif
71. printf("InsertSort runs in %f seconds\n", (t2-t1)/((**double**)CLOCKS\_PER\_SEC));

74. memcpy(toSort, orig, size\***sizeof**(ElemType));
75. #ifdef Debug
76. output(toSort, size,"MergeSort: before");
77. #endif
78. t1 = clock();
79. MergeSort(toSort, 0, size - 1, cmpare);
80. t2 = clock();
81. #ifdef Debug
82. output(toSort, size, "MergeSort: after");
83. #endif
84. printf("MergeSort runs in %f seconds\n", (t2-t1)/((**double**)CLOCKS\_PER\_SEC));


88. memcpy(toSort, orig, size\***sizeof**(ElemType));
89. #ifdef Debug
90. output(toSort, size,"QuickSort: before");
91. #endif
92. t1 = clock();
93. QuickSort(toSort, 0, size-1, cmpare);
94. t2 = clock();
95. #ifdef Debug
96. output(toSort, size, "QuickSort: after");
97. #endif
98. printf("QuickSort runs in %f seconds\n", (t2-t1)/((**double**)CLOCKS\_PER\_SEC));

101. memcpy(toSort, orig, size\***sizeof**(ElemType));
102. #ifdef Debug
103. output(toSort, size,"HeapSort: before");
104. #endif
105. t1 = clock();
106. HeapSort(toSort,size, cmpare);
107. t2 = clock();
108. #ifdef Debug
109. output(toSort, size, "HeapSort: after");
110. #endif
111. printf("HeapSort runs in %f seconds\n", (t2-t1)/((**double**)CLOCKS\_PER\_SEC));
113. **return** 1;
114. }

# 

# 第3章 排序算法

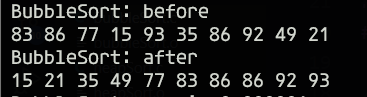
## 2.1 平方级排序算法（6分）

### 2.1.1冒泡排序算法（2分）

相关函数代码与说明：

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include "sort.h"
5. /\* #define DEBUG \*/
7. **void** BubbleSort(ElemType \*orig, **int** n, **int** (\*cmp)(ElemType\*, ElemType\*)){
8. **int** i, j;
9. **for**(i = 0; i < n - 1; i++) {
10. #ifdef DEBUG
11. **if** (i % 10000 == 0){
12. printf("BubbleSort processing %d/100\n", i / 10000);
13. }
14. #endif
15. **for**(j = 0; j < n - i ; j++){
16. **if**(cmp(orig + j - 1, orig + j) > 0){
17. Swap(orig, j - 1, j);
18. }
19. }
20. }
22. }

测试结果截图：



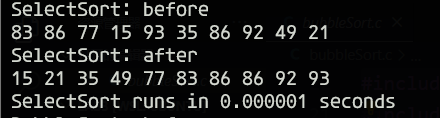
### 2.1.2选择排序算法（2分）

相关函数代码与说明：

1. #include "sort.h"
2. #include <stdio.h>
3. #include <stdlib.h>
5. /\* #define DEBUG \*/
7. **void** SelectionSort(ElemType \*orig, **int** n, **int** (\*cmp)(ElemType\*, ElemType\*)){
8. **int** i, j, k;
9. **for**(i = 0; i < n-1; i++) {
10. #ifdef DEBUG
11. **if** (i % 10000 == 0) {
12. printf("SelectionSort processing %d/100\n", i/10000);
13. }
14. #endif
15. j = i;
16. **for**(k = i + 1; k < n; k++) {
17. **if**(cmp(orig + j, orig + k) > 0){
18. j = k;
19. }
20. }
21. Swap(orig, i, j);
22. }
23. }

1

测试结果截图：

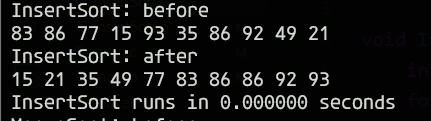


### 2.1.2插入排序算法（2分）

相关函数代码与说明：

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include "sort.h"
5. /\* #define DEBUG \*/
7. **void** InsertSort(ElemType \*orig, **int** n, **int**(\*cmp)(ElemType\*, ElemType\*)){
8. **int** i, j, k;
9. **for**(i = 1; i < n; i++) {
10. #ifdef DEBUG
11. **if** (i %10000 == 0) {
12. printf("InsertSort processing %d/100\n" , i /10000);
13. }
14. #endif
15. ElemType temp = orig[i];
16. j = i - 1;
17. **while**(j >= 0 && cmp(orig + j, &temp) > 0) {
18. orig[j + 1] = orig[j];
19. j--;
20. }
21. orig[j + 1] = temp;
22. }
23. }

