

EE3980 Algorithms

Hw02 Random Data Searches

106061146 陳兆廷

Introduction:

In this homework, I'll be analyzing, implementing and observing 3 searching algorithms: Linear search, Bidirectional search and Random-directional search. The goal of the algorithms is to find a specific data in an array. The input of them will be a string, and the output of them will be the position of the input in the array.

During the analyzing process, I'll be using table-counting method to calculate the time complexities of 3 algorithms. Furthermore, I'll find the worst-case conditions in 3 algorithms each accordingly. Before implementation on C code, I'll try to predict the result based on my analysis.

The implementation of 3 algorithms on C code mainly focus on the average time and worst-case conditions. 9 testing data are given by professor, and the length of them are 10 times power of 2s, from 1 to 9.

Lastly, the observation of them will be focusing on the time complexity of the results from implementation. Moreover, I'll compare 3 algorithms with themselves and make some rankings. Finally, I'll check the experimented results with my analyzation.

Analysis:

1. Linear Search:

a. Abstract:

Linear search is a rather direct approach of searching. It goes through all the items in the array until it finds the requested item, or else it returns "not found".

b. Algorithm:

```
1. // Find the location of word in array list.
2. // Input: word, array list, int n
3. // Output: int i such that list[i] = word.
4. Algorithm search(word, list, n)
5. {
6.     for i := 1 to n do {
7.         if(list[i] = word) return i;
8.     }
9.     return -1;
10. }
```

c. Proof of correctness:

We can find that there's a truth: at the start of the i -th iteration, there is no requested item, v , in the array list $A[0, \dots, i - 1]$. The search index will be n .

During the i -th iteration, there are 2 conditions: $A[i] == v$ or $A[i] != v$. If $A[i] == v$, the requested item's position will be found, which is i . If $A[i] != v$, let $j = i + 1$, then by the aforementioned truth, during the j -th iteration, where $j = i + 1$, there is no requested item, v , in the list $A[1, \dots, j - 1]$, or in $A[1, \dots, i]$. Therefore, the truth applies to all iterations from 1 to n .

The loop terminates in 2 ways. It either ends when v is found, or the iteration will go through n iterations same as what is mentioned in the last paragraph, and finds that in the $n + 1$ -th iteration, $A[1, \dots, n]$ does not contain v , which indicates "not found". Hence proved.

d. Time complexity:

	s/e	freq	total
1. Algorithm search(word, list, n)	0	0	0
2. {	0	0	0
3. for i := 1 to n do {	n+1	1	n+1
4. if(list[i] = word) return i;	1	n	n
5. }	0	0	0
6. return -1;	1	1	1
7. }	0	0	0
			2n+2

Worst-case:

When the specific item is at the end of the array, the steps would be $2n+2$, as the table shows, and the time complexity would be **$O(n)$** .

Best-case:

When the specific item is at the front of the array, the steps would be 1, and the time complexity would be **$O(1)$** .

Average-case:

When the specific item randomly exists in the array, the steps would be $2c+2$, where c is an integer between 1 to n . I choose $n/2$ in this case, since it's the average. Therefore, the steps would be $n+2$, and the time complexity would be **$O(n)$** .

2. Bidirectional Search:

a. Abstract:

Bidirectional Search is the transformation of Linear Search. It only runs

$n/2$ iterations, however in each iteration, it searches 2 items gradually from both directions.

b. Algorithm:

```

1. // Bidirectional search to find the location of word in array list.
2. // Input: word, array list, int n
3. // Output: int i such that list[i] = word.
4. Algorithm search(word, list, n)
5. {
6.     for i := 1 to n/2 do {
7.         if(list[i] = word) return i;
8.         if(list[n - i - 1] = word) return n - i - 1;
9.     }
10.    return -1;
11. }

```

c. Proof of correctness:

We can find that there's a truth: at the start of the i -th iteration, there is no requested item, v , in the array lists $A[1, \dots, i - 1]$ and $A[n - i, \dots, n]$. The search index will be n , and the iteration number will be $n/2$.

