EE3980 Algorithm

HW2 HW02 title?

106061225 楊宇羲

1. Problem description:

In this time's assignment, we are required to carry out and analyze performance and complexity of ? a couple approaches of searching random data, which means to send a target and an array into a function and return the value of the index that contains the target, or return -1 if target isn't found among the array. The following methods are going to be implemented, which are linear search, search 2 algorithm, odd-even search and randomized odd-even search, respectively.

2. Approach

There are 9 lists of strings same as lists provided in homework 1for us to perform analysis. With the input size ranging from 10 to 2560, we can examine each algorithm with a variety of size of input. In this assignment, we will also need to call GetTime() function in order to measure CPU time. We will call GetTime() function first to reset the timer before the array is being initialized to the original list Really?, then each searching method is carried out for R times. After the searching is finished we call the GetTime() function again to obtain finished time. The overall operation will be like this:

What is this?

```
algorithmRandomDataSearch
{
        scan array;   ???
        // do the following 4 steps same with other 3 algorithms average cases
        GetTime();
        LinearSearch(average case);
        GetTime();
        Calculate CPU time;

        // do the following 4 steps same with other 3 algorithms worstcases
        GetTime();
        LinearSearch(worst case);
        GetTime();
        Calculate CPU time;
}
```

3. Analysis of each searching algorithm:

a) linear search:

The most directive searching method. We start from the first index and then look for the next one every time until the last index of the array. The pseudo code will be like this:

```
Algorithm search (target, array, size)
{
	for (i:=1 to size step 1)
	{
		if (array[i] == target) return i;
	}
	return -1;
}
```

Should follow the form in the handout.

The space complexity will be n + 1 + 1 = n + 2 for the array, array size and loop variable. The time complexity will be O(n) in either average case or worst case since we will only have to search the array for at most one time, making worst case's total steps twice the size of array (increment of index and comparing string) . In average case we will implement each index for R times , making total steps 2*R*(1+2+…+n)=R*n*(n+1) times and **(n+1)** times in average, while in the worst case, we will set the target to the worst case and let the algorithm implement for 2*10*R times, which leads to total steps of 2*10*R*n times and **2n** times in average. In theory, the outcome of average case and worst case will have a ratio of 1/2. For the worst case, there is no doubt that the last index will be the slowest one to be searched.

What is it?

b) Search 2 algorithm:

Not clear.

The time complexity and space complexity is same as (a since there are no extra variable and extra steps. Being almost the same as the linear search algorithm but instead did something different inside the algorithm, we will also start from the first index, but this time we will compare the current index and the next one at the same iteration. After comparing, if there is the necessity to move on, we make 2 steps further, so the pseudo code is as the provided:

```
Algorithm search2 (target, array, size)
{
        for (i:=1 to size step 2)
        {
                if (array[i] == target) return i;
                if (array[i+1] ==target) return i+1;
        }
        return -1;
}
```

When comparing with the previous algorithm, although there are less incrementing times in index, but we will have to compare 2 times in an iteration, making it almost the same as the linear search method. Thus, the time complexity is also O(n) and there is also a ratio of 1/2 between execute step of average case and worst case, which is also **n+1** times and **2n** times. In the worst case issue, since the last iteration will compare the last two index, so I will assign the last index for the algorithm to carry out.

c) Odd-even search:

The space complexity is same as the previous methods, so as the time complexity since the average is also O(n). Although there is an Odd-even sort in assignment 1, but in searching, odd-even search are just also a kind of variation of linear searching which do a step 2 searching on odd index then a step 2 searching on even index. The total complexity will also be O(n). When deciding the worst case index, we should consider the size of array — if the size is odd then we will have to choose the index in front of the last one. In this assignment's case the size is all in even so I will ignore the condition and choose the last index right away.

```
Algorithm OEsearch (target, array, size)
{
        for (i:=1 to size step 2)
        {
                if (array[i] == target) return i;
        }
        for (i:=2 to size step 2)
        {
                if (array[i] == target) return i;
        }
        return -1;
}
```

We can clearly see this algorithm as the same with linear search only with the order of index being different, which is 1, 2, 3, …, n in the former one and 1, 3, 5, …, n-2, n-4, n in the latter one, bringing out the expected result of **n+1** times and **2n** times in average case and worst case, respectively.

d) Randomized odd-even search:

