データ構造とアルゴリズム実験レポート

＜課題 5：整列＞

202213025 – 3 クラス – Can Minh Nghia

締切日：2023/12/11

提出日：2023/12/6

必須課題

課題 5-1

5-1-1 実装の仕様

1. 挿入ソート，ヒープソート，クイックソートを実装し，以下の要件を確認すること．なお，関数 swap, compare を改変し，動作の様子を確認できるようにした場合は加点する．
    - 関数 sift_down, build_heap, partition が正しく動作すること．
    - 各ソートについて， 6 ~ 8 個程度のデータを実際に整列する例を示し，動作について説明すること．

5-1-2 実装コードおよび実装コードの説明

まず、発展課題を解くには<math.h>のライブラリが必要だから、Makefile のファイルに -lm　を加えておく。

```
sort_collection: sort_collection.o main_sort_collection.o -lm
```

関数 swap, compare を改変し，動作の様子を確認できるようにする：

Compare 関数：

```
11 int compare(int ldata, int rdata) {
12   compare_count++;
13   if       (ldata  < rdata) {
14       printf("Left value %d is smaller than Right value %d!", ldata, rdata);
15       printf("\n");
16       return -1;
17   } else if (ldata == rdata) {
18       printf("Left value %d is equal to Right value %d!", ldata, rdata);
19       printf("\n");
20       return  0;
21   } else {
22       printf("Left value %d is larger than Right value %d!", ldata, rdata);
23       printf("\n");
24       return  1;
25   }
26 }
```

比較の結果を人間もちゃんと把握できるように printf 関数を加えた。

Swap 関数：

```
28 void swap(int a[], int lidx, int ridx) {
29   int temp = a[lidx];
30   a[lidx] = a[ridx];
31   a[ridx] = temp;
32   printf("Swapped the number at %d position (%d) with the number at %d position (%d)!", lidx, a[ridx], ridx,
   a[lidx]);
33   printf("\n");
34 }
```

同様に、printf 関数を使って、どの数字がどの数字と入れ替えられる(Swapped)か表す。

- 関数 sift_down, build_heap, partition が正しく動作すること.

Sift_down 関数：

```
76 void sift_down(int a[], int i, int n) { //this fuction rearrange the position of a parent and its 2 children, the
   largest number is parent
77   int maxChild, tmp;
78   while ((2 * i) + 1 < n) { //while the node at i position and its children exist in the tree. After sorting, we will
   either break the while loop (complete sorting) or moved to the leaf of the tree (when 2*i+1 = n)
79     maxChild = (2 * i) + 1;
80     if (((2 * i) + 2 < n) && (compare(a[(2 * i) + 2], a[maxChild]) == 1)) { //a[(2 * i) + 2 > a[(2 * i) + 1
81       maxChild = (2 * i) + 2;
82    }
83     if (compare(a[i], a[maxChild]) == -1) {
84       tmp = a[i];
85       a[i] = a[maxChild];
86       a[maxChild] = tmp;
87       i = maxChild; //put the index i to the position of that child that we just moved it to the parent position
88     } else {
89       break;
90    }
91  }
92 }
```

この関数は、親の位置にあるノードとそのノードが持っている子を比較する。親が子より大きかったら、ほかの親のノードに移動する。一方、子が親より大きかったら、2 つのこの中で最も大きい子を親と入れ替える。

Line 77: maxChild は最も大きい子で、temp は一時的な変数で、親と子を交換するときに使う。

Line 78: while ループで、親のノードが存在したら次の処理を行う。n は木のノードの数 (max nodes)。i はノードの位置で、2*i + 1 はそのノードの左の子。2i+ 1 < n とは i 位置のノードが少なくとも一つの子を持つということだ。

Line 79 ~ 82: 2 つのノードの中で、大きい方を探して、maxChild 変数に格納する。1 つの子しか持たない場合、その小は maxChild。特に Line 80 の if 分の条件 2*i + 1 < n とは、i 位置のノードは右の子も持つということだ。つまり i 位置ノードは 2 つの子を持つということだ。

Line 83 ~ 86:親と大きい子と比較して、もし大きい子のほうが大きかったら、その子を親と交換する。

Line 87: while ループで、親が代入された子の位置のノードに i をおく。I 位置のノードをもう一度そのノードの子と比較して、もしある子がそのノードより大きかったら、上記みたいに親を子と交換する。

Line 89: 全部の親が持っている全部の子より大きかったら、while ループを外れ、関数が終わる。

build_heap 関数

```
94 //make a max heap tree
95 void build_heap(int a[], int n) { //n is max node
96   int i;                          // i is index of current node
97   for (i = (n / 2) - 1; i >= 0; i--) { //initially i indicates the last parent node and we keep using sift_down until
   i = 0 (i went to the root)
98     sift_down(a, i, n);
99   }
100 }
```

この関数は max heap tree (親が子より大きい)を作る。

Line 97: (n/2) – 1 とは、全部の親ノードの中での最後の親ノード (最初の親ノードは根)。i--の書き方で、全部のノードを確認できる。

Line 98: sift_down 関数で、全部の親とその親が持つ子と比較して、大きいほうを親にすることができる。

partition 関数

```
116 int partition(int a[], int pivot, int left, int right) {
117   int pivotValue = a[pivot];
118   swap(a, pivot, right);
119   int storeIndex = left;
120
121   for (int i = left; i < right; i++) {
122     if (compare(a[i], pivotValue) == -1) {
123       swap(a, i, storeIndex); //meet a smaller value than pivot => swap it to the leftmost of the list, then forget
    it and move the leftmost position to the next position
124       storeIndex++; //store index = leftmost larger value than pivot => move all larger value next to left of pivot,
    then swap pivot with the leftmost one
125     }
126   }
127
128   swap(a, storeIndex, right); //swap pivot with the leftmost number of those which are all bigger than pivot
129   return storeIndex;
130 }
```

この関数は、pivot より大きい数を pivot の右に、pivot より小さい数を pivot の左に移動する。やり方としては、まず pivot と右端っこの数と交換して、pivot を右端っこに移動させる。次に、行列の全部の要素をチェックして、pivot より小さいものを左端っこに移動させ、pivot より大きいものを全部 pivot の左側の隣に移動させる。最後に、現在右端っこにある pivot を、pivot より大きい数の中で一番左にある数と位置を交換する。

図にすると：

最初：

小	大	小	大	pivot	小	大	大	大	大 (right)

pivot と右端っこの数と交換して、pivot を右端っこに移動させる:

小	大	小	大	大	小	大	大	大	Pivot (right)

pivot より小さいものを左端っこに移動させ、pivot より大きいものを全部 pivot の左側の隣に移動させる。ここで、storeIndex という変数を作って、pivot より大きい数の中で一番左にある数の位置を保存する:

小	小	小	大 (storeIndex)	大	大	大	大	大	Pivot (right)

pivot を storeIndex にある数と交換して、これで終わる：

小	小	小	Pivot (storeIndex)	大	大	大	大	大	大 (right)

説明：

Line 117:まず、pivot の値を pivotValue 変数に保存する。

Line 118:　pivot を右端っこの数(right)と交換する。

Line 119:　左端っこの数(left)の位置を storeIndex に保存する。

Line 121:　for ループを使って、行列の全部の要素をチェックする。

Line 122 ~ 124: pivot の値より小さい数が見つかったら、現在左端っこである storeIndex の位置に移動させる。(Line 123) 移動が終わったら、「これから移動されたばかりの pivot の値より小さい数がもう関係ない、もういらない」という意味で、(Line 124) storeIndex を次の位置に移動させる。次回、もし pivot より小さい数が見つかったら、また処理している行列の左端っこである storeIndex の位置に移動させて、storeIndex をまた次の位置に移動させる