During the i -th iteration, there are 2 conditions: $\{A[i] == v \text{ or } A[n - i + 1] == v\}$, and $\{A[i] != v \text{ or } A[n - i + 1] != v\}$. If it fits the first condition, the requested item's position will be found, which is either i or $n - i + 1$. If it fits the second condition, then by the aforementioned truth, during the j -th iteration, where $j = i + 1$, there is no requested item, v , in the list $A[1, \dots, j - 1]$, or in $A[1, \dots, i]$. Therefore, the truth applies to all iterations from 1 to n .

The loop terminates in 2 ways. It either ends when v is found, or the iteration will go through $n/2$ iterations same as what is mentioned in the last paragraph, and finds that in the $n/2 + 1$ -th iteration, $A[1, \dots, n]$ does not contain v , which indicates "not found". Hence proved.

d. Time complexity:

	s/e	freq	total
1. Algorithm search(word, list, n)	0	0	0
2. {	0	0	0
3. for i := 1 to n/2 do {	$n/2 + 1$	1	$n/2 + 1$
4. if(list[i] = word) return i;	1	$n/2$	$n/2$
5. if(list[n - i - 1] = word) return n - i - 1;	1	$n/2$	$n/2$
6. }	0	0	0

7. return -1;	1	1	1
8. }	0	0	0
			3/2n+2

Worst-case:

When the specific item is at the middle of the array, the steps would be $3/2n+2$ steps, as the table shows, and the time complexity would be **$O(n)$** .

Best-case:

When the specific item is at the front of the array, the steps would be 1 step, and the time complexity would be **$O(1)$** .

Average-case:

When the specific item randomly exists in the array, the steps would be $3/2c+2$, where c is an integer between 1 to n . I choose $n/2$ in this case, since it's the average. Therefore, the steps would be $3/4n+2$, and the time complexity would be **$O(n)$** .

3. Random-directional Search

a. Abstract:

It is basically Linear search, but Random-directional Search changes it's search direction based on a random number between 0 and 1.

b. Algorithm:

```

1. // Random-direction search to find the location of word in array list.
2. // Input: word, array list, int n
3. // Output: int i such that list[i] = word.
4. Algorithm search(word, list, n)
5. {
6.     choose j randomly from the set {0, 1};
7.     if (j=1) then
8.         for i := 1 to n do {
9.             if(list[i] = word) return i;
10.        }
11.    else
12.        for i := n to 1 step -1 do {
13.            if(list[i] = word) return i;
14.        }
15.    return -1;
16. }
```

c. Proof of correctness:

The proof of Random-directional Search is basically the same with Linear Search. Change the direction and apply it again.

d. Time complexity:

	s/e	freq	total
1. Algorithm search(word, list, n)	0	0	0
2. {	0	0	0
3. choose j randomly from the set {0, 1};	1	1	1
4. if (j==1) then	1	1	1
5. for i := 1 to n do {	n+1	1	n+1
6. if(list[i] = word) return i;	1	n	n
7. }	0	n	0
8. else	0	1	0
9. for i := n to 1 step -1 do {	n+1	1	n+1
10. if(list[i] = word) return i;	1	n	n
11. }	0	n	0
12. return -1;	1	1	1
13. }	0	0	0
			2n+3

Worst-case:

When the specific item is at the end of the array when $j == 1$, or the specific item is at the front of the array when $j == 0$, the steps would be $2n+3$, as the table shows, and the time complexity would be **$O(n)$** .

Best-case:

When the specific item is at the front or the end of the array, accordingly, the steps would be 1, and the time complexity would be **$O(1)$** .

Average-case:

When the specific item randomly exists in the array, the steps would be $2c+3$, where c is an integer between 1 to n . I choose $n/2$ in this case, since it's the average. Therefore, the steps would be $n+3$, and the time complexity would be **$O(n)$** .

