Random Data Searches

110191019 YU-TING CHUNG

Introduction

This homework is about implementing three basic search methods: Linear Search, Bidirectional Search, and Random-Direction Search. We need to write these search functions in C and test their performance. By using the word lists data from Homework 1, we will measure how long each algorithm takes by running multiple tests (at least 500 times for average cases and 5000 times for worst cases). Finally, we will analyze their time and space complexities and compare the theoretical results with actual test results to see which method is more efficient.

Approach

- Linear search

1. Algorithm:

```
1   Algorithm search(word,list,n)
2   {
3       for i := 1 to n do {            // compare all possible entries
4           if list[i] = word return i; // successful
5       }
6       return -1;                      // unsuccessful
7
8   }
```

When the algorithm stops, there are two possible outcomes:

(1) If i < n, then A[i] =x, which means the algorithm found x in the array

and correctly returns i as its position.

(2) If i = n, all elements have been checked, and none match x. In this case, the algorithm correctly returns −1 to indicate that x is not in the array.

2. Proof of correctness:

At the start of the algorithm, the index variable i is initialized to 0. In each step of the while loop, i increases by 1. The loop runs as long as i < n, and since n is a fixed number, i cannot be bigger than n. This ensures that the loop will stop after at most n steps, and the algorithm finishes in a finite amount of time. Since the two cases above cover all possibilities, the linear search algorithm always produces the correct result. Hence, proved.

3. Time and space complexities:

		s/e	freq	total
1	Algorithm search(word,list,n)	0	0	0
2	{	0	0	0
3	for i := 1 to n do {	1	n+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
8	}	0	0	0
				2n+2

Best case:

When the word that we want to search is at the front of the array, the

time complexity is $\Theta(1)$.

Average case:

When the word that we want to search is at n/2 , which is in the middle,

the time complexity is $\Theta(n)$.

Worst case:

When the word that we want to search is at the end of the array, the

time complexity is $\Theta(n)$.

Space complexity: 4 (word, list, i, n)

- Bidirection search

1. Algorithm:

```
Algorithm BDsearch(word,list,n)
{
    for i := 1 to n/2 do {                    // compare all entries from both directions
        if list[i] = word return i;           // successful
        if list[n - i +1] = word return n-i+1;  // successful
    }
    return -1;                                // unsuccessful
}
```

The Bidirectional Search algorithm operates by simultaneously

searching from both directions until the two meets.

When the algorithm stops, there are two possible outcomes:

(1) If the forward and backward searches meet at a common node, the

algorithm successfully finds the shortest path.

(2) If all possible nodes have been explored without the two searches

meeting, the algorithm concludes that the item is not found.

2. Proof of correctness:

The algorithm initializes a search frontier from both the start and the goal and expands step by step while avoiding revisiting explored nodes. If the item exists, the two frontiers will meet and find the path; otherwise, it confirms not found. Since bidirectional search meets in the middle, and the two cases above cover all possibilities, the bidirection search algorithm always produces the correct result.

Hence, proved.

3. Time and space complexities:

```
1  Algorithm BDsearch(word,list,n)
2  {
3      for i := 1 to n/2 do {
4          if list[i] = word return i;
5          if list[n - i +1] = word return n-i+1;
6      }
7      return -1;
8  }
```

s/e	freq	total
0	0	0
0	0	0
1	n/2+1	n/2+1
1	n/2	n/2
1	n/2	n/2
0	0	0
1	1	1
0	0	0
		3n/2+2

Best case:

When the word that we want to search is at the front of the array, the time complexity is $\Theta(1)$.

Average case:

When the word that we want to search is at n/2 , which is in the middle,

the time complexity is $\Theta(n)$.

Worst case:

When the word that we want to search is in the middle of the array, the time complexity is $\Theta(n)$.

Space complexity: 4 (word, list, i, n)

- Random-direction search

1. Algorithm:

```
1   Algorithm RDsearch(word,list,n)
2   {
3       choose j randomly from the set {0, 1} ;
4       if j =1 then
5           for i := 1 to n do {                // forward search
6               if list[i] = word return i;     // successful
7       }
8       else
9           for i := n to 1 step -1 do {        // backward search
10              if list[i] = word return i;     // successful
11          }
12      return -1;                              // unsuccessful
13  }
```

The Random-Direction Search starts searching for the item by randomly selecting direction (0 or 1) at each step.