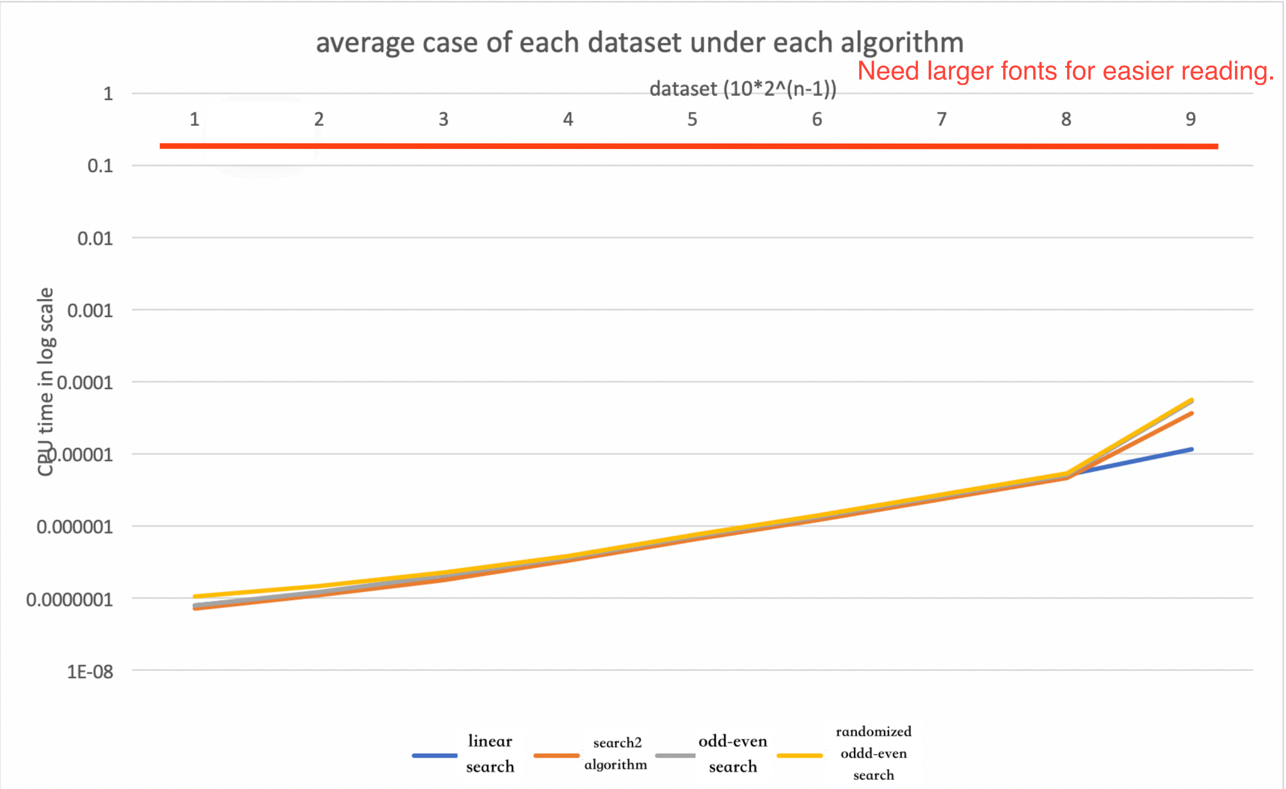
The space complexity will be n+3 since there will be a random number, but theme complexity is also O(n). Being almost the same as the former one, but we are going to add some extra step in the beginning of the function, which is to decide whether we are going to search the even part of the odd part first and two if condition to decide. As the result, there are going to be almost the same result as in the other 3 algorithm, and the same complexity. However, deciding the worst case index may be trivial in this algorithm since the order is chosen randomly, so I will just assign the last index for it to run.

How is it trivial?

```
Algorithm ROEsearch (target, array, size)
{
    decide a random number in set{0, 1};
    if (number == 0)
    {
        for (i:=1 to size step 2)
        {
            if (array[i] == target) return i;
        }
        for (i:=2 to size step 2)
        {
            if (array[i] == target) return i;
        }
    }
    else
    {
        for (i:=2 to size step 2)
        {
            if (array[i] == target) return i;
        }
        for (i:=1 to size step 2)
        {
            if (array[i] == target) return i;
        }
    }
    return -1;
```

}

If the number is 0, then we will search from the odd part first. We will search from the even part first if the number is 1. Adding some extra instruction may lead to slight extra execution time. However, the difference might be insignificant.