4. Comparison:

	Linear		Bidirectional		Random-directional	
Best	1	$O(1)$	1	$O(1)$	1	$O(1)$
Worst	$2n+2$	$O(n)$	$3/2n+2$	$O(n)$	$2n+3$	$O(n)$
Average	$n+2$	$O(n)$	$3/4n+2$	$O(n)$	$n+3$	$O(n)$

Average Speed (fast>slow): BDSearch > LinearSearch > RDSearch.

Worst-case Speed (fast>slow): BDSearch > LinearSearch ≈ RDSearch.

Implementation:

1. Average-case scenario:

Since the most average case of testing the algorithm is randomly pick an item and search, I pick an item in given data randomly, implement on the algorithm, and repeat the procedure 500 times. The mean of the recorded CPU time between those steps are the average CPU runtime of the algorithm.

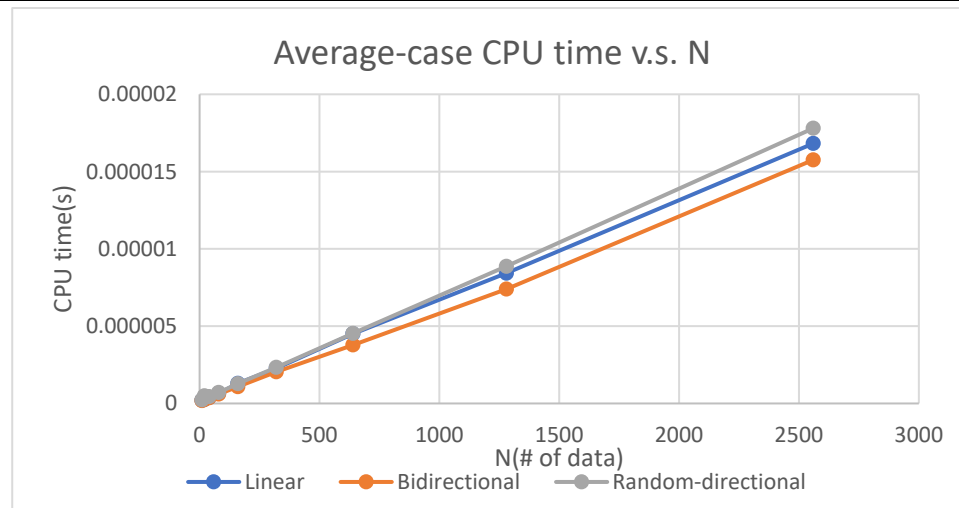
Workflow:

Should search for all items and get the average CPU time!

```
1. t = GetTime(); // initialize time counter
2. for i := 0 to 500 do {
3.     random_num = RandomNumber(N); // generate a random number
4.     Search(random_num); // search item accordingly
5. }
6. t = (GetTime() - t) / 500; // calculate CPU time / iteration
```

Results:

N(num)	Linear(s)	Bidirectional(s)	Random-directional(s)
10	2.00272E-07	1.83582E-07	2.34127E-07
20	2.57969E-07	2.52247E-07	4.95911E-07
40	3.84331E-07	3.63827E-07	4.42028E-07
80	6.46114E-07	6.0606E-07	7.10011E-07
160	1.30606E-06	1.08576E-06	1.28174E-06
320	2.27022E-06	2.04182E-06	2.33221E-06
640	4.50611E-06	3.78609E-06	4.53568E-06
1280	8.42619E-06	7.39813E-06	8.87442E-06
2560	1.682E-05	1.57599E-05	1.78042E-05



2. Worst-case scenario:

For Linear Search, the worst case will happen when the item for searching is at the end of the array. Therefore I choose to **search data[N - 1]** as the worst-case condition.

For Bidirectional Search, the worst case will happen when the item for searching is at the middle of the array. Therefore, I choose to **search data[N/2]** as the worst-case condition.

