Laboratory Project 1
Performance Measurement

Programmer: **Jiahui Shi**
Tester: **Yingchu Sun**
Writer: **Zecheng Qian**

Date: 2017-10-21

Contents

1 Introduction

1.1 Problem Description

Performance measurement is the process to calculate the algorithm run time. Generally, algorithm run time will differ on different problem scales and different chosen algorithms. With regard to the problem in the project, our task is to count processor time of finding a certain number in a given ordered list. In the program, we use two algorithms, they are "sequential search" and "binary search". And we implement an iterative version and a recursive version respectively.The time complexity is O(log n) for binary search and O(n) for sequential search, which will be elaborated below more specifically. That means sequentiall search is no long appropriate when the problem scale is efficiently large.

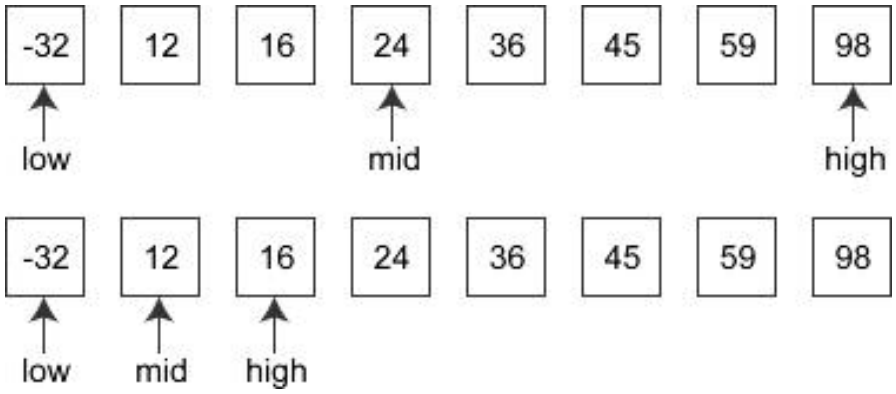

Figure 1: Binary Search

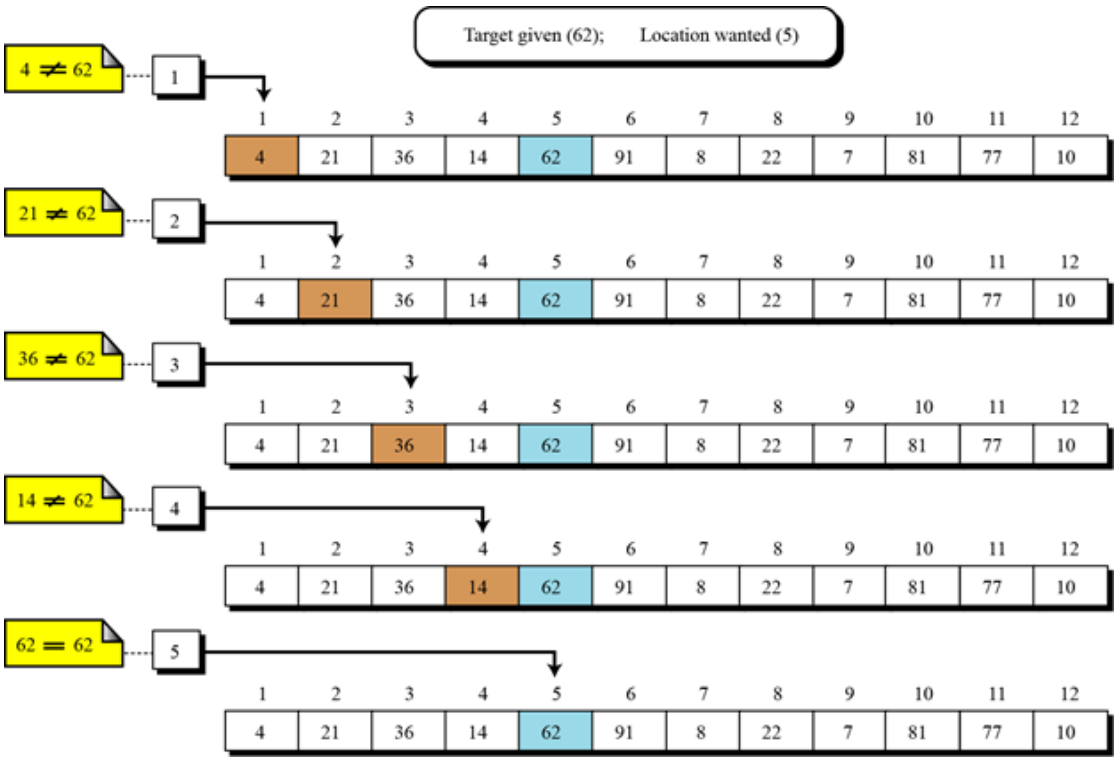

Figure 2: Linear Search

In reality, it is necessary to analyze the algorithm efficiency because we will encounter problems with a large amount of data more or less. Such as search engine algorithms, database retrieval and artificial intelligence. In that case, if an efficient algorithm is proposed, problems will be simplified and the program will running faster.

2 Algorithm Specification

In the program, we use two algorithms, they are "sequential search" and "binary search". And we implement an iterative version and a recursive version respectively. Four functions are studied. And we do some preliminary work to make the program proceed more smoothly.

Specifically, we want to calculate the worst run time of algorithms. That is to say, it is necessary to give a set of appropriate data which can provide a worst case for algorithms. So we use a list of ordered N integers from 0 to N-1, and we attempt to seek number N in the list which does not exist. Besides, sometimes the run time is too little

to doing statistics, so we let the program iterate K times and then calculate the single running time.