上記にも見せたが、上のの処理が終わったら、行列はこうなる：

小	小	小	大 (storeIndex)	大	大	大	大	大	Pivot (right)

Line 128: pivot を storeIndex 位置にある数と交換する。つまり、最後に、行列はこうなる：

小	小	小	pivot (storeIndex)	大	大	大	大	大	大 (right)

Line 129: storeIndex 位置にある pivot を返す。

5-1-3 各ソートの動作確認と説明

- 各ソートについて，６～８個程度のデータを実際に整列する例を示し，動作について説明すること．

挿入ソート

まず main 関数の line 19 を insertion_sort に変える。

```
10 int main(int argc, char *argv[]) {
11   if (argc != 1) {
12     int numdata = atoi(argv[1]);  // set numdata with cmd. argument
13     int *array = (int*)malloc(sizeof(int) * numdata);
14     int i;
15     printf("Enter %d integers\n", numdata);
16     for (i = 0; i < numdata; i++) {
17       scanf("%d", &array[i]);  // enter integers
18    }
19     insertion_sort(array, numdata);
20     printf("sorting result\n");
21     display(array, numdata);
22     printf("# of comparisons: %lu\n", compare_count);
23     free(array);
24   } else {
```

(おさらい)挿入ソート関数：

```
57 // Insertion sort
58 /**********************************/
59 void insertion_sort(int a[], int n) {
60   int i, key, j;
61   for (i = 1; i < n; i++) {
62     key = a[i];
63     j = i - 1;
64
65     while (j >= 0 && compare(a[j], key) == 1) {
66       a[j + 1] = a[j];
67       j = j - 1;
68    }
69     a[j + 1] = key;
70  }
71 }
```

実装の結果：

```
azalea01:~ s2213025$ ./sort_collection 6
Enter 6 integers
1
3
5
2
4
6
Left value 1 is smaller than Right value 3!
Left value 3 is smaller than Right value 5!
Left value 5 is larger than Right value 2!
Left value 3 is larger than Right value 2!
Left value 1 is smaller than Right value 2!
Left value 5 is larger than Right value 4!
Left value 3 is smaller than Right value 4!
Left value 5 is smaller than Right value 6!
sorting result
1 2 3 4 5 6
# of comparisons: 8
```

動作の説明：

まず、行列 a は[1, 3, 5, 2, 4, 6]

最初から：

・i = 1, key = a[i] = 3, j = i − 1 = 0. a[0] = 1 < 3 なので while ループの条件を満たさない。よって、a[j + 1] = key = 3。このとき、行列 a は

[1, 3, 5, 2, 4, 6] （何も変わっていない)。for ループで i = i+1 = 2。

・i = 2, key = a[2] = 5, j = i − 1 = 1. a[1] = 3 < 5 なので while ループの条件を満たさない。よって、a[j + 1] = key = 5。このとき、行列 a は

[1, 3, 5, 2, 4, 6] （何も変わっていない)。for ループで i = i+1 = 3。

・i = 3, key = a[3] = 2, j = i − 1 = 2. a[2] = 5 > 2 なので while ループの条件を満たす。よって、a[j + 1] = a[j] = 5 。このとき、行列 a は

[1, 3, 5, 5, 4, 6] （値 5 が右にシフトされた)。j = j − 1 = 1。このとき、a[j] = a[1] = 3 > 2 なので while ループがまだ実行している。よって、

a[j + 1] = a[j] = 3 。このとき、行列 a は [1, 3, 3, 5, 4, 6] （値 3 が右にシフトされた)。while ループから出て、

a[j + 1] = key = 2。行列 a は [1, 2, 3, 5, 4, 6]。for ループで i = i+1 = 4。

・i = 4, key = a[4] = 4, j = i − 1 = 3. a[3] = 5 > 4 なので while ループの条件を満たす。よって、a[j + 1] = a[j] = 5 。このとき、行列 a は

[1, 2, 3, 5, 5, 6] （値 5 が右にシフトされた)。j = j − 1 = 2。このとき、a[j] = a[2] = 3 < 4 なので while ループから出て、a[j + 1] = key = 4。この時、行列 a は [1, 2, 3, 4, 5, 6]。for ループで i = i+1 = 5。

・i = 5, key = a[5] = 6, j = i – 1 = 4. a[4] = 5 < 6 なので while ループの条件を満たさない。よって、a[j + 1] = key = 6。このとき、行列 a は [1, 2, 3, 4, 5, 6]。for ループで i = i+1 = 6。

・i = 6 = n なので、for ループが終わる。従って、挿入ソート関数も終わる。

ヒープソート

main 関数の line 19 を heap_sort に変える。

(おさらい) ヒープソート関数：

```
74 // Functions for Heap sort
75 /***********************************/
76 void sift_down(int a[], int i, int n) { //this fuction rearrange the position of a parent and its 2 children, the
   largest number is parent
77   int maxChild, tmp;
78   while ((2 * i) + 1 < n) { //while the node at i position and its children exist in the tree. After sorting, we will
   either break the while loop (complete sorting) or moved to the leaf of the tree (when 2*i+1 = n)
79     maxChild = (2 * i) + 1;
80     if (((2 * i) + 2 < n) && (compare(a[(2 * i) + 2], a[maxChild]) == 1)) { //a[(2 * i) + 2 > a[(2 * i) + 1
81       maxChild = (2 * i) + 2;
82    }
83     if (compare(a[i], a[maxChild]) == -1) {
84       tmp = a[i];
85       a[i] = a[maxChild];
86       a[maxChild] = tmp;
87       i = maxChild; //put the index i to the position of that child that we just moved it to the parent position
88     } else {
89       break;
90    }
91   }
92 }
93
94 //make a max heap tree
95 void build_heap(int a[], int n) { //n is max node
96   int i;                          // i is index of current node
97   for (i = (n / 2) - 1; i >= 0; i--) { //initially i indicates the last parent node and we keep using sift_down until
   i = 0 (i went to the root)
98     sift_down(a, i, n);
99   }
100 }
```

```
103 void heap_sort(int a[], int n) {
104   build_heap(a, n); //after this build_heap funtion, we have a complete reverse heap tree (like [6, 5, 4, 3, 2])
105   int i;
106   for (i = n - 1; i > 0; i--) {
107       //2 function below work as we delete the root from a heap tree
108     swap(a, 0, i); //keep swapping the last element to the first element (like [2, 5, 4, 3, 6]), now we "deleted" the
   last element (6) and use sift_down to reaggange the reverse heap tree
109     sift_down(a, 0, i); //this sift_down rearranges the reverse heap tree as it keeps the largest number at root
110
111 }
```

