EE3980 Algorithms

Homework 7. Linear Sort

By 105061212 王家駿

1. Introduction

In the previous homework, the best time complexity of comparison-based sorting we got before is $O(n * \log n)$. In this homework, we try to use a non-comparison-based sorting algorithm to reach the time complexity of $O(n)$, which is linear. And then we compare the average time consumed for sorting strings by using linear sort and heap sort, respectively.

2. Implementation

For implementation, we first read in all the input data, and store them in a 2D array. Then, we use the radix sort, a non-comparison-based sorting algorithm, to sort the input strings in lexicographical order, and repeat the sorting for R times. Moreover, we also implement the heap sort from homework 2 for R times and get the average running time. Finally, we print out the array after sorted as well as the CPU time consumed for the two algorithms. The details are given below.

2.1. Method of storing data

In order to store the input strings, we use a 2D array of character. The size of the array is N*14, since there are N input strings and at most 14 characters in a string.

Now we examine the English characters in UTF-8 encoding rules. The lowercase letters are encoded form 97 for “a” to 122 for “z”, so we put the characters which are not belong to the string content, i.e. after ‘\0’, to be “`”, whose UTF-8 code is 96. After this step, we now have 14 characters for a string, and they are at the range from 96 to 122 in UTF-8 code.

The reason why we let the rest of the characters be “`” is that for the two strings with the same contents at the begin but one has extra letters, for example, “bit” and “bitwise”, the lexicographical order must be “bit” before “bitwise”. So if we put the “`” character at the end of “bit”, like “bit````”, the two words have the same number of digits and it matches the alphabetical order in UTF-8. So, we could easily use it in the radix sort since the number of digits are same.

Also, we use a global variable to record the maximum string length. For the characters after the maximum string length, all of them are not the content of strings. That is, the characters after the maximum string length are all “`”. Thus, when we use the radix sort, we could neglect the digits after the maximum string length, since they are all the same kind and no need to be sorted.

2.2. Radix Sort

```
1.  Algorithm RadixSort(A, d)
2.  {
3.      for i := max_string_length to 1 step -1 do {
```

```
4.          Sort array A by digit i using CountingSort;
5.      }
6.  }
```

With the input data stored in the array, we use radix sort to sort it in lexicographical order. Radix sort applies sorting algorithm with respect to the certain digit from the LSB to MSB, and the sorting algorithm has to be stable. Thus, when we sort the higher digit, the lower digit we have sorted would not change its order if two of the higher digits are same. That is, the two string would arrange in the alphabetical order.

In our implementation, we use the counting sort for the stable sorting algorithm. And then we seem each character as a digit to be sorted, so we apply counting sort from the maximum string length (LSB) to the first character in the string (MSB). The result would be all the string arranged in lexicographical order.

2.3. Counting Sort

```
1.  Algorithm CountingSort(A,B,n,k)
2.  {
3.      // Initialize C to all 0
4.      for i := 1 to k do {
5.          C[i] := 0;
6.      }
7.
8.      // Count number elements in C[A[i]]
9.      for i := 1 to n do {
10.         C[A[i]] := C[A[i]] + 1;
11.     }
```

```
12.
13.     // C[i] is the accumulate number of elements
14.     for i := 1 to k do {
15.         C[i] := C[i] +C[i-1];
16.     }
17.
18.     // Store sorted order in array B
19.     for i := n to 1 step -1 do {
20.         B[C[A[i]]] := A[i];
21.         C[A[i]] := C[A[i]]-1;
22.     }
23. }
```

We use the counting sort as the stable sorting algorithm in the radix sort. First, we construct an array B with size N to store the sorting result, and an array C with size 27 to store the number of each element.

At the next step, we iterate through the array, and record the number of times each letter appears in C. Since we have only the lowercase letters and "`" we added, we need 26 + 1 = 27 space to store the number of each letter in A. We use UTF-8 code to decide where we should record. Since we have known that the characters in the strings only have 96 to 122 in UTF-8 code. We stores the character with code 96 in C[0], and 91 in C[1], and so on. Thus, we could get all of the 27 indexes in the array C to record the appearing times of each letter respectively.

Then, we make the C array be the accumulation of the number of letters by adding the value from C[1] to C[i-1]. Thus, after the additions through the whole array, the content in the array C would be (not strictly) increasing.

