EE3980 Algorithms

HW6 Linear Sort

104061212 馮立俞

2018/4/15

Introduction

In this assignment, we're asked to sort a list of words using algorithm of linear time complexity. However, compared with previous sorting assignments. The words to be sorted share two properties, i.e.

1. All words consist of lower-case letters only.

2. The maximum number of letters of the words is 14.

Approach

Since the characters are all lower-case, which means there's only 27(a ~z and '\0') possible value for each letter in a word string. Plus, the words are no longer than 14 characters (limited length). In such case, a linear-complexity algorithm, radix sort, can be applied.

Radix Sort

```
Algorithm RadixSort(list, N) {
    For i = LSB to MSB do CountingSort(list, N, I);
}
```

RadixSort is simply calling CountingSort from Least Significant Bit (letter) to Most Significant Bit (letter).

It's noteworthy that as we use `scanf` to import data, the characters fill from index 0 (MSB). Then, if the word is shorter than the length of given array, remaining elements in array would be filled with '\0'.

Counting Sort

```
Algorithm CountingSort(list, N) {
    Init count = {  0, 0, …0  };
//count   has       k members, k is all possible   value in list
    for i = list[1] to list[N] do count[i]++;
    for i = 2 to k do count[i] += count[i - 1];
    for i = N to 1 do A[ --count[ list[i] ] ] = list[i];
    return A;
}
```

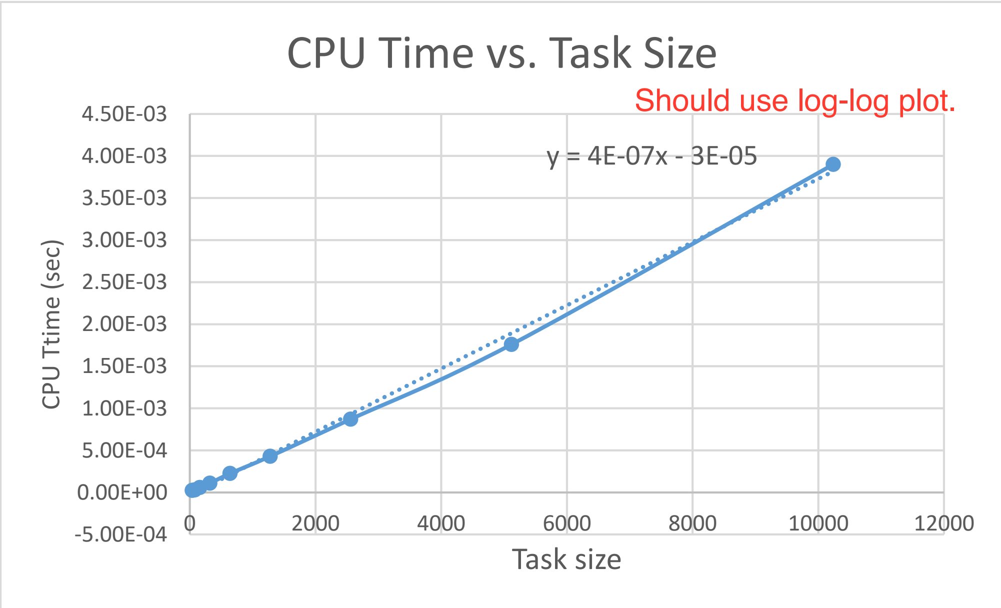
In the above algorithm, first we use `count` array to calculate how many members are less than or equal to the i-th possible value. Then, from back to top we place the elements in `list` to `A` according to the position indicated by `count` array.

As we can observe from the looping bounds, the time complexity is $O(n+k)$. Where n is the task size and k is the number of possible value in `list`. Additionally, we used another `A` and `count array,` so the space complexity is also $O(n+k)$. Therefore the complexity of RadixSort is $O(\ r(n+k)\ )$. r is the maximum length of word in wordlist to be sorted.

Results and analysis

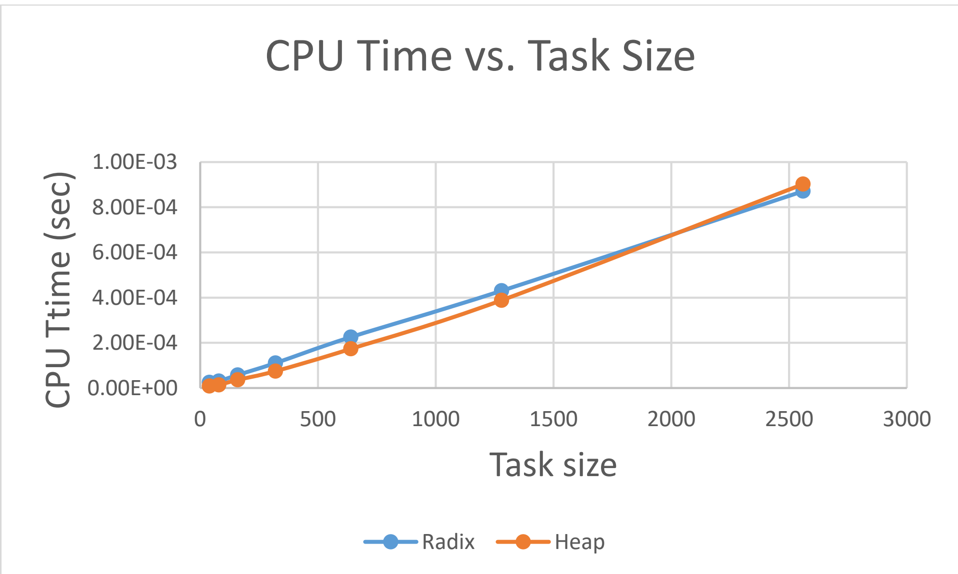
Table. CPU Time (in sec) w.r.t. task size

Task Size	40	80	160	320	640	1280	2560	5120	10240
CPU Time	2.43E-05	3.03E-05	5.75E-05	1.10E-04	2.25E-04	4.30E-04	8.71E-04	1.76E-03	3.90E-03



It's obvious in above chart that when r, k << n, RadixSort has linear time complexity.