When the algorithm stops, there are two possible outcomes:

(1) If the randomly chosen sequence of steps reaches the goal, the algorithm successfully returns to the path.

(2) If a stopping criterion is met (e.g., exceeding a maximum number of

steps or visiting too many states without progress), the algorithm concludes that the goal is not reachable via the random path.

2. Proof of correctness:

Since the search direction is random, the same input may produce different results. In an ideal case, if the number of steps is unlimited and the search space is connected, the goal will eventually be found. However, in practice, the success rate depends on the structure of the search space and the impact of randomness. Therefore, while the algorithm is correct, it may not efficiently find a solution within a limited time.

3. Time and space complexities:

		s/e	freq	total
1	Algorithm RDsearch(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 {	1	n+1	n+1
6	if list[i] = word return i;	1	n	n
7	}	0	0	0
8	else	0	1	0
9	for i := n to 1 step -1 do {	1	n+1	n+1
10	if list[i] = word return i;	1	n	n
11	}	0	0	0
12	return -1;	1	1	1
13	}	0	0	0
				2n+4

Best case:

When the word that we want to search is at the front or at the end of the array, the time complexity is $\Theta(1)$.

Average case:

When the word that we want to search is at n/2 , which is in the middle, the time complexity is $\Theta(n)$.

Worst case:

If the word that we want to search is at the front of the array when j=0, or when it is at the end of the array when j=1, the time complexity is $\Theta(n)$.

Space complexity: 5 (word, list, i, j, n)

4. Comparison:

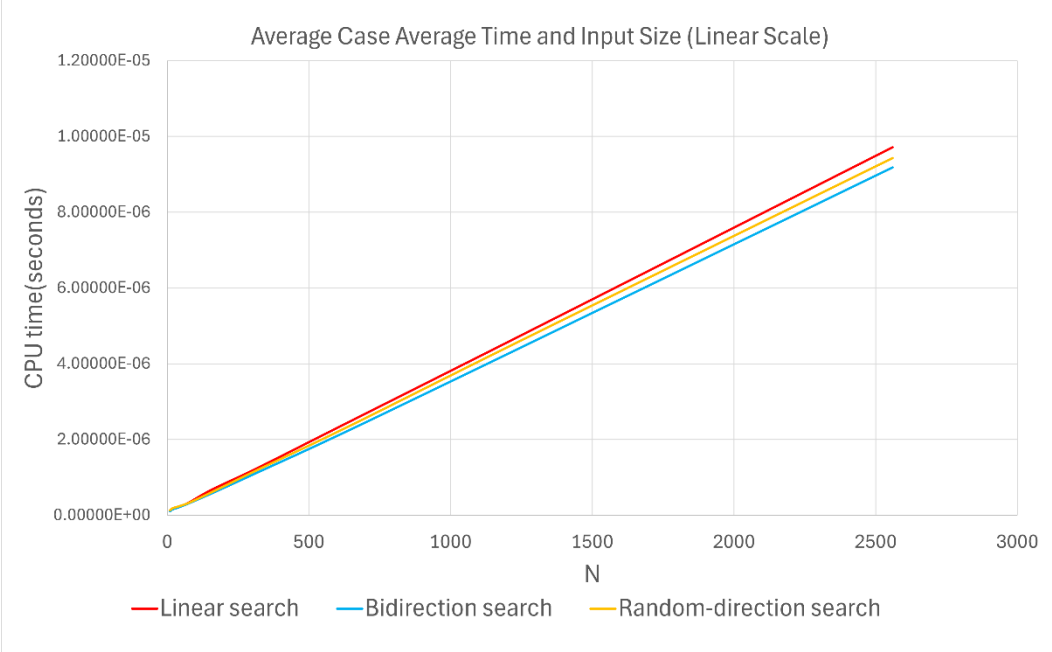
	Linear search	Bidirection search	Random-direction search
	2n+2	3n/2+2	2n+4
Best	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Average	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Worst	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