At the final step, because we had known how many times each letter appears in the array, the only thing we have to do is place the string to array B directly. For a letter ltr, we would find C[ltr] to get the accumulation of index, and the index is where we should put the string in B. And then minus the accumulation by one, which indicates that the next string with the same letter would be place before the previous string. We keep executing this step until the iteration goes to the end, and it also means every component in array A was arranged well.

Finally, we copy the content in array B to A and finish the sorting of the digit.

2.4. Complexity analysis

For the time complexity, first in the radix sort function, the iteration at line 3 would run at most the time of maximum string length, which is no greater than 14. And at each iteration step, the counting would run for one time. Thus, the time complexity of radix sort is $14 * \text{time complexity of counting sort}$.

In the counting sort function, the iteration at line 4 and 14 would run for k times, where k is the number of possible values = 27. The iteration at line 9 and 19 would

run for n times. And there are only assignments and additions in the four loops. Thus, the time complexity is $O(4 * \max(n, k)) = O(4 * n) = O(n)$ since the data size n is often larger than k. Hence, the overall time complexity is $14 * O(n) = O(n)$, which is linear.

For the space complexity, we need two extra arrays B and C with size n and k for each counting sort, and the counting sort would execute for at most 14 times. Thus, the space complexity is $O\big(14 * (n + k)\big) = O(14 * 2n) = O(n)$.

Time complexity: $O(n)$

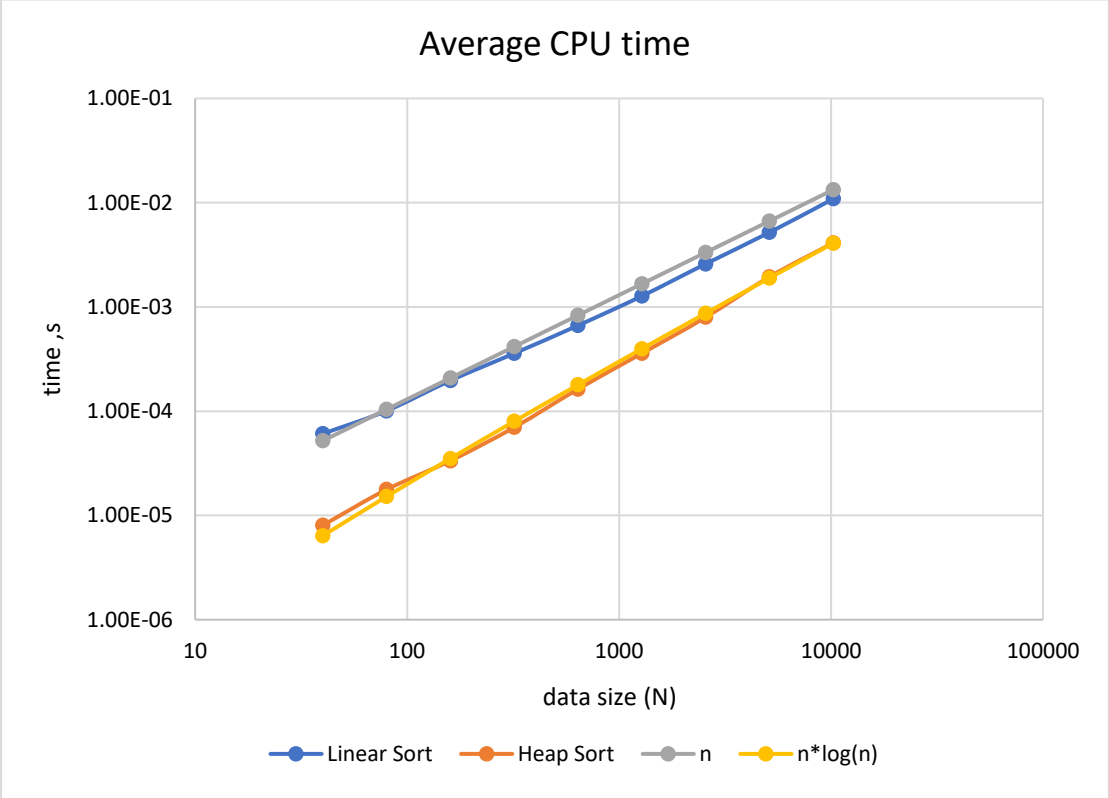
Space complexity: $O(n)$

3. Executing results

For repeating 500 times, we run the testing data from wl1.dat to wl9.dat with different input data size, and record the average CPU time used.