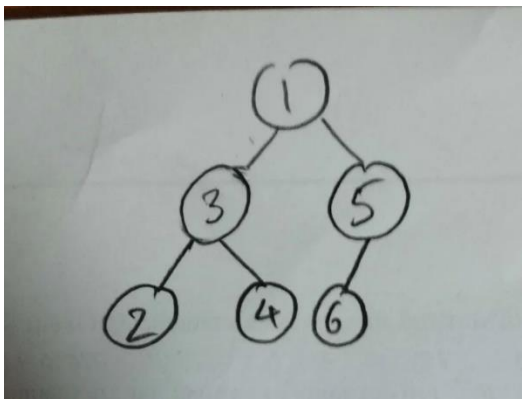
実装の結果：

行列 a[1, 3, 5, 2, 4, 6]を使う。

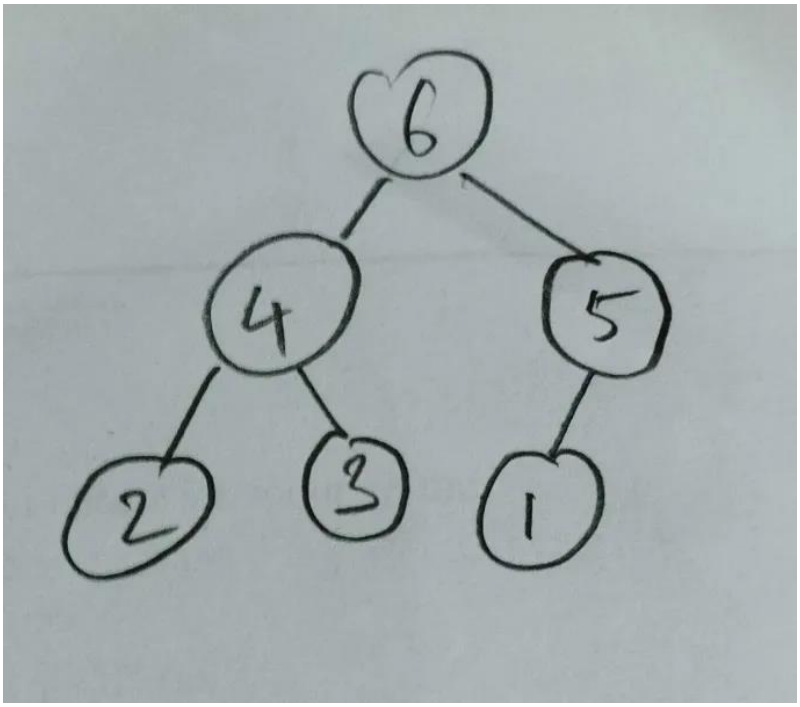
```
Left value 5 is smaller than Right value 6!
Left value 4 is larger than Right value 2!
Left value 3 is smaller than Right value 4!
Left value 6 is larger than Right value 4!
Left value 1 is smaller than Right value 6!
Left value 1 is smaller than Right value 5!
Swapped the number at 0 position (6) with the number at 5 position (1)!
Left value 5 is larger than Right value 4!
Left value 1 is smaller than Right value 5!
Swapped the number at 0 position (5) with the number at 4 position (3)!
Left value 1 is smaller than Right value 4!
Left value 3 is smaller than Right value 4!
Left value 3 is larger than Right value 2!
Swapped the number at 0 position (4) with the number at 3 position (2)!
Left value 1 is smaller than Right value 3!
Left value 2 is smaller than Right value 3!
Swapped the number at 0 position (3) with the number at 2 position (1)!
Left value 1 is smaller than Right value 2!
Swapped the number at 0 position (2) with the number at 1 position (1)!
sorting result
1 2 3 4 5 6
# of comparisons: 14
```

動作の説明：

最初に、max heap tree は：

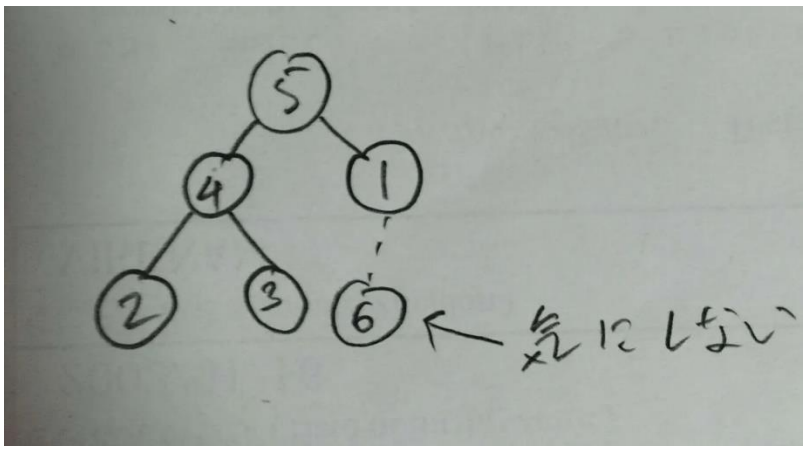


heap_sort 関数の中の build_heap 関数が実行された後、max heap tree はこうなる：

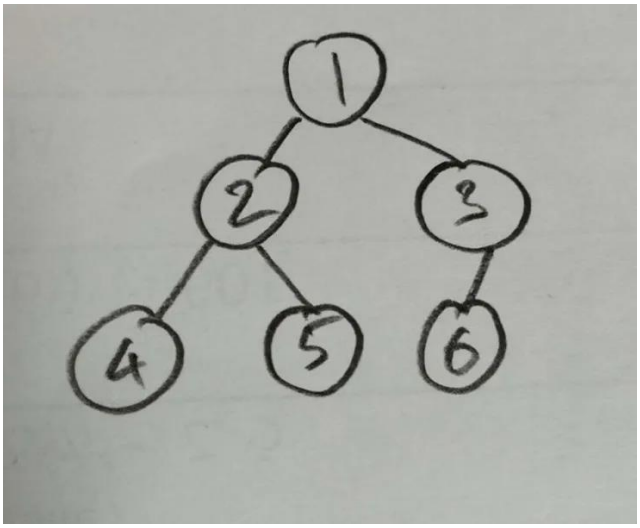


つまり、max heap tree は成功に作られた。

次、heap_sort 関数の for ループによって、i は最後のノードの位置で、根の値 6 を最後の葉ノード 1 と交換する。その後、6 を気にしないで、根から sift_down(a, 0, i)で max heap tree を作り直す。この時、max heap tree はこうなる：



その次、for ループでの i--記号によって、i が 6 を外して、次の最後の葉ノード 3 に移動する。また根となる 5 を最後の葉ノードと交換してから、sift_down(a, 0, i)で max heap tree を作り直す。そういう過程を繰り返し、結局、max heap tree はこうなる：



従って、a はこうなる：a[1, 2, 3, 4, 5, 6]

クイックソート

main 関数の line 19 を q_sort に変える。

(おさらい) クイックソート関数：

```
114 // Functions for Quick sort
115 /**********************************/
116 int partition(int a[], int pivot, int left, int right) {
117   int pivotValue = a[pivot];
118   swap(a, pivot, right);
119   int storeIndex = left;
120
121   for (int i = left; i < right; i++) {
122     if (compare(a[i], pivotValue) == -1) {
123       swap(a, i, storeIndex); //meet a smaller value than pivot => swap it to the leftmost of the list, then forget
    it and move the leftmost position to the next position
124       storeIndex++; //store index = leftmost larger value than pivot => move all larger value next to left of pivot,
    then swap pivot with the leftmost one
125     }
126   }
127
128   swap(a, storeIndex, right); //swap pivot with the leftmost number of those which are all bigger than pivot
129   return storeIndex;
130 }
131
132 void quick_sort(int a[], int left, int right) {
133   if (left < right) { //there are still element(s)in list
134     int pivot = left + (right - left) / 2;
135     int newPivot = partition(a, pivot, left, right); //newPivot still has the value of pivot, however the list a[]
    now is sorted as smaller than pivot went to left, larger than pivot went to right
136     quick_sort(a, left, newPivot - 1);
137     quick_sort(a, newPivot + 1, right);
138   }
139 }
```

```
141 void q_sort(int a[], int n) {
142   quick_sort(a, 0, n - 1);
143 }
```

