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:

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

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, …, 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, …, j - 1], or in A[1, …, 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, …, n] does not contain v, which indicates "not found". Hence proved.

d. Time complexity:

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

<u>Worst-case:</u>

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)**.

<u>Best-case:</u>

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

<u>Average-case:</u>

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:

```
// Bidirectional search to find the location of word in array list.
// Input: word, array list, int n
// Output: int i such that list[i] = word.
Algorithm search(word, list, n)
{
    for i := 1 to n/2 do {
        if(list[i] = word) return i;
        if(list[n – i - 1] = word) return n – i - 1;
    }
    return -1;
}
```

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, …, i - 1] and A[n - i, …, 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 or A[n – i + 1] == v}, and {A[i] != v 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, …, j - 1], or in A[1, …, 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, …, n] does not contain v, which indicates "not found". Hence proved.

d. Time complexity:

	s/e	freq	total
`Algorithm search(word, list, n)`	0	0	0
`{`	0	0	0
`for i := 1 to n/2 do {`	n/2+1	1	n/2+1
`if(list[i] = word) return i;`	1	n/2	n/2
`if(list[n – i - 1] = word) return n – i - 1;`	1	n/2	n/2
`}`	0	0	0

`return -1;` `}`	1 0	1 0	1 0
			3/2n+2

<u>Worst-case:</u>

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)**.

<u>Best-case:</u>

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)**.

<u>Average-case:</u>

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:

```
// Random-direction search to find the location of word in array list.
// Input: word, array list, int n
// Output: int i such that list[i] = word.
Algorithm search(word, list, n)
{
    choose j randomly from the set {0, 1};
    if (j=1) then
        for i := 1 to n do {
            if(list[i] = word) return i;
        }
    else
        for i := n to 1 step -1 do {
            if(list[i] = word) return i;
        }
    return -1;
}
```

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
`Algorithm search(word, list, n)`	0	0	0
`{`	0	0	0
`choose j randomly from the set {0, 1};`	1	1	1
`if (j=1) then`	1	1	1
`for i := 1 to n do {`	n+1	1	n+1
`if(list[i] = word) return i;`	1	n	n
`}`	0	n	0
`else`	0	1	0
`for i := n to 1 step -1 do {`	n+1	1	n+1
`if(list[i] = word) return i;`	1	n	n
`}`	0	n	0
`return -1;`	1	1	1
`}`	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:**

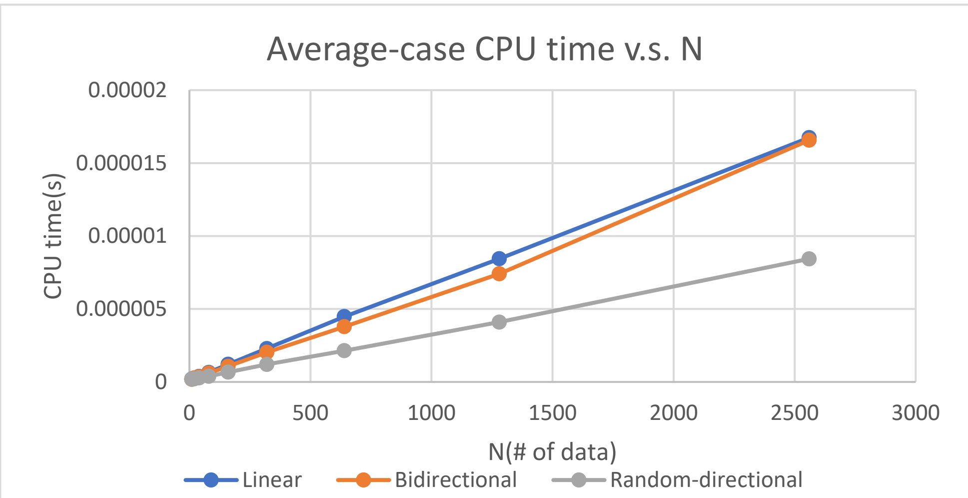
Since the most average case of testing the algorithm is randomly pick an item and search, I pick an item in given data randomly, implement on the algorithm, and repeat the procedure 500 times. The mean of the recorded CPU time between those steps are the average CPU runtime of the algorithm.