Actually, "binary search" performs better than "sequential search" in O(log n) if the list is ordered. In terms of iteration and recursion, ordinarily, recursion will take up a lot of stack space and will provide low operating efficiency mainly because a lot of function calls. So we do not use recursive algorithm if a better algorithm exists.

As for the main data structure, we use single linked list to solve the problem. Because single linked list is distinctive in its scalability. But it show low efficiency in memory access and have to follow the order.

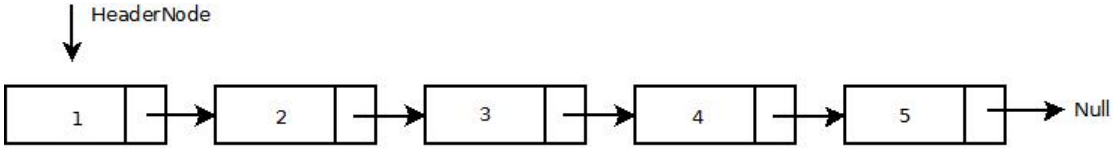

Figure 3: Single Linked List

2.1 pseudo-code

The pseudo-code of main function is following.

```
function main()
{
   input choice;   /*choose which function you want to
       use*/
   input N, M and K;   /*N is the number of data, M is
       the value we are looking for*/
                   /*K is the number of iterations*/
   a = GenerateLinkList(N);  /*a is the list which
      stores the datas*/
   start clock;   /*start the timer*/
   switch(choise){
      case 1:
         for(i from 1 to K)  /*run K times*/
         seqsearch_iteration;
         break;
      case 2:
         for(i from 1 to K)   /*run K times*/
         seqsearch_recursion;
         break;
      case 3:
         for(i from 1 to K)  /*run K times*/
         binsearch_interation;
         break;
      case 4:
         for(i from 1 to K)  /*run K times*/
         binsearch_recursion;
         break;
      default:
         exit;
```

```
            break;
    }
    stop clock;
    ticks = start - clock;   /*get ticks of running the
        program*/
    duration = ticks / K;  /*get total time of running
       the program*/
    print;
}
```

The pseudo-code of sequential search of iteration is following.

```
function sequential_iteration(pointer a, int N, int M)
{
    int i;
    for(i form 0 to N-1 )  /*traversal*/
    {
        if a[i]=M then
            return i;
        else
            next i;
        again;
    }
    if i=N
        return -1;  /*flag of not found*/
}
```