実装の結果：

行列 a[1, 3, 5, 2, 4, 6]を使う。

```
Left value 1 is smaller than Right value 5!
Swapped the number at 0 position (1) with the number at 0 position (1)!
Left value 3 is smaller than Right value 5!
Swapped the number at 1 position (3) with the number at 1 position (3)!
Left value 6 is larger than Right value 5!
Left value 2 is smaller than Right value 5!
Swapped the number at 3 position (2) with the number at 2 position (6)!
Left value 4 is smaller than Right value 5!
Swapped the number at 4 position (4) with the number at 3 position (6)!
Swapped the number at 4 position (6) with the number at 5 position (5)!
Swapped the number at 1 position (3) with the number at 3 position (4)!
Left value 1 is smaller than Right value 3!
Swapped the number at 0 position (1) with the number at 0 position (1)!
Left value 4 is larger than Right value 3!
Left value 2 is smaller than Right value 3!
Swapped the number at 2 position (2) with the number at 1 position (4)!
Swapped the number at 2 position (4) with the number at 3 position (3)!
Swapped the number at 0 position (1) with the number at 1 position (2)!
Left value 2 is larger than Right value 1!
Swapped the number at 0 position (2) with the number at 1 position (1)!
sorting result
1 2 3 4 5 6
# of comparisons: 9
```

実装の説明：

まず、q_sort 関数を読んで、right = 0, left = n-1 = 5 という 2 つの引数を quick_sort 関数に渡す。pivot = 5 / 2 = 2 で、つまり a[pivot] = 5。quick_sort 関数にある partition 関数を呼んだ後、a はこうなる：a[1, 3, 2, 4, 5, 6]。詳しく説明すると:

・まず a[pivot]と a[right]を交換して、a は[1, 3, 6, 2, 4, 5]。pivotValue = 5。

・i = 0, storeIndex = 0, a[i] = 1 < 5、よって a[i]を a[pivot]と交換する。この 2 つは重ねているので、行列 a は変わっていない。storeIndex++より storeIndex = 1。

・i = 1, storeIndex = 1, a[i] = 3 < 5、よって a[i]を a[pivot]と交換する。この 2 つは重ねているので、行列 a は変わっていない。storeIndex++より storeIndex = 2。

・i = 2, storeIndex = 2, a[i] = 6 ＞5、if 分の条件を満たさないから何も起こらない。

・i = 3, storeIndex = 2, a[i] = 2＜5、よって a[i]を a[pivot]と交換する。この時、a は[1, 3, 2, 6, 4, 5]。storeIndex++より storeIndex = 3。

・i = 4, storeIndex = 3, a[i] = 4＜ 5、よって a[i]を a[pivot]と交換する。この時、a は[1, 3, 2, 4, 6, 5]。storeIndex++より storeIndex = 4。

・i = 5, for ループの条件を満たさないから for ループから出る。

・Line 128: swap(a, storeIndex, right)のよって、a は[1, 3, 2, 4, 5, 6]

・storeIndex = 4 を返す。この時、storeIndex 位置にあるのは pivot であるから、つまり pivot を返す。

Line 136 ~ 137:ここで、quick_sort 関数の処理を引き続き行う。2 回再帰を使って、上記の過程を繰り返し、pivot の左の部分と右の部分をソートする。

課題 5-2

5-2-1 実装の仕様

ヒープソートおよびクイックソートの性能を分析せよ。

5-2-2 実装コードおよび実装コードの説明

1. ランダムな入力データを要素として持つ大きさ numdata (numdata = 1000, 2000, ..., 10,000)の配列を生成し，各整列法を実行し，整列に要するデータの比較回数を調べ，結果をグラフを用いて分析すること.

ヒープソート：

まず、main 関数の line 29 を heap_sort に変える。

```
10 int main(int argc, char *argv[]) {
11   if (argc != 1) {
12     int numdata = atoi(argv[1]);  // set numdata with cmd. argument
13     int *array = (int*)malloc(sizeof(int) * numdata);
14     int i;
15     printf("Enter %d integers\n", numdata);
16     for (i = 0; i < numdata; i++) {
17       scanf("%d", &array[i]);  // enter integers
18     }
19     q_sort(array, numdata);
20     printf("sorting result\n");
21     display(array, numdata);
22     printf("# of comparisons: %lu\n", compare_count);
23     free(array);
24   } else {
25     int numdata;
26     for (numdata = 1000; numdata <= 10000; numdata += 1000) {  // numdata is 1000, 2000, ..., 10,000
27       int *array = (int*)malloc(sizeof(int) * numdata);
28       int i;
29       for (i = 0; i < numdata; i++) {
30         array[i] = GetRandom(0, (numdata * 10 - 1));  // random number from 0 to numdata * 10 - 1
31      }
32       heap_sort(array, numdata);
33       printf("%d %lu\n", numdata, compare_count);
34       cmp_cnt_reset();
35       free(array);
36    }
37  }
```

ターミナルで、このように入力する：

```
azalea01:~ s2213025$ make

azalea01:~ s2213025$ ./sort_collection
```

./sort_collection の後は何も書かない。従って、結果はこうなる：

```
azalea01:~ s2213025$ ./sort_collection
1000 16835
2000 37769
3000 60289
4000 83407
5000 107653
6000 132549
7000 157515
8000 182826
9000 208940
10000 235395
```

つまり、numdata = 1000 の時、compare_count = 16835 で、numdata = 2000 の時、compare_count = 37769 等々。