测试结果截图：



## 

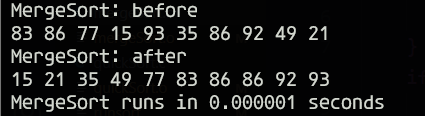
## 2.2 nlogn排序算法（14分）

## 2.2.1归并排序算法（7分）

相关函数代码与说明：

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include "sort.h"


8. **void** MergeSort(ElemType \*orig, **int** start, **int** end, **int** (\*cmp)(ElemType\*, ElemType\*)){
9. **int** middle = (start + end) / 2;
10. **if**(start >= end) {
11. **return**;
12. }
13. MergeSort(orig, start, middle, cmp);
14. MergeSort(orig, middle + 1, end, cmp);
16. ElemType\* temp = (ElemType \*)malloc(**sizeof**(ElemType) \* (end - start + 1));
18. **int** p1 = start, p2 = middle + 1, p3 = 0;
19. **while**(p1 <= middle && p2 <= end) {
20. **if**(cmp(orig + p1, orig + p2) < 0){
21. temp[p3] = orig[p1];
22. p1++;
23. p3++;
24. }
25. **else**{
26. temp[p3] = orig[p2];
27. p2++;
28. p3++;
29. }
30. }
31. **if**(p1 <= middle) {
32. memcpy(temp + p3, orig + p1, (middle - p1 + 1) \* **sizeof**(ElemType));
33. }
34. **if**(p2 <= end) {
35. memcpy(temp + p3, orig + p2, (end - p2 + 1) \* **sizeof**(ElemType));
36. }
37. memcpy(orig + start, temp, **sizeof**(ElemType) \* (end - start + 1));
38. free(temp);
40. }



测试结果截图：

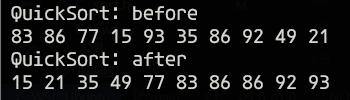
## 2.2.2快速排序算法（7分）

相关函数代码与说明：

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include "sort.h"

6. **void** QuickSort(ElemType \*orig, **int** start, **int** end, **int** (\*cmp)(ElemType\*, ElemType\*)){
7. **if** (start > end) {
8. **return**;
9. }
10. ElemType temp = orig[start];
11. **int** left = start;
12. **int** right = end;
13. **while** (left < right) {
14. **while**(cmp(orig + right, &temp) >= 0 && right > left) {
15. right --;
16. }
17. **while**(cmp(orig + left, &temp) <= 0 && right > left) {
18. left ++;
19. }
20. **if**(left < right) {
21. Swap(orig, left, right);
22. }
23. }
25. orig[start] = orig[left];
26. orig[left] = temp;
28. QuickSort(orig, start, left - 1, cmp);
29. QuickSort(orig, right + 1, end, cmp);
30. }