For Bidirectional Search, the worst case will happen when the item for searching is at the front or end of the array, depending on the random number algorithm generates. But, there's no way for me to set the item for searching according to the random number generated by algorithm without altering the algorithm itself. Therefore, I choose to **search data[N/2]**, as the worst-case condition.

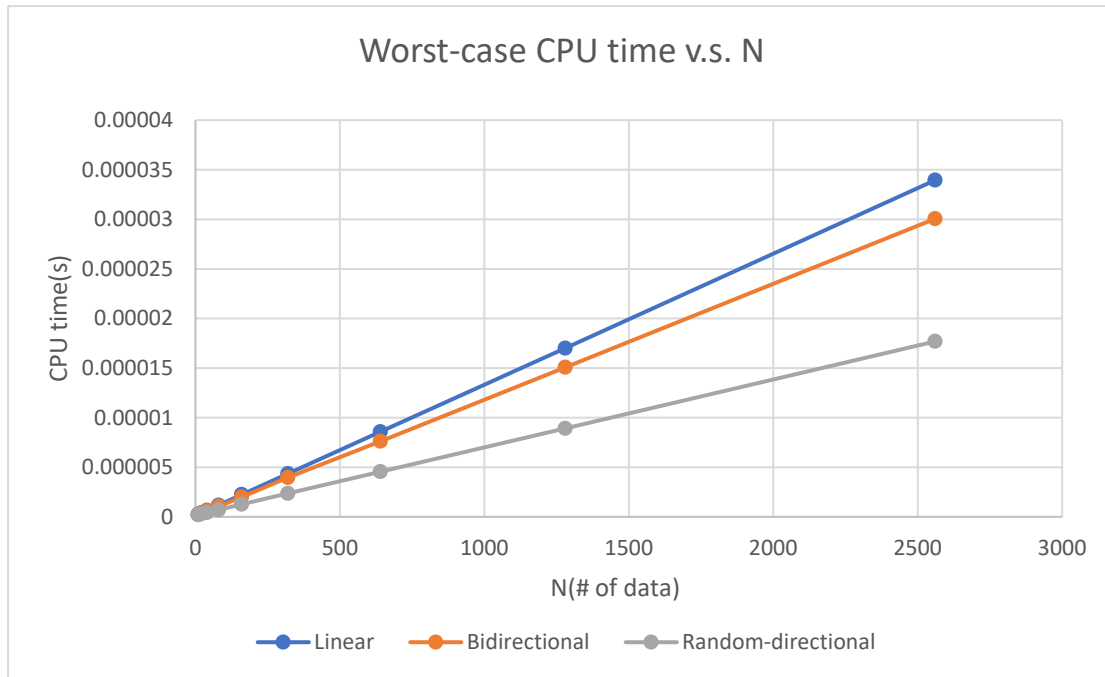
Since the worst-case scenario must compare with average case, I generate a random number in this workflow, too. This way, they are compared with each other in the same condition.

Workflow:

```
1. t = GetTime(); Should be 5000.           // initialize time counter
2. for i := 0 to 500 do {
3.     random_num = RandomNumber(N);           // generate a random number
4.     Search(n);      // search for item (list[N])
5. }
6. t = (GetTime() - t) / R;                    // calculate CPU time / iteration
```