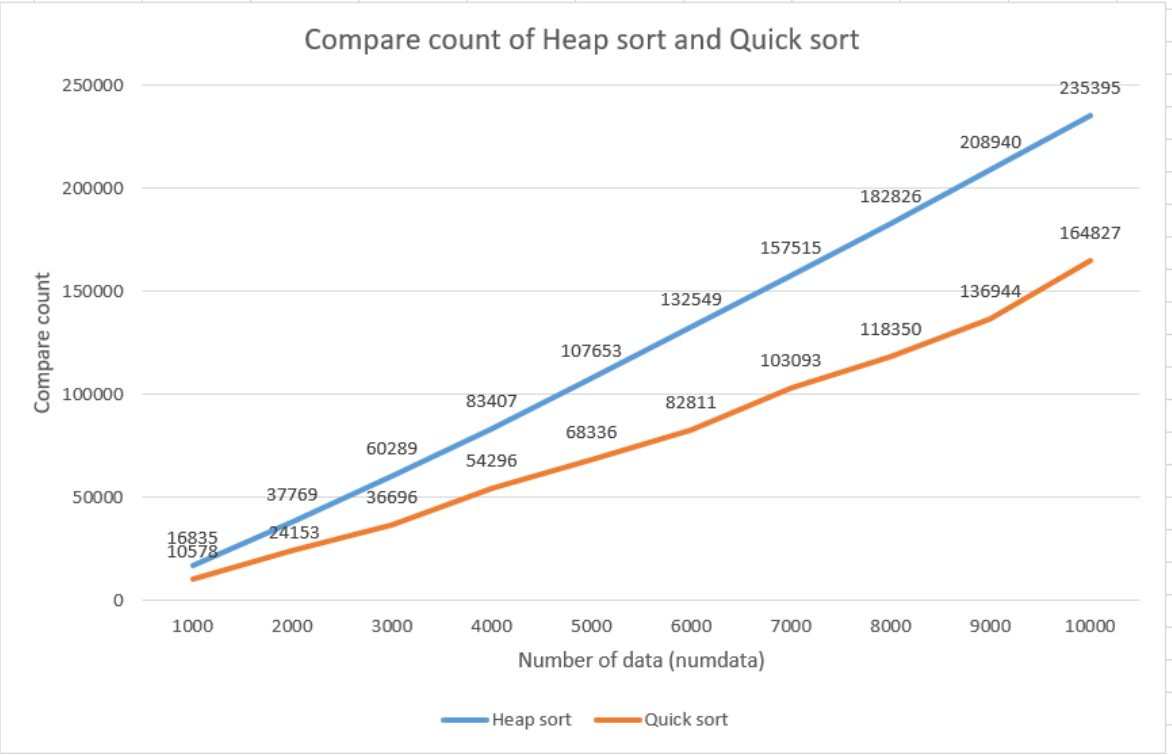
クイックソート：

main 関数の line 29 を q_sort に変えて、ターミナルは上記のようにしたら、結果はこうなる：

```
azalea01:~ s2213025$ ./sort_collection
1000 10578
2000 24153
3000 36696
4000 54296
5000 68336
6000 82811
7000 103093
8000 118350
9000 136944
10000 164827
```

グラフ：

	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Heap sort	16835	37769	60289	83407	107653	132549	157515	182826	208940	235395
Quick sort	10578	24153	36696	54296	68336	82811	103093	118350	136944	164827



分析：グラフを見ると、クイックソートがヒープソートより比較数(compare count)が少ないのは明らかに見える。ただし、面白いのは、もう少し近くに見ると、データ量が多いほど、ヒープソートとクイックソートの比較数の差の比率が小さくなる。最初のところ、データ量が 1000 個の場合、ヒープソートの比較数は 16835 で、クイックソートの比較数の約 160%であるが、データ量が多くなると、その差の比率がどんどん縮まり続ける。結局、データ量が 10000 個の場合 (最初のデータ量の 10 倍)、その差が 235385 対 164827 で、約 143%で、最初の比率よりおよそ 17%少ない。もう一つの注意すべき点は、ヒープソートの比較数の増加がすごく安定で、凸凹のない斜めのように見える。一方、クイックソートの方は折れ線のように見えて、安定性がヒープソートより劣る。

2．ランダムな入力データを要素として持つ配列を生成し，各整列法を実行し，整列に要する実行時間を調べ，結果をグラフを用いて分析すること． 配列の大きさは，実行時間が 0.1 ~ 数秒程度になるように選ぶこと (実行時間を調べる際には，課題 3: ハッシュ法に記載した事項を念頭に置きましょう).

まず、main 関数を sys/time.h ライブラリを導入する。

```
5 #include <sys/time.h>
```

次に、このように変更する：

```
25  } else {
26    int numdata;
27
28    struct timeval start, end; //counting time
29    gettimeofday(&start, NULL); //counting time
30
31    for (numdata = 1000000; numdata <= 10000000; numdata += 1000000) {  // numdata is 1 million, 2 million, ..., 10
  million
32      int *array = (int*)malloc(sizeof(int) * numdata);
33      int i;
34      for (i = 0; i < numdata; i++) {
35        array[i] = GetRandom(0, (numdata * 10 - 1));  // random number from 0 to numdata * 10 - 1
36     }
37      q_sort(array, numdata);
38      gettimeofday(&end, NULL); //counting time
39      double elapsed_time = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6; //counting time
40      printf("numdata = %d: time = %lf[sec]\n", numdata, elapsed_time); //print time
41      //printf("%d %lu\n", numdata, compare_count);
42      cmp_cnt_reset();
43      free(array);
44    }
45  }
```

変更したもの：

・struct timeval, gettimeofday(), elapsed_time を加えて、時間を計る。

・printf("numdata = %d: time = %lf[sec]¥n", numdata, elapsed_time);　を使って、処理時間を表す。

・numdata を 1000000 の倍数にする。

・Line 37 は heap_sort、または q_sort に変更する。

ヒープソート

結果はこうなる：

```
azalea01:~ s2213025$ ./sort_collection
numdata = 1000000: time = 0.193051[sec]
numdata = 2000000: time = 0.600955[sec]
numdata = 3000000: time = 1.234753[sec]
numdata = 4000000: time = 2.105723[sec]
numdata = 5000000: time = 3.231377[sec]
numdata = 6000000: time = 4.639540[sec]
numdata = 7000000: time = 6.290191[sec]
numdata = 8000000: time = 8.216521[sec]
numdata = 9000000: time = 10.432662[sec]
numdata = 10000000: time = 12.963673[sec]
```