Workflow:

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

Results:

N(num)	Linear(s)	Bidirectional(s)	Random-directional(s)
10	2.00272E-07	1.84059E-07	1.94073E-07
20	2.63691E-07	2.61784E-07	2.16007E-07
40	3.80039E-07	3.56197E-07	2.76089E-07
80	6.51836E-07	6.0606E-07	3.83854E-07
160	1.20592E-06	1.07002E-06	6.68049E-07
320	2.28834E-06	2.03419E-06	1.2002E-06
640	4.47989E-06	3.78799E-06	2.14386E-06
1280	8.44193E-06	7.41005E-06	4.10032E-06
2560	1.67422E-05	0.000016572	8.43811E-06



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.

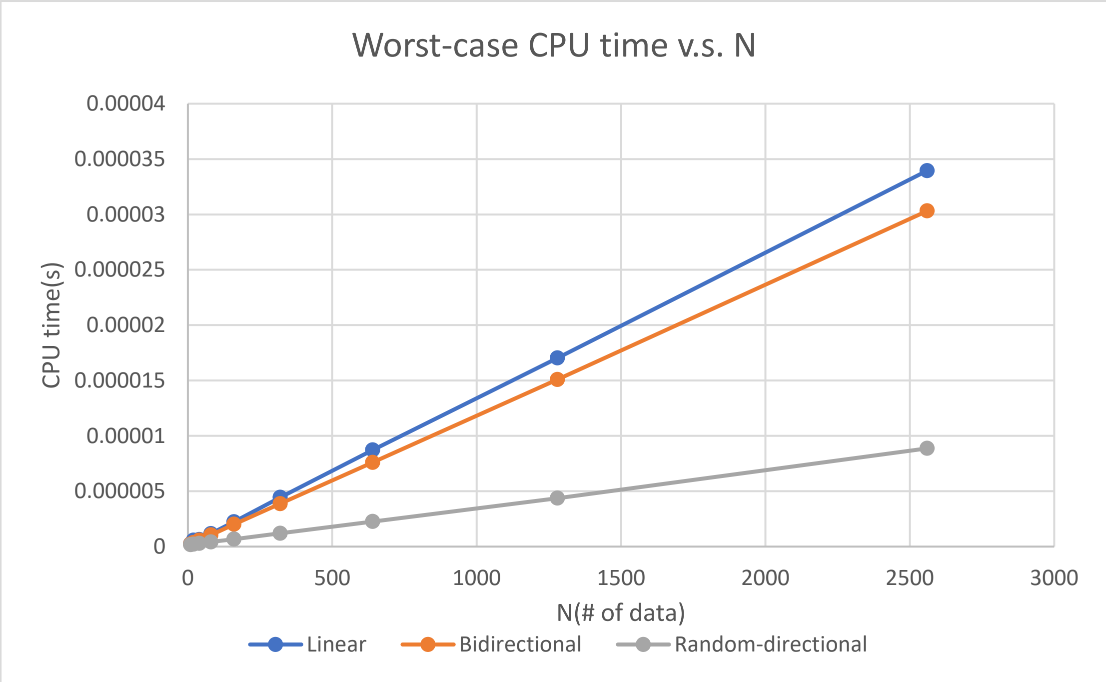
Since the worst-case scenario must compare with average case, I generate a random number in this workflow, too. This way, they are compared with each other in the same condition.

Workflow:

```
t = GetTime();                                  // initialize time counter
for i := 0 to 500 do {
      random_num = RandomNumber(N);          // generate a random number
      Search(n);     // search for item (list[N])
}
t = (GetTime() - t) / R;                        // calculate CPU time / iteration
```