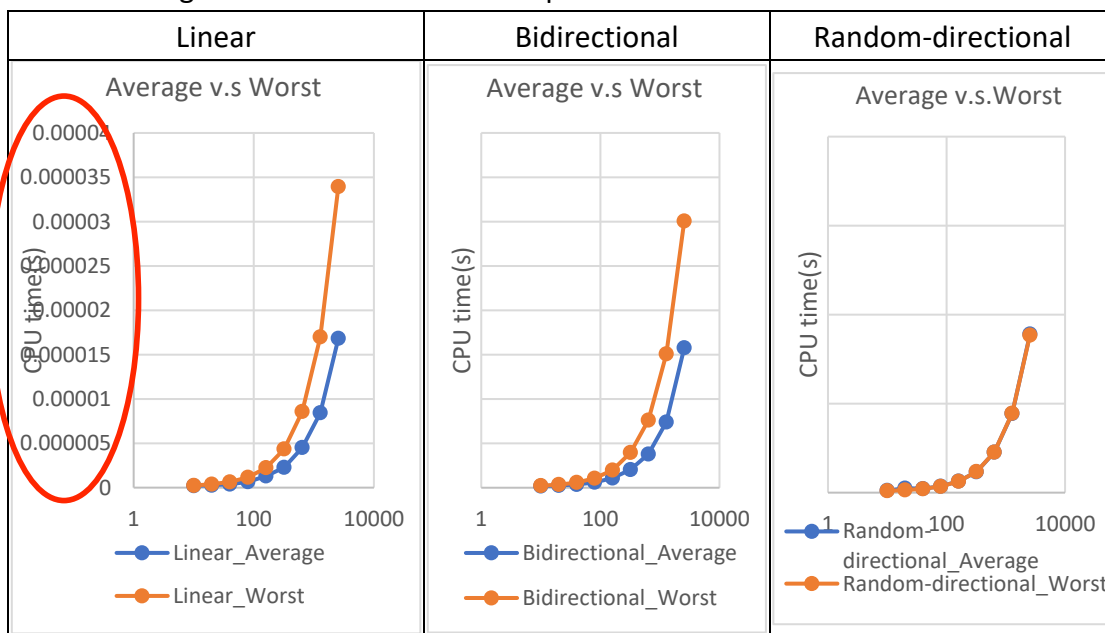
Results:

N(num)	Linear(s)	Bidirectional(s)	Random-directional(s)
10	2.39849E-07	2.26021E-07	1.99795E-07
20	3.85761E-07	3.48091E-07	2.89917E-07
40	6.38008E-07	5.8794E-07	4.26292E-07
80	1.16205E-06	1.0643E-06	6.97613E-07
160	2.2378E-06	1.98603E-06	1.266E-06
320	4.34637E-06	3.96013E-06	2.36225E-06
640	8.57401E-06	7.61604E-06	4.54807E-06
1280	1.69978E-05	1.50738E-05	8.9159E-06
2560	3.3946E-05	3.00579E-05	1.76859E-05



Observation:

1. Average-case v.s. Worst-case comparisons:



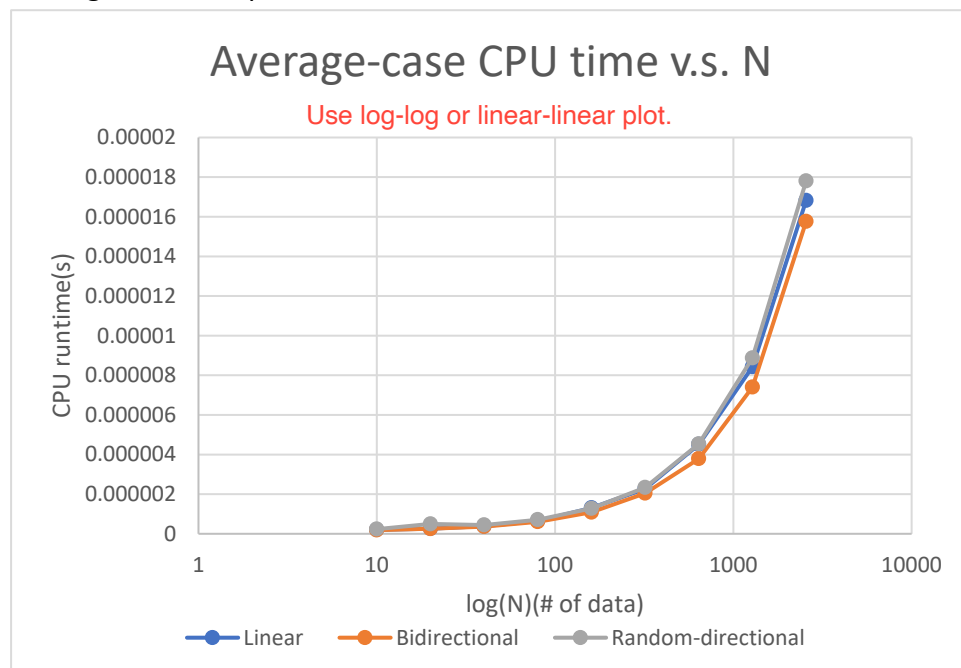
p.s. The x-axis is $\log(\# \text{ of data})$, y-axis is CPU runtime. The scale of three charts are the same, y: from 0 to 0.00004, x: 1 to 10000.