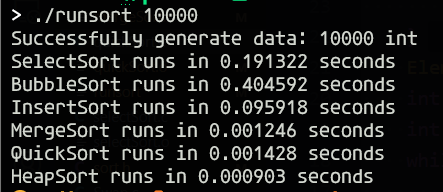
测试结果截图：



## 2.3 算法性能比较（3分）

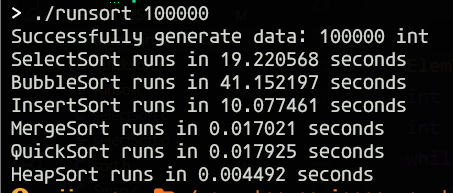
## 2.3.1数量级10000上的性能比较（1分）

测试结果截图：



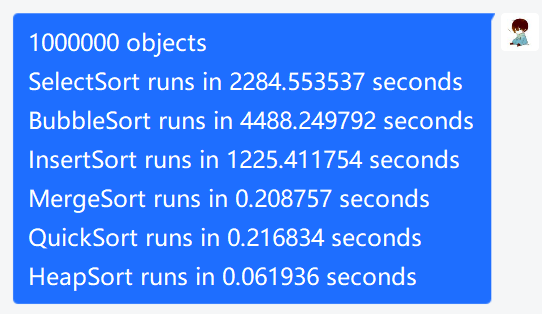
## 2.3.2数量级100000上的性能比较（1分）

测试结果截图：



## 2.3.3数量级1000000上的性能比较（1分）

测试结果截图：



# 第3章 堆排序与扩展研究

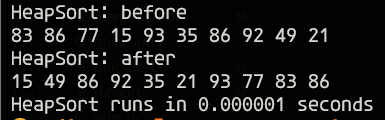
## 3.1 堆排序算法（5分）

函数代码与说明：

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include "sort.h"
5. **void** AdjustMaxHeap(ElemType\* orig, **int** n, **int** i, **int** (\*cmp)(ElemType\*, ElemType\*)){
6. **int** left = 2 \* i + 1;
7. **int** right = 2 \* i + 2;
8. **if** (left >= n) {
9. **return**;
10. }
11. **int** max = i;
12. **if** (cmp(orig + left, orig + i) > 0) {
13. max = left;
14. }
15. **if**(right < n && cmp(orig + right, orig + i) > 0) {
16. max = right;
17. }
19. **if** (max == i) {
20. **return**;
21. } **else**{
22. Swap(orig, max, i);
23. AdjustMaxHeap(orig, n, max, cmp);
24. }
25. }
27. **void** BuildMaxHeap(ElemType\* orig, **int** n, **int** (\*cmp)(ElemType\*, ElemType\*)) {
28. **for** (**int** i = (n-1)/2; i >= 0; i--) {
29. AdjustMaxHeap(orig, n, i, cmp);
30. }
31. }

34. **void** HeapSort(ElemType \*orig, **int** n, **int** (\*cmp)(ElemType\*, ElemType\*)){
35. BuildMaxHeap(orig, n, cmp);
36. **for**(**int** i = n - 1; i > 0; i--) {
37. Swap(orig, i, 0);
38. AdjustMaxHeap(orig, i, 0, cmp);
39. }
40. }