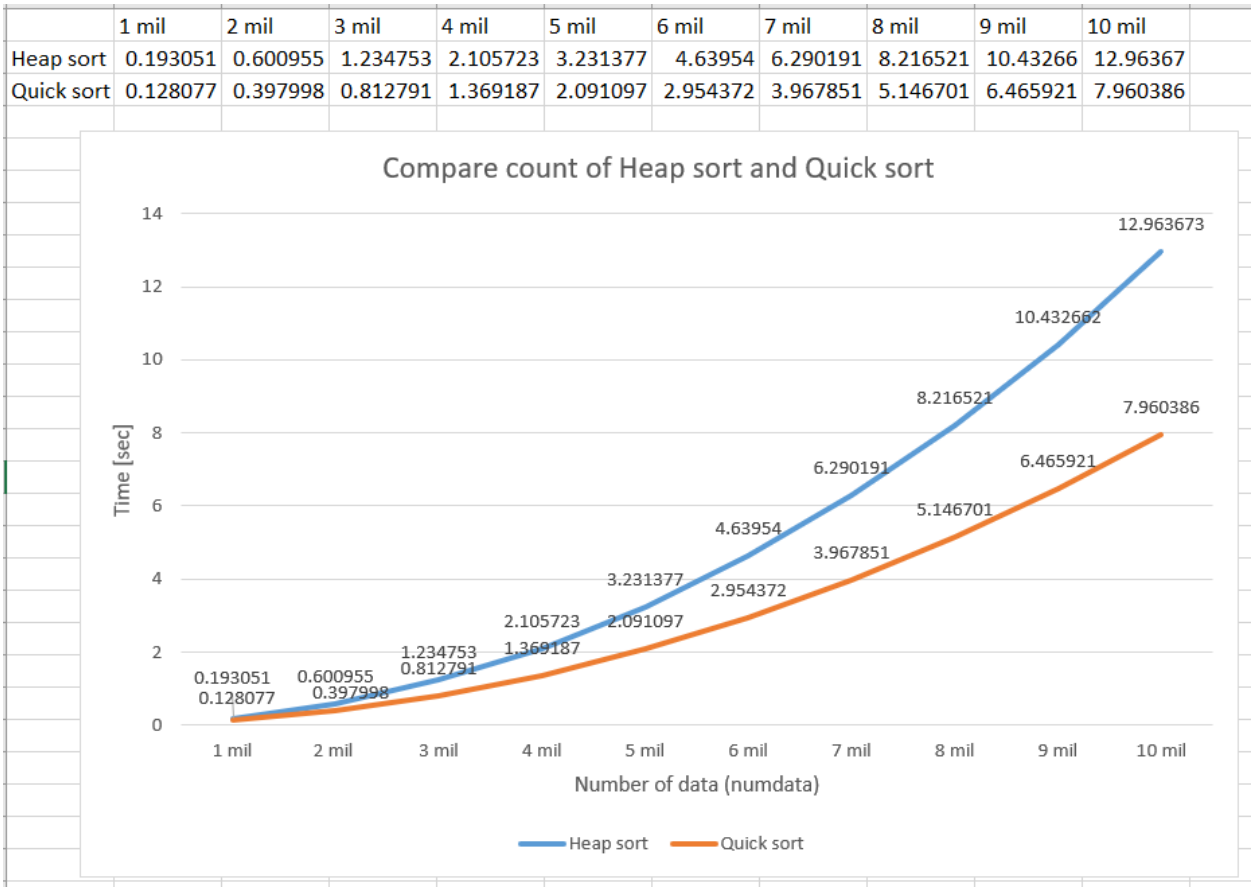
ヒクイックソート

結果はこうなる：

```
azalea01:~ s2213025$ ./sort_collection
numdata = 1000000: time = 0.128077[sec]
numdata = 2000000: time = 0.397998[sec]
numdata = 3000000: time = 0.812791[sec]
numdata = 4000000: time = 1.369187[sec]
numdata = 5000000: time = 2.091097[sec]
numdata = 6000000: time = 2.954372[sec]
numdata = 7000000: time = 3.967851[sec]
numdata = 8000000: time = 5.146701[sec]
numdata = 9000000: time = 6.465921[sec]
numdata = 10000000: time = 7.960386[sec]
```

グラフ

	1 mil	2 mil	3 mil	4 mil	5 mil	6 mil	7 mil	8 mil	9 mil	10 mil
Heap sort	0.193051	0.600955	1.234753	2.105723	3.231377	4.63954	6.290191	8.216521	10.43266	12.96367
Quick sort	0.128077	0.397998	0.812791	1.369187	2.091097	2.954372	3.967851	5.146701	6.465921	7.960386



分析：グラフを見ると、ヒープソートの処理時間がクイックソートの処理時間より長いことがわかる。特に、データ量が多いほど、その差も広がる。最初のところ、1,000,000 個のデータを処理するには、ヒープソートに 0.193051 秒が必要で、クイックソートの約 150%である。しかし、最後のところ、10,000,000 個のデータを処理するには、ヒープソートには 12.963673 秒が必要で、クイックソートのおよそ 163%である。結論として、データが多いほど、クイックソートの方が優れているともいえるだろう。

3.

クイックソートの計算量が最悪 (すなわち，$O(n^2)$の計算量) になるような配列を生成し，最悪の場合の性能を整列に要するデータの比較回数を調べ，結果をグラフを用いて分析すること.

最悪ケース(worst case)は、データはもうソートされたデータ、それに pivot は端っこにある。

次に、もうソートされたデータを作る。main 関数をこう変更する：

```
25  } else {
26    int numdata;
27
28    struct timeval start, end; //counting time
29    gettimeofday(&start, NULL); //counting time
30
31    for (numdata = 10000; numdata <= 100000; numdata += 10000) {  // numdata is 10000, 20000, ..., 100000
32      printf("numdata %d\n", numdata);
33      int *array = (int*)malloc(sizeof(int) * (numdata));
34      int i;
35      for (i = 0; i < numdata; i++) {
36        //array[i] = GetRandom(0, (numdata * 10 - 1));  // random number from 0 to numdata * 10 - 1
37        array[i] = i; //create increasing data (sorted data)
38      }
39      printf("data created!");
40      q_sort(array, numdata);
```

Line 31: numdata の範囲を 100000 個にする。実は、教室でも TA さんたちと相談して、200000 個ぐらいまでは実行してみたが、segmentation fault になる。原因は計算量が多すぎて、segmentation fault になることが判断できる。これで、n が大きいほど、O(n logn)と $O(n^2)$の違いは想像以外にすごいことがわかる。

Line 37: 単調増加行列を作る。ソートされたデータの役を立つ。

Line 39: ソートされたデータが作られたかどうか確認する。

quick_sort 関数もこのように変更する：

```
132 void quick_sort(int a[], int left, int right) {
133   if (left < right) { //there are still element(s)in list
134     //int pivot = left + (right - left) / 2;
135     int pivot = right; //reate pivot in worst case
136       int newPivot = partition(a, pivot, left, right); //newPivot still has the value of pivot, however the list
    a[] now is sorted as smaller than pivot went to left, larger than pivot went to right
137       quick_sort(a, left, newPivot - 1);
138       quick_sort(a, newPivot + 1, right);
139   }
140 }
```

Line 135: pivot を端っこにおく。

実行結果：

```
azalea02:~ s2213025$ ./sort_collection
numdata 10000
data created!numdata = 10000: time = 0.144285[sec]
numdata 20000
data created!numdata = 20000: time = 0.690561[sec]
numdata 30000
data created!numdata = 30000: time = 1.914602[sec]
numdata 40000
data created!numdata = 40000: time = 4.148914[sec]
numdata 50000
data created!numdata = 50000: time = 7.586868[sec]
numdata 60000
data created!numdata = 60000: time = 12.546064[sec]
numdata 70000
data created!numdata = 70000: time = 19.451598[sec]
numdata 80000
data created!numdata = 80000: time = 28.281091[sec]
numdata 90000
data created!numdata = 90000: time = 39.454554[sec]
numdata 100000
data created!numdata = 100000: time = 53.257874[sec]
```

次に、最悪ケースではない場合：

main 関数はこう変更する：

```
25  } else {
26    int numdata;
27
28    struct timeval start, end; //counting time
29    gettimeofday(&start, NULL); //counting time
30
31    for (numdata = 10000; numdata <= 100000; numdata += 10000) {  // numdata is 10000, 20000, ..., 100000
32      printf("numdata %d\n", numdata);
33      int *array = (int*)malloc(sizeof(int) * (numdata));
34      int i;
35      for (i = 0; i < numdata; i++) {
36        array[i] = GetRandom(0, (numdata - 1));  // random number from 0 to numdata - 1
37        //array[i] = i; //create increasing data (sorted data)
38      }
39      printf("data created!");
40      q sort(array, numdata);
```

Line 36: GetRandom 関数を使って、ソートされていないデータを作る。データの範囲は最悪ケースと同じだ。

quick_sort 関数もこのように変更する：

```
132 void quick_sort(int a[], int left, int right) {
133   if (left < right) { //there are still element(s)in list
134    int pivot = left + (right - left) / 2; //create pivot in norrmal case
135    //int pivot = right; //create pivot in worst case
136     int newPivot = partition(a, pivot, left, right); //newPivot still has the value of pivot, however the list
    a[] now is sorted as smaller than pivot went to left, larger than pivot went to right
137     quick_sort(a, left, newPivot - 1);
138     quick_sort(a, newPivot + 1, right);
139   }
140 }
```