For Linear Search and Bidirectional Search, **the Worst-case scenario has a longer CPU runtime**. This indicates that the experiments between average case and worst case is valid. However, the result of Random-directional Search did not seem valid since the worst-case scenario should spend longer time than average-case. The reason of this is because the worst-case of this is to pick the target in the middle, and this seem like an average case, too.

Therefore, there's no big changes between average and worst case of RDSearch.

I did one more experience, I gave RDSearch list[0] or list[n-1] to search, depending on the direction it generated. This way, I can **simulate the true worst-case**, which would happen if it's an unlucky day. The spent time of this **worst-case scenario is much longer than the original result**.

2. Average-case comparisons:



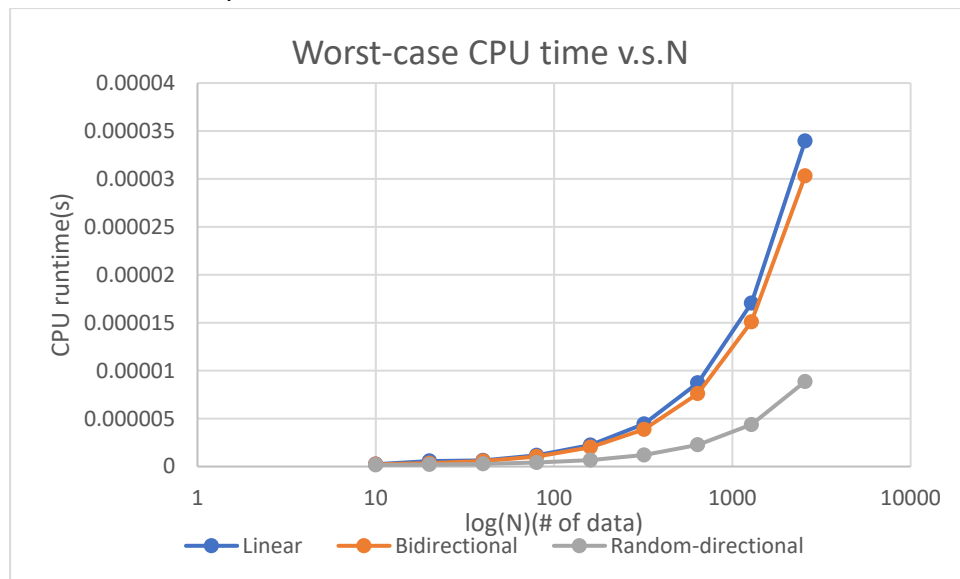
According to the graph in **Implementation**, all 3 algorithms' time complexities are clearly $O(n)$, since they are linear.

Furthermore, according to this graph, we can conclude the speeds of the 3 algorithms, is **BDSearch > Linear Search > RDSearch (> means faster than)**. This result exactly meets my prediction. However, based on the results and the analysis, there's only a slightly difference between the speeds of the three algorithms.

The reason why Bidirectional search is the fastest, I think, is because: **having half of the iterations but contains double amount of statements (BDSearch) could run faster than full iterations with single statement (Linear Search)**. Bidirectional search runs through only $n/2$ iterations while Linear search runs though n iterations.

The reason why Random-directional search is the slowest, I think, is because it contains a random number generator and some if/else in it. Therefore, it is the slowest in the average case.