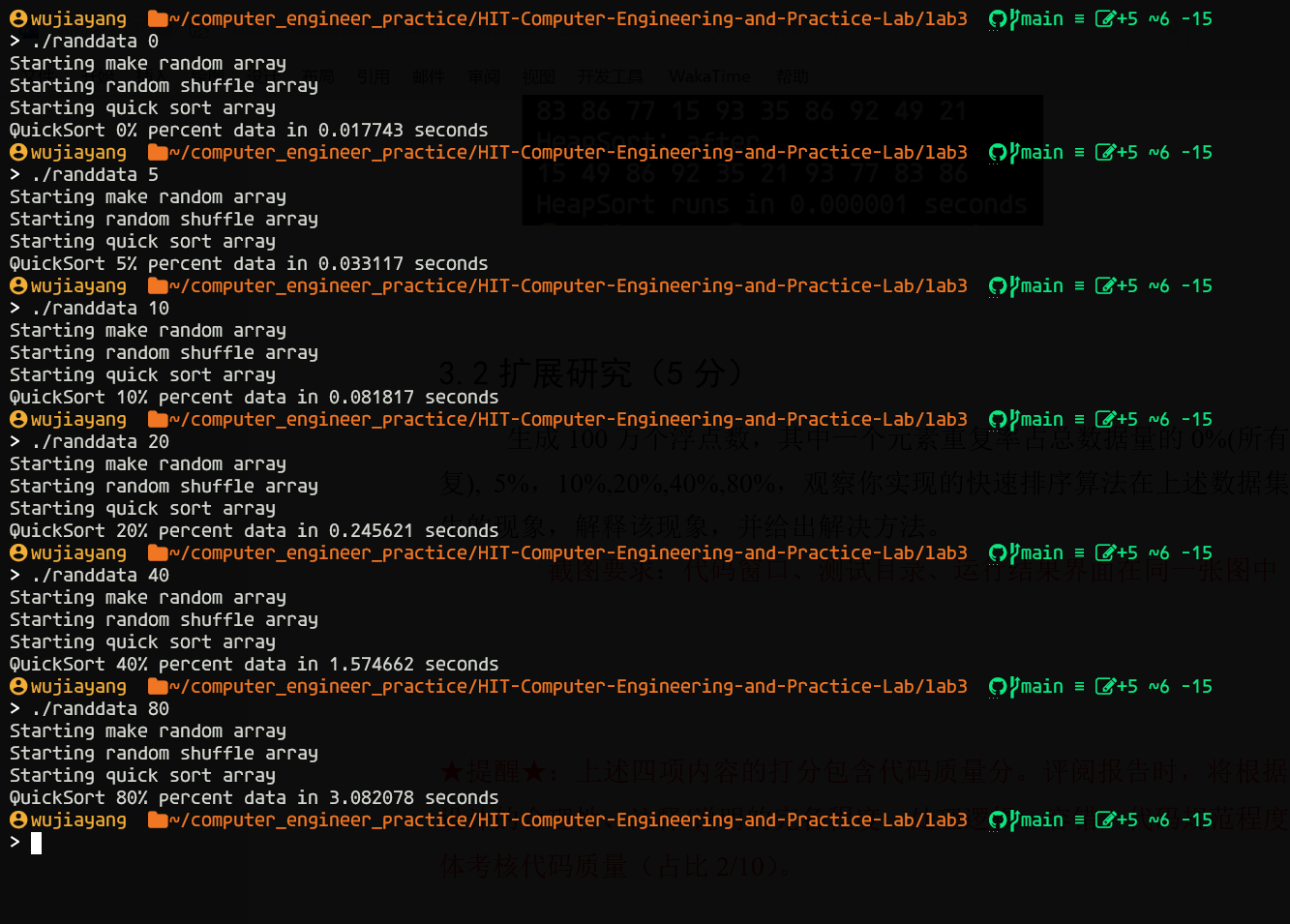
运行结果截图：



## 3.2扩展研究（5分）

生成100万个浮点数，其中一个元素重复率占总数据量的0%(所有元素不重复), 5%，10%,20%,40%,80%，观察你实现的快速排序算法在上述数据集序列上产生的现象，解释该现象，并给出解决方法。

1. #include "sort.h"
2. #include <time.h>
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <string.h>
7. #define NUM 100000
9. **int** IsRepeat(**float**\* A, **int** i, **float** temp) {
10. **for**(**int** j = 0; j < i; j++) {
11. **if**(\*(A + i) == temp) {
12. **return** 1;
13. }
14. }
15. **return** 0;
16. }
18. **void** MakeRand(**float** \*A, **int** percent) {
19. printf("Starting make random array\n");
20. **float** initial = rand()%(10 \* NUM);
21. **for**(**int** i = 0; i < NUM \* percent / 100; i++) {
22. \*(A + i) = initial;
23. }
24. **for**(**int** i = NUM \* percent / 100; i < NUM; i++) {
25. **float** temp= 0;
26. **do**{
27. temp = rand()%(10\*NUM);
28. } **while**(IsRepeat(A, i, temp) == 1);
29. \*(A + i) = temp;
30. }
31. printf("Starting random shuffle array \n");
32. **for**(**int** i = 0; i < NUM - 1; i++) {
33. **int** position = i + rand() % (NUM - i - 1);
34. **float** temp = A[i];
35. A[i] = A[position];
36. A[position] = temp;
37. }
38. }
40. **int** cmp(**float** \*a, **float** \*b) {
41. **if**(\*a ==\*b) {
42. **return** 0;
43. } **else** **if** (\*a > \*b) {
44. **return** 1;
45. } **else** **if**(\*a < \*b) {
46. **return** -1;
47. }
48. }
50. **int** main(**int** argc, **char** \*argv[]) {
51. **size\_t** t1, t2;
52. **int** (\*compare)(**float** \*a, **float** \*b);
53. compare = &cmp;
55. **if**(argc != 2) {
56. **return** -1;
57. }
58. **int** percent = atoi(argv[1]);
59. **float**\* A = (**float** \*)malloc(**sizeof**(**float**) \* NUM);
60. MakeRand(A, percent);
61. printf("Starting quick sort array\n");
62. t1 = clock();
63. QuickSort(A, 0, NUM - 1, compare);
64. t2 = clock();
65. printf("QuickSort %d%% percent data in %f seconds\n", percent, (t2 - t1)/((**double**)CLOCKS\_PER\_SEC));
66. **return** 0;
67. }



快速排序随着重复元素的增加，时间变长，因为一直对重复元素执行swap操作，导致时间变长