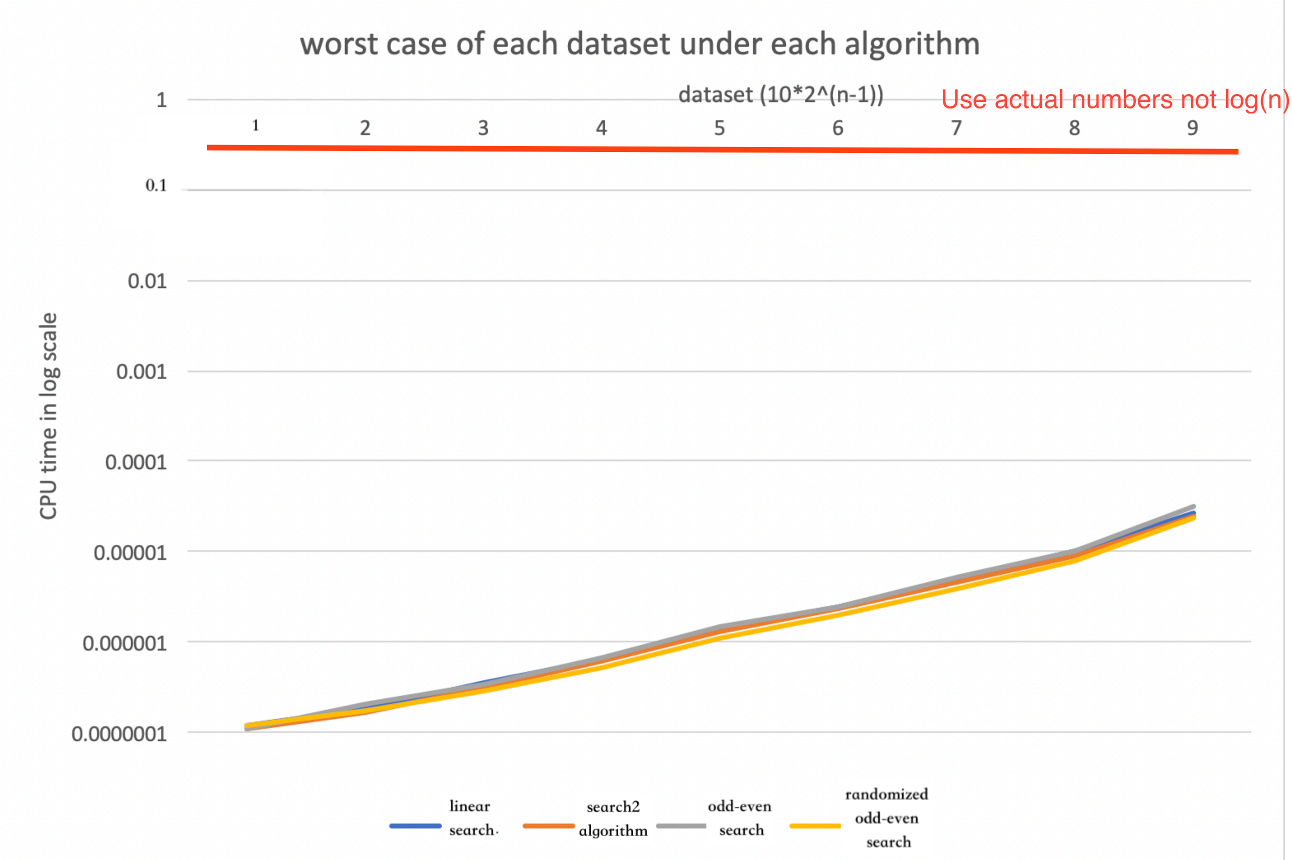


Table of each data For what? Unit?

average case	linear search	search 2	odd-even search	randomized odd-even search
1	7.96E-08	7.20E-08	7.96E-08	1.04E-07
2	1.19E-07	1.12E-07	1.21E-07	1.47E-07
3	1.97E-07	1.81E-07	2.03E-07	2.24E-07
4	3.66E-07	3.31E-07	3.70E-07	3.96E-07
5	7.10E-07	6.44E-07	7.19E-07	7.42E-07
6	1.34E-06	1.21E-06	1.36E-06	1.39E-06
7	2.64E-06	2.36E-06	2.67E-06	2.69E-06
8	5.22E-06	4.68E-06	5.25E-06	5.28E-06
9	1.14E-05	3.60E-05	5.38E-05	5.68E-05
worst case	linear search	search 2	odd-even search	randomized odd-even search
1	1.18E-07	1.10E-07	1.10E-07	1.22E-07
2	1.84E-07	1.66E-07	2.06E-07	1.78E-07
3	3.50E-07	3.16E-07	3.40E-07	2.92E-07
4	6.40E-07	6.12E-07	6.56E-07	5.14E-07
5	1.43E-06	1.29E-06	1.47E-06	1.11E-06
6	2.42E-06	2.37E-06	2.47E-06	1.98E-06
7	5.01E-06	4.63E-06	5.11E-06	3.85E-06
8	1.02E-05	9.17E-06	1.02E-05	7.76E-06
9	2.70E-05	2.41E-05	3.10E-05	2.37E-05

Worst-case is faster than average case?