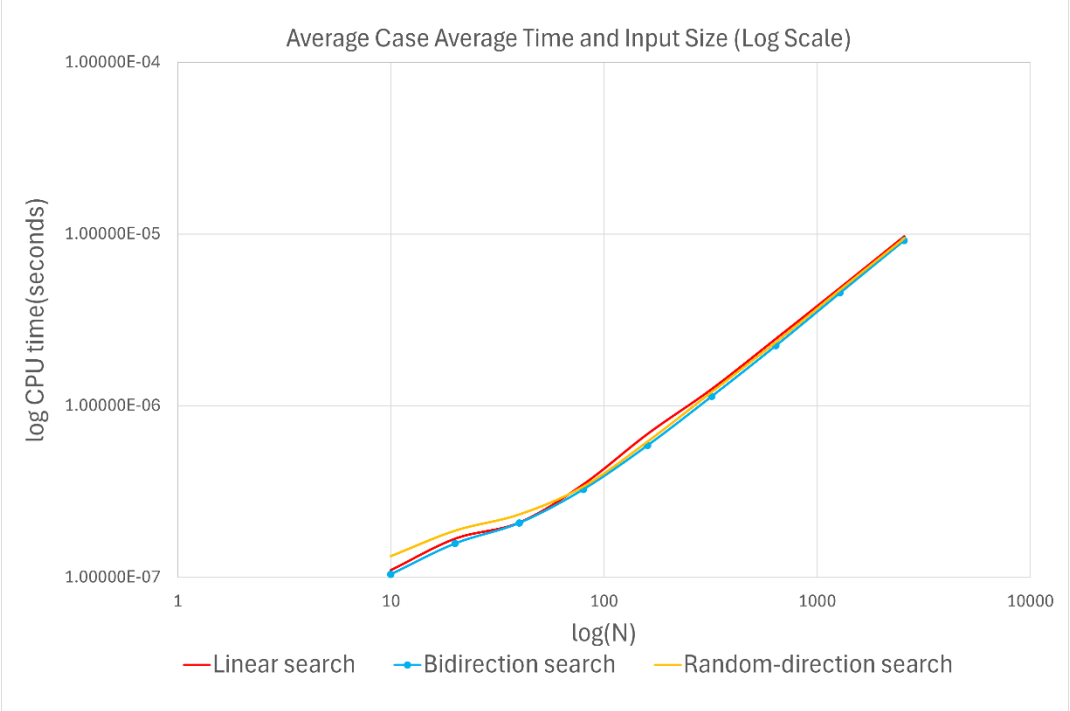
(fast to slow): Bidirection search > Linear search > Random-direction search

Results and observations

1. Results in average case:

average CPU time (seconds)			
N	Linear search	Bidirection search	Random-direction search
10	1.10006E-07	1.04189E-07	1.32799E-07
20	1.67799E-07	1.57404E-07	1.86896E-07
40	2.08306E-07	2.07353E-07	2.32291E-07
80	3.48252E-07	3.25799E-07	3.37946E-07
160	6.84649E-07	5.87088E-07	6.17787E-07
320	1.25050E-06	1.13379E-06	1.20079E-06
640	2.45638E-06	2.23800E-06	2.34387E-06
1280	4.86483E-06	4.54230E-06	4.72936E-06
2560	9.71212E-06	9.17948E-06	9.43161E-06