Data size (*N*)	Linear Sort	Heap Sort
40	60.74 μ s	8.036 μ s
80	100.8 μ s	17.72 μ s
160	196.7 μ s	33.41 μ s
320	358.5 μ s	69.90 μ s
640	663.8 μ s	163.0 μ s
1280	1.271ms	358.4 μ s

2560	2.588ms	795.4 μ s
5120	5.187ms	1.944ms
10240	10.88ms	4.119ms



4. Result analysis and conclusion

From the graph, we could observe that the radix sort has linear trend, and the heap sort has the trend of $O(n * \log(n))$, which are same as our estimation.

However, for the case N<10000, the radix sort is slower than the heap sort in spite of the face that it has the lower time complexity. It may be the effect that at the each counting sort step in the radix sort, we have to construct two new arrays, and apply assignment through the whole array at least four times. These steps make the radix

sort slower than the heap sort in a small input data size. Yet, since the heap sort has a faster growing trend, it may become slower with regard to the radix sort if the input size keeps growing up.

We get the time complexity of $O(n)$ by neglecting the effect of string length. That is, assume the string length is far smaller than input data size. However, if the string length is close to, or greater than the data size, the time complexity would become $O(n * \text{string length}) = O(n^2)$. So, it the algorithm would become slower than the non-comparison-based sorting algorithm we used. Thus, the radix sort is only suitable when a small string length.

For the space complexity, the radix sort is $O(n)$, while the heap sort is $O(1)$ since it is in-place sorting, so the radix sort would take more space than the latter. In the end, we have to choose the algorithm carefully to be suitable for different problems.

Score: 92

o. See return.

[Coding] hw07.c spelling errors: initailize(1)

[Coding] can be more efficient.

[Coding] can be improved.