5. Observation:

According the theory we have analyzed in the previous part, CPU time between each dataset should be ??? exponentially increased, which is mostly corresponded, as seen in the plot and the table. Furthermore, the relationship between average case and worst case in same dataset should be a ratio of 2, Not in your table. which is also mostly corresponded. Although there are some abrupt boost in some measuring, the graph still prove most of the relationship, including that CPU time between each algorithm doesn't have too much difference.

6. Conclusion:

At first, there are some frustrating moments including somehow increasing CPU time occurring in random dataset. However, by adjusting the size of R (repetition of each algorithm), I finally adjusted to a size big enough to avoid the occurrence of error, making every thing run as expected.

```
$ gcc hw02.c
$ a.out < s6.dat
Linear search average CPU time: 6.100059e-07
Search 2 algorithm average CPU time: 5.984619e-07
Odd-even search average CPU time: 6.603569e-07
Random odd-even search average CPU time: 6.357864e-07
Linear search worst-case CPU time: 1.195621e-06
Search 2 algorithm worst-case CPU time: 1.194811e-06
Odd-even search worst-case CPU time: 1.294994e-06
Random odd-even search worst-case CPU time: 9.344101e-07
```

Need to print out the size of the array, n.

Allocated strings are 1-byte too small.

---

score: 56.0

- Report format
  - Report title should take 3 lines (Course title, HW title, ID and name)
- Introduction
  - Introduction can still be strengthened
- Approach
  - Keys of your implementation should be described
  - Should argue each algorithm correctly solve the problem given.
  - Pseudo codes should follow that of the handout.
- Time/Space complexity
  - Worst-case time complexity should be clearly stated.
  - What is ROEsearch worst-case time complexity correct?
- Results
  - Both table and figures can be more readable.
  - Either table of figure is incorrect?
- Conclusion/observation
  - Can correlate your CPU times to the algorithm complexities
  - Can also compare the speed of different algorithms
- Program format can be improved