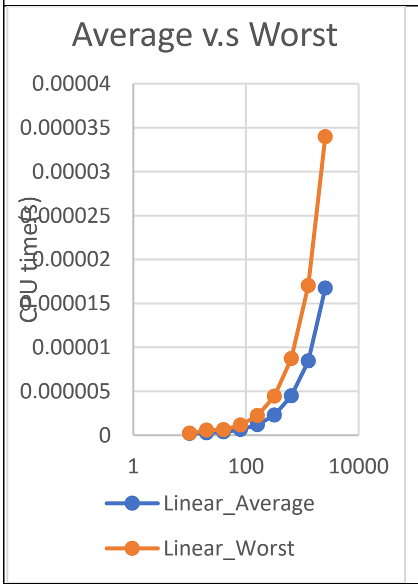
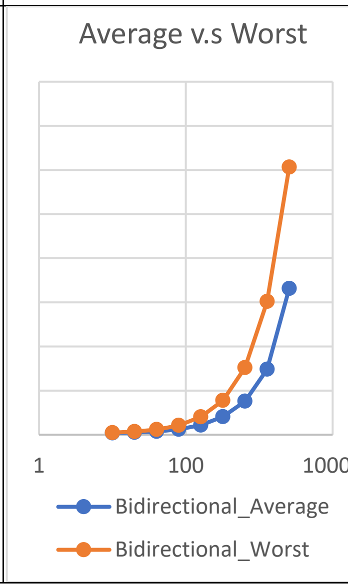
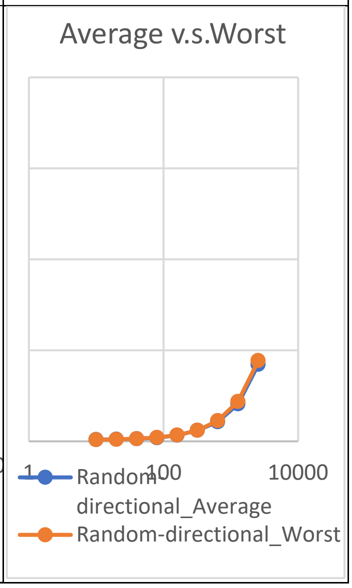
Results:

N(num)	Linear(s)	Bidirectional(s)	Random-directional(s)
10	2.41756E-07	2.26021E-07	1.68324E-07
20	5.77927E-07	3.51906E-07	2.11716E-07
40	6.36101E-07	5.79834E-07	2.77996E-07
80	1.17159E-06	1.04618E-06	4.18186E-07
160	2.23637E-06	2.01416E-06	6.69956E-07
320	4.44365E-06	3.8743E-06	1.2002E-06
640	8.72183E-06	7.60222E-06	2.26402E-06
1280	1.70341E-05	1.50881E-05	4.37212E-06
2560	3.39518E-05	3.03121E-05	8.86822E-06



Observation:

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

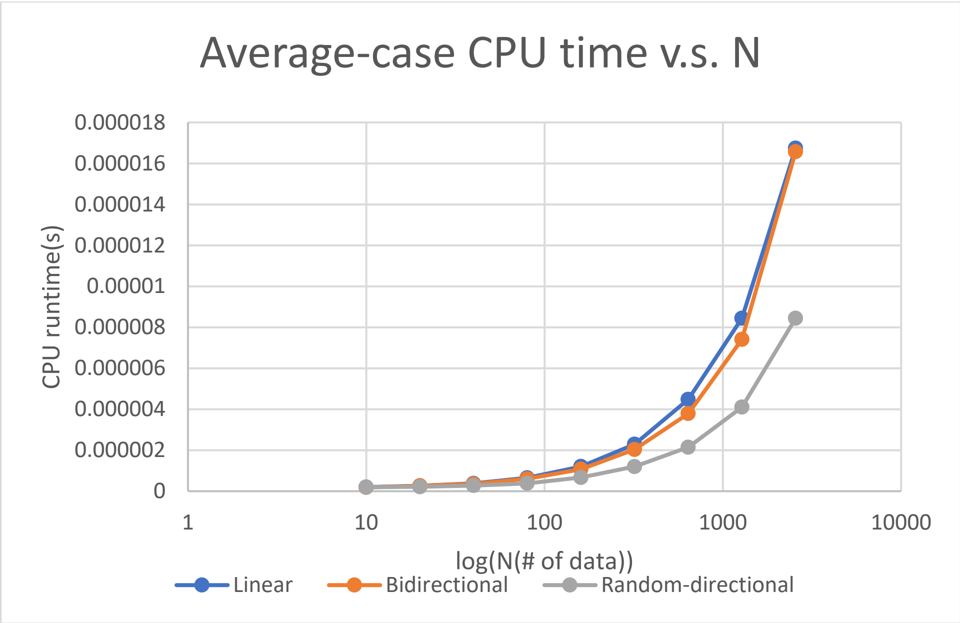
<table>
<tr><th>Linear</th><th>Bidirectional</th><th>Random-directional</th></tr>
<tr>
<td>
</td>
<td>
</td>
<td>
</td>
</tr>
</table>

p.s. The x-axis is log(# 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.