hw07.c

```c
/* EE3980 HW07 Linear Sort
 * 105061212, 王家駿
 * 2019/04/20
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

int N;                                  // data size
int max_size;                           // maximum string length
int R = 500;                            // repeat times
char** data;                            // input data
char** A;                               // array to be sorted

void readInput(void);                   // read all input
double GetTime(void);                   // get local time in seconds
void copyArray(char** data, char** A);  // copy data to array A
void radixSort(char** A);               // linear radix sort
void countingSort(char** A, int d);     // stable counting sort
void HeapSort(char** list, int n);      // in-place heap sort
void Heapify(char** list, int root, int n); // rearrange to form max heap
void printResult();                     // print out the result

int main(void)
{
    int i;                              // loop index
    double t0, t1, t2, t_radix, t_heap;     // CPU time

    readInput();                        // read all input

    t0 = GetTime();                     // get current CPU time

    for (i = 1; i <= R; i++) {          // repeat R times
        copyArray(data, A);
        radixSort(A);                   // linear sort
        if (i == 1) printResult();      // print out the result
    }

    t1 = GetTime();                     // get current CPU time

    for (i = 1; i <= R; i++) {          // repeat R times
        copyArray(data, A);
        HeapSort(A, N);                 // in-place heap sort
    }

    t2 = GetTime();                     // get current CPU time
```

```
49
50      // calculate average time consumed
51      t_radix = (t1 - t0) / R;
52      t_heap = (t2 - t1) / R;
53
54      // print out time results
55      printf("N = %d\n", N);
56      printf("  Linear Sort CPU time = %e seconds\n", t_radix);
57      printf("  Heap Sort CPU time = %e seconds\n", t_heap);
58
59      // free dynamic memories
60      free(A);
61      free(data);
62
63      return 0;
64  }
65
66  void readInput(void)                        // read all input
67  {
68      int i, j;                               // loop index
69      int check;                              // check for the end of string
70      char s[15];                             // temporary string
71
72      scanf("%d", &N);                        // data size
73
74      // allocate dynamic memories
75      data = (char**)malloc(sizeof(char*) * (N+1));
76      A = (char**)malloc(sizeof(char*) * (N+1));
77
78      max_size = 0;                           // initailize maximum string size
79
80      for (i = 1; i <= N; i++) {              // for all data
81          // allocate dynamic memories
82          data[i] = (char*)malloc(sizeof(char) * 15);
83          A[i] = (char*)malloc(sizeof(char) * 15);
84
85          scanf("%s", s);                     // input string
86
87          // find for the maximum string length
88          if(strlen(s) > max_size) max_size = strlen(s);
            if (strlen(s) > max_size) max_size = strlen(s); // space
89
90          check = 0;
91          for (j = 0; j <= 14; j++) {
92              // mark characters after the string to be '`'
93              if (s[j] == '\0') check = 1;
94              if (check == 1) data[i][j] = '`';
95              else data[i][j] = s[j];
96          }
97      }
```

```
 98 }
 99
100 double GetTime(void)                          // get local time in seconds
101 {
102     struct timeval tv;                        // time interval structure
103
104     gettimeofday(&tv, NULL);                  // write local time into tv
105
106     // return time with microsecond
107     return tv.tv_sec + tv.tv_usec * 0.000001;
108 }
109
110 void copyArray(char** data, char** A)         // copy data to array A
111 {
112     int i;                                    // loop index
113
114     for(i = 1; i <= N; i++){
        for (i = 1; i <= N; i++) { // space
115         A[i] = data[i];
116         // assign the pointer A[i] to point to the data array
117     }
118 }
119
120 void radixSort(char** A)                      // linear radix sort
121 {
122     int i;                                    // loop index
123
124     for (i = max_size-1; i >= 0; i--) {       // for all characters
125         countingSort(A, i);                   // stable counting sort
126     }
127 }
128
129 void countingSort(char** A, int d)            // stable counting sort
130 {
131     int i;                                    // loop index
132     char** B;                                 // array to store result
133     int* C;                                   // array to count the number
134     int t;                                    // temporary index
135
136     // allocate dynamic memories
137     B = (char**)malloc(sizeof(char*) * (N+1));
138     for (i = 1; i <= N; i++) {
139         B[i] = (char*)malloc(sizeof(char) * 15);
140     }
141
142     // allocate dynamic memories
143     C = (int*)malloc(sizeof(int) * 27);
144
145     // initialize C
146     for (i = 0; i <= 26; i++) {
```

```
147         C[i] = 0;
148     }
149 
150     // count number of elements in C[A[i]]
151     for (i = 1; i <= N; i++) {
152         t = A[i][d] - 96;
153         C[t]++;
154     }
155 
156     // C[i] is the accumulate number of elements
157     for (i = 1; i <= 26; i++) {
158         C[i] = C[i] + C[i-1];
159     }
160 
161     // store sorted order in B
162     for (i = N; i >= 1; i--) {
163         t = A[i][d] - 96;
164         B[C[t]] = A[i];
165         C[t]--;
166     }
167 
168     // assign B to A
169     for (i = 1; i <= N; i++) {
170         A[i] = B[i];
171     }
172 
173     // free dynamic memories
174     free(B);
175     free(C);
176 }
177 
178 void HeapSort(char** list,int n)            // in-place heap sort
179 {
180     int i;                                  // loop index
181     char* tp;                               // temporary pointer for swap
182 
183     // heapify all the subtrees and be a max heap
184     for(i = n/2; i >= 1; i--){
185         Heapify(list, i, n);
186     }
187 
188     for(i = n; i >= 2; i--){                // repeat n-1 times
189         //swap the first node and the last node
190         tp = list[i];
191         list[i] = list[1];
192         list[1] = tp;
193 
194         // make list[1:i-1] be a max heap
195         Heapify(list, 1, i-1);
196     }
```

```
197 }
198
199 void Heapify(char** list, int root, int n)  // rearrange to form max heap
200 {
201     char* tp = list[root];                  // assign root value to tp
202     int j;                                  // loop index
203     int done = 0;                           // check if heapify ends
204
205     // j is the leftchild of root and keeps finding its leftchild
206     for(j = root*2; j <= n && done == 0; j = j*2){
207         if(j < n && strcmp(list[j], list[j+1]) < 0){
208             // j is the rchild if rchild > lchild
209             j++;
210         }
211         if(strcmp(tp, list[j]) > 0){
212             done = 1;                       // done if root > children
213         }
214         else{
215             list[j/2] = list[j];            // place child node to parent
216         }
217     }
218
219     list[j/2] = tp;                         // put root to the proper place
220 }
221
222 void printResult()                          // print out the result
223 {
224     int i, j;                               // loop index
225
226     for (i = 1; i <= N; i++) {              // for all data
227         for (j = 0; j < 15 && A[i][j] != '`'; j++) {
228             // print out results
229             printf("%c", A[i][j]);
230         }
231         printf("\n");
232     }
233 }
```