hw02.c

```
// EE3980 HW02 Random Data Searches
// 106061225, 楊宇羲
// 2021/3/20

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

double GetTime(void);                               // get local time
int Search(char *word, char **list,int n);          // Linear Search
int Search(char *word, char **list, int n);          // Linear Search
int Search2(char *word, char **list,int n);         // Search 2 Algorithm
int Search2(char *word, char **list, int n);         // Search 2 Algorithm
int OEsearch(char *word, char **list,int n);        // Odd-even search
int OEsearch(char *word, char **list, int n);        // Odd-even search
int ROEsearch(char *word, char **list,int n);       // Randomized odd-even search
int ROEsearch(char *word, char **list, int n);       // Randomized odd-even search

int main(void)
{

    int R = 500;                                    // time of repetitions
    int n;                                          // size of the array
    int i, j;                                       // for loop counting
    int result;                                     // check search result
    double t, t0, t1;                               // for CPU time counting

    scanf("%d", &n);                                // size of array

    char **a = (char**)malloc(n * sizeof(char*));   // the original array
    char buffer[100];                                   // buffer for storing

    for (i = 0; i < n; i++)
    {                                               // storing array to **a
        scanf("%s", buffer);
        a[i] = malloc(strlen(buffer));
a[i] is 1 byte too small.
        strcpy(a[i], buffer);
```

```
    }

    // linear search

    t0 = GetTime();                             // start counting time
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < R; j++)
        {
            result = Search(a[i], a, n);
        }
    }
    t1 = GetTime();                             // stop counting time
    t = (t1 - t0) / (n * R);                    // calculate CPU time
    printf("Linear search average CPU time: %e\n", t);

    // search 2

    t0 = GetTime();                             // start counting time
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < R; j++)
        {
            result = Search2(a[i], a, n);
        }
    }
    t1 = GetTime();                             // stop counting time
    t = (t1 - t0) / (n * R);                    // calculate CPU time
    printf("Search 2 algorithm average CPU time: %e\n", t);

    // odd even search

    t0 = GetTime();                             // start counting time
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < R; j++)
        {
            result = OEsearch(a[i], a, n);
        }
    }
    t1 = GetTime();                             // stop counting time
```

```
    t = (t1 - t0) / (n * R);                        // calculate CPU time
    printf("Odd-even search average CPU time: %e\n", t);

    // randomized odd even search

    t0 = GetTime();                                 // start counting time
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < R; j++)
        {
            result = ROEsearch(a[i], a, n);
        }
    }
    t1 = GetTime();                                 // stop counting time
    t = (t1 - t0) / (n * R);                        // calculate CPU time
    printf("Random odd-even search average CPU time: %e\n", t);

    t0 = GetTime();                                 // start counting time
    for (j = 0; j < 10 * R; j++)
    {
        result = Search(a[n-1], a, n);
        result = Search(a[n - 1], a, n);
    }
    t1 = GetTime();                                 // stop counting time
    t = (t1 - t0) / (10 * R);                       // calculate CPU time
    printf("Linear search worst-case CPU time: %e\n", t);

    t0 = GetTime();                                 // start counting time
    for (j = 0; j < 10 * R; j++)
    {
        result = Search2(a[n-1], a, n);
        result = Search2(a[n - 1], a, n);
    }

    t1 = GetTime();                                 // stop counting time
    t = (t1 - t0) / (10 * R);                       // calculate CPU time
    printf("Search 2 algorithm worst-case CPU time: %e\n", t);

    t0 = GetTime();                                 // start counting time
    for (j = 0; j < 10 * R; j++)
    {
```

```
        result = OEsearch(a[n - 1], a, n);
    }
    t1 = GetTime();                             // stop counting time
    t = (t1 - t0) / (10 * R);                   // calculate CPU time
    printf("Odd-even search worst-case CPU time: %e\n", t);

    t0 = GetTime();                             // start counting time
    for (j = 0; j < 10 * R; j++)
    {
        result = ROEsearch(a[n - 1], a, n);
    }
    t1 = GetTime();                            // stop counting time
    t = (t1 - t0) / (10 * R);                  // calculate CPU time
    printf("Random odd-even search worst-case CPU time: %e\n", t);

    return 0;
}

int Search(char *word, char **list,int n)
int Search(char *word, char **list, int n)
Need comments.
{

    int i;                              // for loop counting

    for (i = 0; i < n; i++)             // start from 0 to n with step 1
    {
        if (strcmp(list[i], word) == 0) return i;
    }
    return -1;
}

int Search2(char *word, char **list,int n)
int Search2(char *word, char **list, int n)
Need comments.
{

    int i;                              // for loop counting

    for (i = 0; i < n; i+=2)            // start from 0 to n with step 2
    for (i = 0; i < n; i += 2)            // start from 0 to n with step 2
```

```
    {
        if (strcmp(word, list[i]) == 0) return i;
        if (strcmp(word, list[i+1]) == 0) return i+1;
        if (strcmp(word, list[i + 1]) == 0) return i + 1;
    }
    return -1;
}

int OEsearch(char* word, char **list,int n)
int OEsearch(char* word, char **list, int n)
Need comments.
{

    int i;                              // for loop counting

    for (i = 0; i < n; i = i + 2)       // start from odd part with step 2
    {
        if (strcmp(word, list[i]) == 0) return i;
    }
    for (i = 1; i < n; i = i + 2)       // followed by even part with step 2
    {
        if (strcmp(word, list[i]) == 0) return i;
    }
    return -1;
}

int ROEsearch(char* word, char** list,int n)
int ROEsearch(char* word, char** list, int n)
Need comments.
{

    int i;                              // for loop counting
    int ran = rand() % 2;               // for deciding order

    if (ran == 0)                       // case 0: odd-even search
    {
        for (i = 0; i < n; i+=2)
        for (i = 0; i < n; i += 2)
        {
            if (strcmp(word, list[i]) == 0) return i;
        }
```

```
        for (i = 1; i < n; i+=2)
        for (i = 1; i < n; i += 2)
        {
            if (strcmp(word, list[i]) == 0) return i;
        }
    }
    else                                // else: even-odd search
    {
        for (i = 1; i < n; i+=2)
        for (i = 1; i < n; i += 2)
        {
            if (strcmp(word, list[i]) == 0) return i;
        }
        for (i = 0; i < n; i+=2)
        for (i = 0; i < n; i += 2)
        {
            if (strcmp(word, list[i]) == 0) return i;
        }
    }
    return -1;
}

double GetTime(void)
Need comments.
{

    struct timeval tv;

    gettimeofday(&tv, NULL);
    return tv.tv_sec + 1e-6 * tv.tv_usec;    // sec + micro sec
}

```