3. Worse-case comparisons:



According to the graph in **Implementation**, the **worst-case scenario fits our calculation of time complexity, which are all $O(n)$** .

The **worst case of Random-directional Search is much faster**. This is because we cannot make sure that which direction it will go, so we can only assign a target that is in the middle of the array for the algorithm to find. It is no different than the average-case.

I've done another worst-case scenario for Random-directional search, which is choosing the worst case depending on the random-generated direction. The result fits the observation in Average-case part and my calculation.

Conclusion:

1. Time complexities of the 3 algorithms:

	Linear	Bidirectional	Random-directional
Best	$O(1)$	$O(1)$	$O(1)$
Worst	$O(n)$	$O(n)$	$O(n)$
Average	$O(n)$	$O(n)$	$O(n)$

And they are verified by implementation on gcc on EE Workstation.

2. Actual runtime comparison (Average):

BDSearch > Linear Search > RDSearch (> means runs faster)

3. Actual runtime comparison (Worst):

RDSearch > BDSearch > Linear Search (> means runs faster)

4. Half iteration but contains double statement (BDSearch) could run faster than whole iteration with single statement (Linear Search).

5. The worst-case of Random-directional Search is not accurate.

hw02.c

```
1 // EE3980 HW02 Random Data Searches
2 // 106061146, Jhao-Ting, Chen
3 // 2020/03/18
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8 #include <sys/time.h>
9
10 int N; // input size
11 char **data; // input data
12 char **A; // array to be sorted
13 int R = 500; // number of repetitions
14
15 void readInput(void); // read all inputs
16 double GetTime(void); // get local time in seconds
17 int Search(char *word, char **list, int n); // Linear Search
18 int BDSearch(char *word, char **list, int n); // Bidirection Search
19 int RDSearch(char *word, char **list, int n); // Random-direction Search
20
21 int main(void)
22 {
23     int i, result = 0, random_num; // loop index
24     double t; // for CPU time tracking
25
26     readInput(); // read input data
27     printf("n: %d\n", N); // print N
28
29     t = GetTime(); // initialize time counter
30     for (i = 0; i < R; i++) {
31         random_num = (rand() % N); // generate a random number
32         result = Search(data[random_num], data, N); // search item accordingly
33     }
34     t = (GetTime() - t) / R; // calculate CPU time / iteration
35     printf("Linear search average CPU time: %e\n", t); // print result
36     printf("Linear search average CPU time: %e\n", t); // print result
37
38     t = GetTime(); // initialize time counter
39     for (i = 0; i < R; i++) {
40         random_num = (rand() % N); // generate a random number
41         result = BDSearch(data[random_num], data, N); // search item
42     }
43     t = (GetTime() - t) / R; // calculate CPU time / iteration
44     printf("Bidirection search average CPU time: %e\n", t); // print result
45     printf("Bidirection search average CPU time: %e\n", t); // print result
46
47     t = GetTime(); // initialize time counter
48     for (i = 0; i < R; i++) {
```

```

47     random_num = (rand() % N);           // generate a random number
48     result = RDSearch(data[random_num], data, N); // search item
49 }
50 t = (GetTime() - t) / R;                 // calculate CPU time / iteration
51 printf("Random-direction search average CPU time: %e\n", t); // result
52 printf("Random-direction search average CPU time: %e\n", t); // result
53
52 t = GetTime();                          // initialize time counter
53 for (i = 0; i < R; i++) {
54     random_num = (rand() % N);           // generate a random number
55     result = Search(data[N-1], data, N); // search for the middle one
56     result = Search(data[N - 1], data, N); // search for the middle one
57 }
58 t = (GetTime() - t) / R;                 // calculate CPU time / iteration
59 printf("Linear search worse-case CPU time: %e\n", t); // result
60 printf("Linear search worse-case CPU time: %e\n", t); // result
61
60 t = GetTime();                          // initialize time counter
61 for (i = 0; i < R; i++) {
62     random_num = (rand() % N);           // generate a random number
63     result = BDSearch(data[N/2], data, N); // search for the middle one
64     result = BDSearch(data[N / 2], data, N); // search for the middle one
65 }
66 t = (GetTime() - t) / R;                 // calculate CPU time / iteration
67 printf("Bidirection search worse-case CPU time: %e\n", t); // result
68 printf("Bidirection search worse-case CPU time: %e\n", t); // result
69
68 t = GetTime();                          // initialize time counter
69 for (i = 0; i < R; i++) {
70     random_num = (rand() % N);           // generate a random number
71     result = RDSearch(data[N/2], data, N); // search for the middle one
72     result = RDSearch(data[N / 2], data, N); // search for the middle one
73 }
74 t = (GetTime() - t) / R;                 // calculate CPU time / iteration
75 printf("Random-direction search worse-case CPU time: %e\n", t); // result
76 printf("Random-direction search worse-case CPU time: %e\n", t); // result
77
76 return 0;
77 }
78
79 void readInput(void)                    // read all inputs
80 {
81     int i, j, len;                      // for looping and dynamic store
82     char str[50];                       // for dynamic store
83
84     scanf("%d", &N);                    // read N for # of words
85
86     data = (char **)calloc(N, sizeof(char *)); // initialize data size (N)
87     A = (char **)calloc(N, sizeof(char *));   // initialize A size (N)
88
89