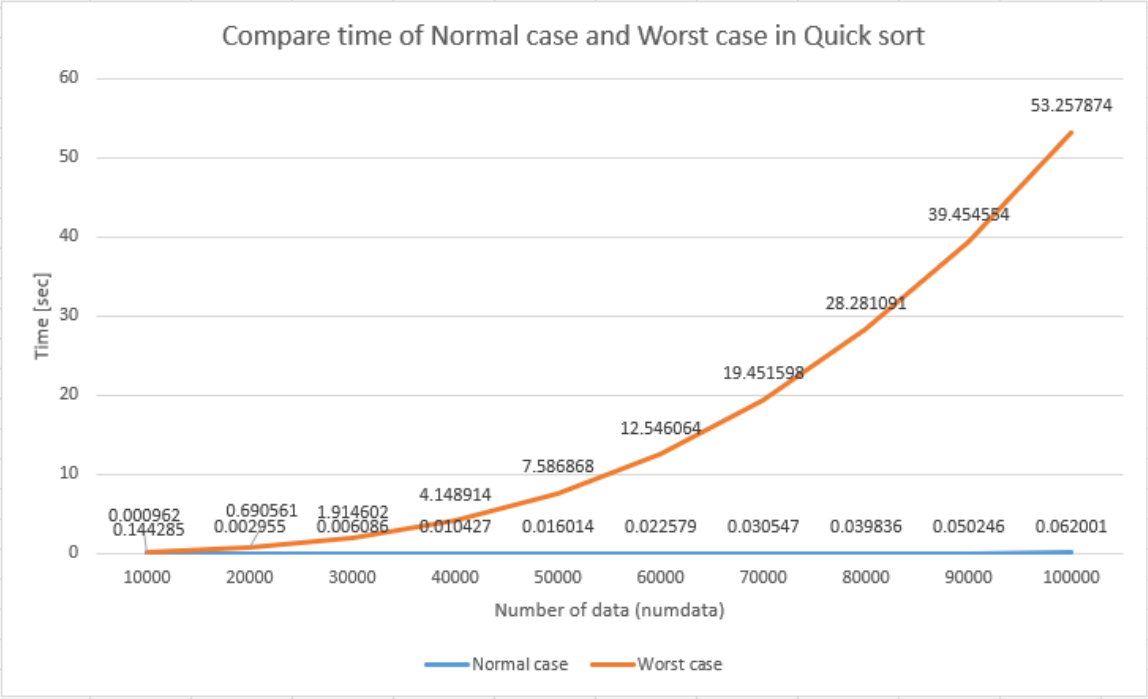
Line 134: 普通ケースの pivot を作る。

実行結果：

```
azalea02:~ s2213025$ ./sort_collection
numdata 10000
data created!numdata = 10000: time = 0.000962[sec]
numdata 20000
data created!numdata = 20000: time = 0.002955[sec]
numdata 30000
data created!numdata = 30000: time = 0.006086[sec]
numdata 40000
data created!numdata = 40000: time = 0.010427[sec]
numdata 50000
data created!numdata = 50000: time = 0.016014[sec]
numdata 60000
data created!numdata = 60000: time = 0.022579[sec]
numdata 70000
data created!numdata = 70000: time = 0.030547[sec]
numdata 80000
data created!numdata = 80000: time = 0.039836[sec]
numdata 90000
data created!numdata = 90000: time = 0.050246[sec]
numdata 100000
data created!numdata = 100000: time = 0.062001[sec]
```

グラフ：

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Normal case	0.000962	0.002955	0.006086	0.010427	0.016014	0.022579	0.030547	0.039836	0.050246	0.062001
Worst case	0.144285	0.690561	1.914602	4.148914	7.586868	12.54606	19.4516	28.28109	39.45455	53.25787



分析：データ量が多いほど、普通ケースと最悪ケースの処理時間は大きく違う。データ量が 100000 個があっても、普通ケースの処理時間は 0.1 秒にも及ばないことがわかる。その一方で、最悪ケースは、たかが 30000 個のデータで約 2 秒がかかる。結局、最悪ケースでの 100000 個の場合は約 1 分がかかる。結論としては、普通ケースの計算量 O(n logn)は、最悪ケースの計算量 $O(n^2)$より本当に小さいことがわかる。

4. 1～3 の実験結果から，クイックソートとヒープソートの性能について，比較，考察せよ．

クイックソートの処理は流石に早い。ほとんどの場合、ヒープソートの処理時間と比べると、約 3 分の 2 であることがわかる。しかし、最悪ケースであれば、処理できないほど計算量が大きい。一方、ヒープソートはすご

く安定性で、計算量がいつも O(n logn)であることがわかる。これで、クイックソートを使うとき、最悪ケースを避けるために、pivot を端っこにおかないといういいことがわかる。結論として、データが散らかっている時に、クイックソートを使ったほうがいい。一方、全部のデータがソートされていないけど、ほとんどソートされた時にヒープソートを使ったほうがいい。

発展課題

課題 5-3

5-3-1 実装の仕様

sort_collection に，基数ソート `void radix_sort(int a[], int n, int k)` を追加せよ．実装には，教科書 115 ~ 121 ページを参考にすること． この基数ソートは，10 進数の各桁にバケットソートを適用し，整数を整列するアルゴリズムであり，配列 a の各要素 a[i] は, $0 \leq a[i] < 10^k$ とする．

- 以下の要件を全て満たすことを確認すること．
    - 整数 143, 322, 246, 755, 123, 563, 514, 522 を要素とする配列に対して動作を確認すること．
    - 各桁の処理の後のバケットの内容を表示し，確認すること．

5-3-2 実装コードおよび実装コードの説明

まず、バケットソート関数を作る：

```
146 // Functions for Radix sort
147 /************************************/
148 void bucket_sort (int a[], int n, int digit) {
149       int i = 0;
150       int total_bucket = 10;
151       int bucket_index = 0;
152       int bucket[total_bucket][n];
153
154     int bucket_sizes[total_bucket];
155     for (i = 0; i < total_bucket; i++) {
156         bucket_sizes[i] = 0; //bucket_sizes = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, each number shows how many
    element in each bucket, starts from bucket 0 and ends with bucket 10
157     }
158
159     //move from array a to sub-array bucket
160       for (i = 0; i < n; i++) {
161             bucket_index = (a[i] % (int)pow(10, digit + 1) / (int)pow(10, digit)) %  total_bucket; //maybe we
    dont need this "%  total_bucket" part
162             bucket[bucket_index][bucket_sizes[bucket_index]] = a[i];
163             bucket_sizes[bucket_index]++;
```

Line 148: bucket_sort 関数に 3 つの引数を渡す。a は array、n は array の要素、digit は radix_sort 関数から渡す数の桁。

Line 152: 2-dimensional array “bucket”を作る。bucket のイメージはこのように：

Bucket 0:	要素	要素	要素	要素	要素
Bucket 1:	要素	要素	要素	要素	要素
Bucket 2:	要素	要素	要素	要素	要素

Line 156: array bucket_sizes は、bucket_sizes の要素に相当する bucket の中にいくら要素があるのかを表す。例えば

bucket_sizes = [0, 0, 0, 1, 0, 2]の時、bucket 3 に 1 つの要素が入っていたり、bucket 5 に 2 つの要素が入っていたり、残りの bucket は要素なしと意味している。

Line 160 ~ 163: 数の桁を 10 に割って、余りを k だったらをの数を bucket k に入れる。ここで k を bucket_index 変数で保存する。10 に

割るとき、余りが全部 10 つある(0 から 9 まで)ので、10 つの bucket を準備した(Line 150)。

```
169       // Loop to print all buckets
170       printf("Before sorting:");
171       printf("\n");
172       for (i = 0; i < total_bucket; i++){
173       for (int j = 0; j < n; j++){
174       printf("%d ", bucket[i][j]);
175       }
176       printf("\n");
177       }
178       //sort values inside pocket
179       for (bucket_index = 0; bucket_index < total_bucket; bucket_index++) {
180             //printf("bucket[bucket_index] before sorting is %d, bucket_sizes[bucket_index] before sorting is
    %d\n", bucket_index, bucket_sizes[bucket_index]);
181             q_sort(bucket[bucket_index], bucket_sizes[bucket_index]);
182       }
183
184       // Loop to print all buckets
185       printf("After sorting:");
186       printf("\n");
187       for (i = 0; i < total_bucket; i++){
188       for (int j = 0; j < n; j++){
189       printf("%d ", bucket[i][j]);
190       }
191       printf("\n");
192       }
```