However, we can take HeapSort from HW2 to compare together.



Though the theoretical time complexity is different, their actual execution time didn't differ a lot when sorting the test cases of this assignment.

Observations and Conclusion

1. RadixSort / CountingSort are of great use when the data to be sorted have limited possible value. (r, k << n)
2. Lower time complexity does not always guarantee shorter execution time.

```c
/*************************************
  EE3980 Algorithms HW05 Linear Sort
  Li-Yu Feng 104061212
  Date:2018/4/15
**************************************/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include<stdbool.h>
#define LEN 14
char **A;


double GetTime(void);
void RadixSort(char **list,int n);
void CountingSort(char **list,int n, int loc);
void Heapify(char **list, int i, int n);
void HeapSort(char **list,int n);
double GetTime(void)
{
    struct timeval tv;
    gettimeofday(&tv,NULL);
    return tv.tv_sec+1e-6*tv.tv_usec;
}


void CountingSort(char **list, int n, int loc){
    int count[27];              //n:size loc:sorting radix location
    int i,index;
    char *temp;


    for (i = 0; i < 27; ++i)    //init count array
    {
        count[i] = 0;
    }


    for (i = 0; i < n; i++){                //count alphabet occuring times
        if(list[i][loc] == '\0')
            count[0]++;
        else count[ list[i][loc] - 96 ]++;
    }


    for(i = 1; i< 27; i++){         // calculate each alphabet's
        count[i] += count[i-1];     // dictionary order
```

Need comments here to explain the purpose of each function.

```
    }


    for(i = n-1; i >= 0; i-- ){
        temp = list[i];
        if (temp[loc] == '\0')   index = 0;     //deal with '\0' (ASCII 0)
        else index = temp[loc] - 96;            //lower case alphabet
        A[ count[ index ]- 1] = temp;           //put word in new array
        count[ index ]--;                       //update count array
    }

    for ( i = 0; i < n; ++i)                    //output current result
    {
        list[i] = A[i];
    }

}



void RadixSort(char **list,int n){
    int i;

    for (i = LEN-1; i >= 0; --i){
        CountingSort(list, n, i);
    }

}

void Heapify(char **list, int i, int n){
    int j = i*2;
    char *temp = list[i-1];
    bool done = false;

    while(j<=n && !done){
        if(j<n && strcmp(list[j-1],list[j+1-1]) < 0) j++;
        if(strcmp(temp,list[j-1] ) > 0 ) done = true;
        else{
            list[j/2-1] = list[j-1];
            j *= 2;
        }
        //printf("%d\n",j);
    }
    list[j/2-1] = temp;
}

void HeapSort(char **list,int n){
    char *temp;
    int i;

```

```
    for( i = n/2 ; i>0 ; i--)
        {Heapify(list,i,n);}
    for(i = n; i > 1; i-- ){
        temp = list[i-1];
        list[i-1] = list[0];
        list[0] = temp;
        Heapify(list,1,i-1);
    }
}

int main(){
    int i,j;
    int Nwords;
    double t;
    char **words,**temp;

    scanf("%d", &Nwords);
    A = (char **)malloc( Nwords * sizeof(char*) );
    temp = (char **)malloc( Nwords * sizeof(char*) );
    words = (char**)malloc(Nwords * sizeof(char*));         //
    for(i = 0; i < Nwords; i++)                             //
        words[i] = (char *)malloc( (LEN+1) * sizeof(char)); //


    for(i = 0; i < Nwords ; i++){                           //
        scanf("%s", words[i]);                              //scan words
    }
    t = GetTime();

    for(i =0; i<500;i++){                                   //sort 500 times
        for (j = 0; j < Nwords; ++j)
        {
            temp[j] = words[j];
        }
        RadixSort(temp, Nwords);
    }


    for(i = 0; i < Nwords ; i++){                       //print result
        printf("%d %s\n",i, temp[i]);
    }

    t = GetTime() - t;
    printf("%s:\nN=%d\nCPU time = %.3g seconds\n", "Linear Sort",
                        Nwords, t / 500.0);

    t = GetTime();

    for(i =0; i<500;i++){                                   //sort 500 times
        for (j = 0; j < Nwords; ++j)
```

```
        {
            temp[j] = words[j];
        }
        HeapSort(temp, Nwords);
    }


    for(i = 0; i < Nwords ; i++){                       //print result
        printf("%d %s\n",i, temp[i]);
    }

    t = GetTime() - t;
    printf("%s:\nN=%d\nCPU time = %.3g seconds\n", "Heap Sort",
                        Nwords, t / 500.0);


    return 0;

}
```

Score: 91

[Table] can be better presented.

[Figure] can be better presented.

- Use log-log scale.
- Can compare two algorithms on the same plot.