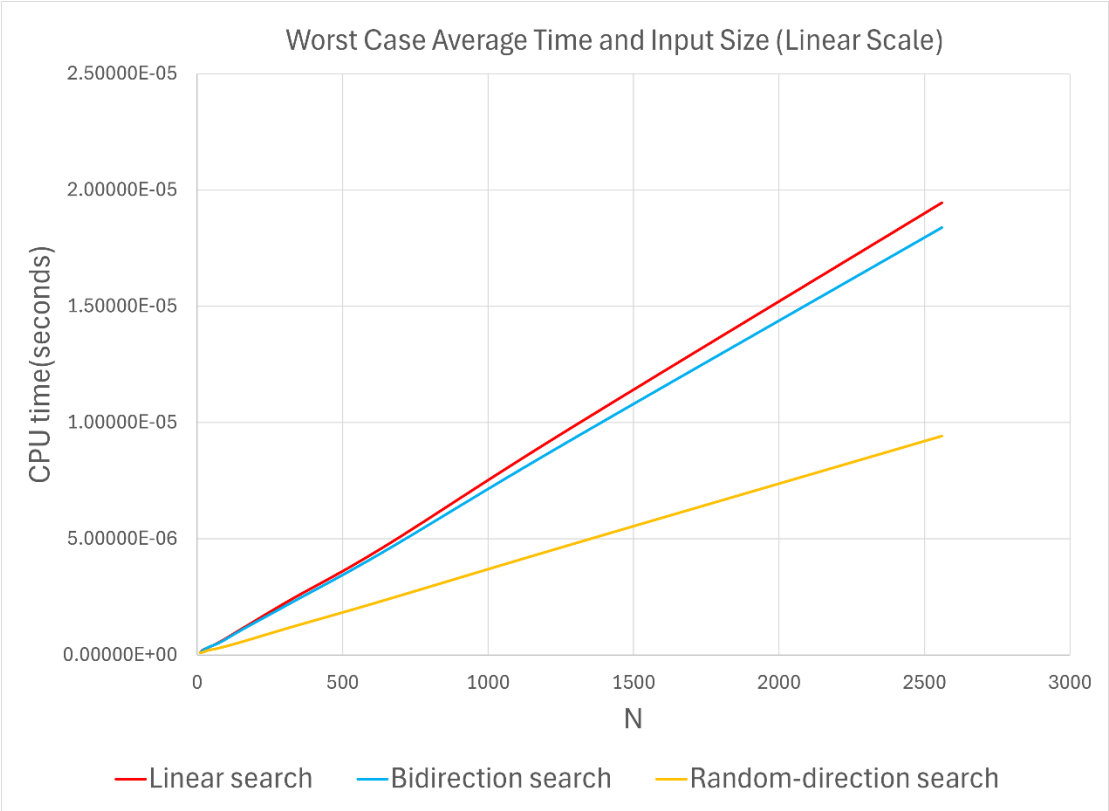
2. Observation in the average case:

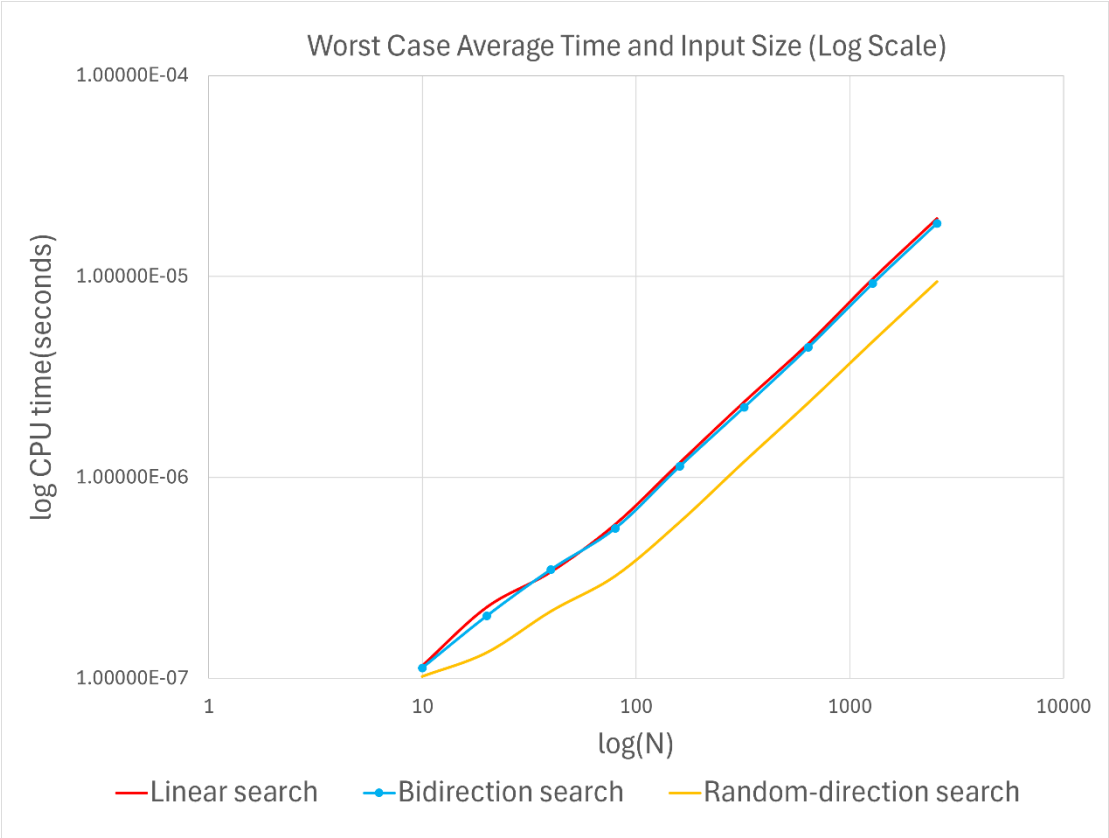
In average case, from log-log graph, we can find that the bidirection search is the fastest. When the number of data is small, the linear search is faster than the random-direction search. However, when the number of data gets larger, the random-direction search is little faster than the linear search. I think that it is because the random-direction search sometimes selects a more efficient search direction by chance, so it reduces the number of comparisons needed in some cases.