ここで、クイックソートを流用して、各 bucket にある要素をソートする。注意してほしいのは、bucket にある全部の要素をソートするわけではなく、ただ array a から取って入れた要素だけをソートする。なぜかというと、2-dimensional array を作るときに、散らかっているデータがすでに入っているからだ。

Line 170 ~ 177: データがソートされていない状態の bucket を表す。

Line 184 ~ 192: データがソートされた状態の bucket を表す。

```
195       //move back values from bucket to array a
196       for (i = 0, bucket_index = 0; bucket_index < total_bucket; bucket_index++) {
197       int value_inside_bucket = 0;
198       //bucket_sizes[bucket_index] != 0 mean that bucket has elements in it
199       //value_inside_bucket < bucket_sizes[bucket_index] and value_inside_bucket++ means only take the number
    that we put into bucket
200       while (bucket_sizes[bucket_index] != 0 && value_inside_bucket < bucket_sizes[bucket_index] && i < n) {
201             a[i] = bucket[bucket_index][value_inside_bucket];
202             i++;
203             value_inside_bucket++;
204       }
205       }
206 }
```

Line 196 ~ 205: bucket にあるデータをソートしてから、a array に戻す。

基数ソート関数：

```
208 void radix_sort(int a[], int n, int k) {
209         //k is the maximum digits
210         int digit = 0;
211         for (digit; digit < k; digit++) {
212                 bucket_sort (a, n, digit);
213                 printf("at digit %d, array a is ", digit);
214                 for (int i = 0; i < n; i++) {
215                         printf("%d ", a[i]);
216                 }
217                 printf("\n");
218         }
219 }
```

各桁に対して、bucket_sort 関数を使ってソートする。一番右の桁から一番左の桁へ。

main 関数：

```
11 int main(int argc, char *argv[]) {
12   if (argc != 1) {
13     int numdata = atoi(argv[1]);  // set numdata with cmd. argument
14     int *array = (int*)malloc(sizeof(int) * numdata);
15     int i;
16     printf("Enter %d integers\n", numdata);
17     for (i = 0; i < numdata; i++) {
18       scanf("%d", &array[i]);  // enter integers
19    }
20     radix_sort(array, numdata, 3);
21     printf("sorting result\n");
22     display(array, numdata);
23     printf("# of comparisons: %lu\n", compare_count);
24     free(array);
25   } else {
26     int numdata;
```

Line 20 を radix_sort に変更する。radix_sort の第 3 引数は、ソートしたいデータの中で、桁が一番多い数の桁量である。

5-3-3 実行結果

```
azalea02:~ s2213025$ ./sort_collection 8
Enter 8 integers
143
322
246
755
123
563
514
522
```

```
Before sorting:
0 0 8064 65535 0 16 1547538984 32767
0 32 0 0 0 0 832 832
322 522 61765110 1 0 0 832 832
143 123 563 832 832 832 1999743716 32561
514 32561 656 0 0 0 1999250886 32561
755 0 1999255438 32561 1586614272 21989 0 0
246 0 0 0 135168 0 -2119577088 1416060190
3480 0 2000788608 32561 640 0 -72 -1
656 0 41 0 39 939524096 -2119577088 1416060190
0 0 0 0 1547452200 32767 1571458864 21989
After sorting:
0 0 8064 65535 0 16 1547538984 32767
0 32 0 0 0 0 832 832
322 522 61765110 1 0 0 832 832
123 143 563 832 832 832 1999743716 32561
514 32561 656 0 0 0 1999250886 32561
755 0 1999255438 32561 1586614272 21989 0 0
246 0 0 0 135168 0 -2119577088 1416060190
3480 0 2000788608 32561 640 0 -72 -1
656 0 41 0 39 939524096 -2119577088 1416060190
0 0 0 0 1547452200 32767 1571458864 21989
at digit 0, array a is 322 522 123 143 563 514 755 246
```

Before sorting の下には、ソートされていない bucket の状態である。bucket 2 には 2 個のデータ、bucket 3 には 3 個のデータ、bucket 4 には 1 個のデータ、bucket 5 には 1 個のデータ、bucket 6 には 1 個のデータが入っていて(入っているデータは各バケットの先頭に並んでいる)、残りの bucket は 2-dimensional array を作ったときに入った、「関係なく散らかっているデータ」である。そういうデータは気にしなくてもいい。

After sorting のところ、array a からバケットに入ったデータがソートされたことがわかる。ほかの関係のないデータはそのままでほっておいている。

Digit 0 (一番右の桁)によってソートした後、array a の状態は”at digit 0, array a is”のように表れている。

同じ動作で、digit 1 と digit 2 によってソートする。

Digit 1:

```
Before sorting:
0 0 8064 65535 0 16 1547538984 32767
514 32 0 0 0 0 832 832
322 522 123 1 0 0 832 832
123 143 563 832 832 832 1999743716 32561
143 246 656 0 0 0 1999250886 32561
755 0 1999255438 32561 1586614272 21989 0 0
563 0 0 0 135168 0 -2119577088 1416060190
3480 909390336 -2119577088 1416060190 640 0 0 0
1547451776 32767 1547452200 32767 1571458864 21989 1571470688 21989
2002382912 32561 1998981023 32561 16 48 1999154925 32561
After sorting:
0 0 8064 65535 0 16 1547538984 32767
514 32 0 0 0 0 832 832
123 322 522 1 0 0 832 832
123 143 563 832 832 832 1999743716 32561
143 246 656 0 0 0 1999250886 32561
755 0 1999255438 32561 1586614272 21989 0 0
563 0 0 0 135168 0 -2119577088 1416060190
3480 909390336 -2119577088 1416060190 640 0 0 0
1547451776 32767 1547452200 32767 1571458864 21989 1571470688 21989
2002382912 32561 1998981023 32561 16 48 1999154925 32561
at digit 1, array a is 514 123 322 522 143 246 755 563
```

Digit 2:

```
Before sorting:
0 0 8064 65535 0 16 1547538984 32767
123 143 0 0 0 0 832 832
246 322 522 1 0 0 832 832
322 143 563 832 832 832 1999743716 32561
143 246 656 0 0 0 1999250886 32561
514 522 563 32561 1586614272 21989 0 0
563 0 0 0 135168 0 -2119577088 1416060190
755 859190528 -2119577088 1416060190 640 0 0 0
1547451776 32767 1547452200 32767 1571458864 21989 1571470688 21989
2002382912 32561 1998981023 32561 16 48 1999154925 32561
After sorting:
0 0 8064 65535 0 16 1547538984 32767
123 143 0 0 0 0 832 832
246 322 522 1 0 0 832 832
322 143 563 832 832 832 1999743716 32561
143 246 656 0 0 0 1999250886 32561
514 522 563 32561 1586614272 21989 0 0
563 0 0 0 135168 0 -2119577088 1416060190
755 859190528 -2119577088 1416060190 640 0 0 0
1547451776 32767 1547452200 32767 1571458864 21989 1571470688 21989
2002382912 32561 1998981023 32561 16 48 1999154925 32561
at digit 2, array a is 123 143 246 322 514 522 563 755
```

Digit 2 によってデータをソートした後、array a のソートが完了。

```
sorting result
123 143 246 322 514 522 563 755
# of comparisons: 11
```

明らかにソートが成功したことがわかる。

感想

今回、色々なソート方法に触れた。n によって $O(n \log n)$ と $O(n^2)$ の違いが著しくなることも分かった。