In each case, **the Worst-case scenario has a longer CPU runtime.** This indicates that the experiments between average case and worst case is valid.

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 more longer than the original result.

2. Average-case comparisons:

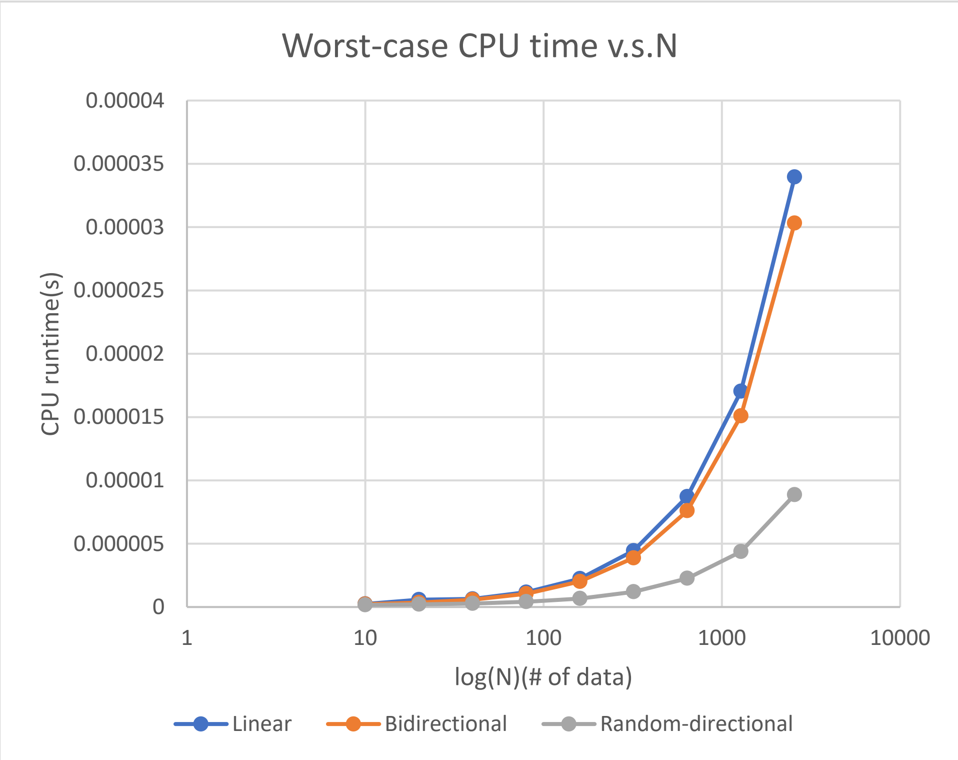


According to the graph in **Implementation**, all 3 algorithms' time complexities are clearly O(n), since they are linear.

Furthermore, according to this graph, we can conclude the speeds of the 3 algorithms, is **RDSearch > BDSearch > Linear Search (> means faster than)**. This did not meet my prediction, a possible reason is that, if they are given the same targets to find, **RDSearch has 50% chances of starting from direction that is closer to the targets**, while BDSearch and Linear Search's directions are fixed. Therefore, RDSearch will be faster than the other two.

As for BDSearch and Linear Search, Bidirectional Search has half lesser iteration but double statements in an iteration, comparing to Linear Search, it runs slightly faster than Linear Search. So, the result indicates that **half iteration but contains double statement could run faster than whole iteration with single statement.**

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).

Since the result fits the Average-case comparison result, all the observations are similar to those in Average-case comparisons.

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 & Worst-case):
RDSearch > BDSearch > Linear Search (> means runs faster)
3. Random-direction Search has 50% chances of starting from direction that is closer to the target, therefore it is the fastest. (RDSearch)
4. Half iteration but contains double statement (BDSearch) could run faster than whole iteration with single statement (Linear Search).