When the data is small, the overhead of randomly choosing a direction may not provide a significant advantage over linear search. However, as the data grows larger, the probability of randomly starting closer to the target increases, so it leads to a slightly better average performance compared to linear search.

3. Results in the worst case:

worst-case CPU time (seconds)			
N	Linear search	Bidirection search	Random-direction search
10	1.15738E-07	1.12777E-07	1.02239E-07
20	2.26169E-07	2.04370E-07	1.34280E-07
40	3.39570E-07	3.48336E-07	2.15800E-07
80	5.85603E-07	5.58305E-07	3.23250E-07
160	1.18422E-06	1.13393E-06	5.98940E-07
320	2.37326E-06	2.23059E-06	1.19567E-06
640	4.64110E-06	4.44147E-06	2.35056E-06
1280	9.72818E-06	9.21467E-06	4.74063E-06
2560	1.94483E-05	1.83904E-05	9.42632E-06


Worst Case Average Time and Input Size (Linear Scale)
2.50000E-05
2.00000E-05
1.50000E-05
1.00000E-05
5.00000E-06
0.00000E+00
CPU time(seconds)
0
500
1000
1500
2000
2500
3000
N
Linear search
Bidirection search
Random-direction search


Worst Case Average Time and Input Size (Log Scale)
1.00000E-04
1.00000E-05
1.00000E-06
1.00000E-07
log CPU time(seconds)
1
10
100
1000
10000
log(N)
Linear search
Bidirection search
Random-direction search

4. Observation in the worst case:

In the worst case, we can find that random-direction search is much faster than other two algorithms. I think that it is because in the worst case, the linear search has to scan the entire data sequentially until it finds the last element, and the bidirectional search must expand from both ends and meet in the middle. However, the random-direction search may sometimes get lucky and choose a more efficient path, so it can reduce the number of comparisons needed.

Even in the worst case, the random-direction search does not always follow a strict order like linear or bidirectional search, which allows it to occasionally find the target faster than the other two methods. This randomness introduces variability, and while it does not guarantee an optimal search time, it avoids the deterministic worst-case scenario of linear and bidirectional searches.

Conclusion

1. Time complexity:

	Linear search	Bidirection search	Random-direction search
Best	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Average	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Worst	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

2. Actual CPU time comparison (average case, fast to slow):

Small data: Bidirection search > Linear search > Random-direction search

Large data: Bidirection search > Random-direction search > Linear search

3. Actual CPU time comparison (worst case, fast to slow):

Random-direction search> Bidirection search ~== Linear search