```

```

90     for (i = 0; i < N; i++) {
91         scanf("%s", str);                // store each word in data
92         len = strlen(str);
93         data[i] = (char *)calloc(len, sizeof(char));
94         A[i] = (char *)calloc(len, sizeof(char));
95         for (j = 0; j < len; j++) {
96             data[i][j] = str[j];
97             A[i][j] = str[j];
98         }
99     }
100 }
101
102 double GetTime(void)                    // demonstration code from 1.1.3
103 {
104     struct timeval tv;
105
106     gettimeofday(&tv, NULL);
107     return tv.tv_sec + 1e-6 * tv.tv_usec;
108 }
109
110 int Search(char *word, char **list, int n)    // Linear Search
111 {
112     int i;                                // for looping
113     for (i = 0; i < n; i++) {              // compare all possible entries
114         if (list[i] == word) {
115             Need to use strcmp for string comparison!
116             return i;
117         }
118     }
119     return -1;                            // unsuccessful search
120 }
121
122 int BDSearch(char *word, char **list, int n)    // Bidirection Search
123 {
124     int i;                                // for looping
125     for (i = 0; i <= n/2; i++) {            // compare all entries from both ends
126         for (i = 0; i <= n / 2; i++) {      // compare all entries from both ends
127             if (list[i] == word) {
128                 return i;
129             }
130             if (list[n - i - 1] == word) {
131                 return n - i - 1;
132             }
133         }
134     }
135     return -1;                            // unsuccessful search
136 }
137
138 int RDSearch(char *word, char **list, int n)    // Random-direction Search
139 {
140     int i, j;

```

```

138     j = rand() % 2;
139     if (j == 1) {
140         for (i = 1; i < n; i++) {           // forward search
141             if (list[i] == word) {
142                 return i;
143             }
144         }
145     }else{
146         } else {
147             for (i = n - 1; i >= 0; i--) {   // backward search
148                 if (list[i] == word) {
149                     return i;
150                 }
151             }
152         return -1;                          // unsuccessful search
153     }

```

[Program Format] can be improved.

[Writing] hw02a.pdf spelling errors: analyzation(1)

[Worst-case] measurement should repeat 5000 times.

[Average] CPU time should search for all items and then take the average.

[strcmp] should be used to compare the word with the array elements.

[Average and worst-case] time complexities can be more clearly described.

[Space] complexity should be analyzed and reported in big-O notation.

[Conclusions] should be re-drawn after correcting the program.

[Report] needs to have double line space.

[Good] effort in writing the report. Please continue to improve the report writing.

Score: 70