The pseudo-code of sequential search of recursion is following.

```
function sequential_recursion(pointer a, int M)
{
   if a[0]=M then
      return times of itetation; /*equal to the
         subscript of the element to be found*/
   else if a[0]=-1
      return -1;  /*flag of not found*/
   else
      sequential_recursion(next pointer a, M);
}
```

The pseudo-code of sequential search of iteration is following.

```
function binsearch_iteration(pointer a, int N, int M)
{
   int high, low, mid;

   while low<=high do
      mid = (high+low)/2;
      if a[mid]=M then
         retrun mid;
      else if a[mid]<M then
         low = mid+1;
      else
         high = mid-1;
      end

   return -1; /*flag of not found*/
}
```

The pseudo-code of sequential search of recursion is following.

```
function binsearch_recursion(pointer a, int high, int
   low, int M)
{
   int mid;
   mid = (hign+low)/2;
   if low>hign then
      return -1;  /*flag of not found*/
   else
      if a[mid]=M
         then returrn mid;
      else if a[mid]<M then
         binsearch_recursion(pointer a, high, mid+1, M
            )
      else
         binsearch_recursion(pointer a, mid-1, low, M)
}
```

2.2 Postscripts

The whole idea is given above in the pseudo-code, and more details will be available in the appendix which provides the source code.

3 Testing Results

3.1 Testing datas

The test results is listed in the following table.

	N	100	500	1000	2000	4000	6000	8000	10000
Binary Search (iterative version)	Iterations (K)	1.00E+08	1.00E+08	1.00E+08	1.00E+08	1.00E+08	1.00E+08	1.00E+08	1.00E+08
	Ticks	3906	5125	5748	6402	7048	7778	7823	8374
	Total Time (sec)	3.906	5.125	5.748	6.402	7.048	7.778	7.823	8.374
	Duration (sec)	3.91E-08	5.13E-08	5.75E-08	6.40E-08	7.05E-08	7.78E-08	7.82E-08	8.37E-08
Binary Search (recursive version)	Iterations (K)	1.00E+08	1.00E+08	1.00E+08	1.00E+08	1.00E+08	1.00E+08	1.00E+08	1.00E+08
	Ticks	4640	6622	6875	7562	8203	8984	9016	9796
	Total Time (sec)	4.64	6.622	6.875	7.562	8.203	8.984	9.016	9.796
	Duration (sec)	4.64E-08	6.62E-08	6.88E-08	7.56E-08	8.20E-08	8.98E-08	9.02E-08	9.80E-08
Sequential Search (iterative version)	Iterations (K)	1E+07	1E+07	1.00E+06	1.00E+06	1.00E+06	5.00E+05	5.00E+05	1.00E+05
	Ticks	2843	13699	2735	5438	10964	8203	10921	2734
	Total Time (sec)	2.843	13.699	2.735	5.438	10.964	8.203	10.921	2.734
	Duration (sec)	2.84E-07	1.37E-06	2.74E-06	5.44E-06	1.10E-05	1.64E-05	2.18E-05	2.73E-05
Sequential Search (recursive version)	Iterations (K)	5.00E+06	1.00E+06	1.00E+06	1.00E+06	5.00E+05	1.00E+05	1.00E+05	1.00E+05
	Ticks	2913	2981	5921	12101	12036	3641	4968	6544
	Total Time (sec)	2.913	2.981	5.921	12.101	12.036	3.641	4.968	6.544
	Duration (sec)	5.83E-07	2.98E-06	5.92E-06	1.21E-05	2.41E-05	3.64E-05	4.97E-05	6.54E-05

Figure 4: Testing Results

3.2 Testing process

I carefully selected the corresponding value of K, which shouldn not be too long or too short. In order to ensure the accuracy of the Duration, the total time is controlled between 2 seconds and 10 seconds.

4 Analysis and Comments

4.1 General Performance

Because the time of one duration is much shorter than a tick, the iterations are very large.In order to make the difference between different Algorithms more clearly, I make a line chart as follow:

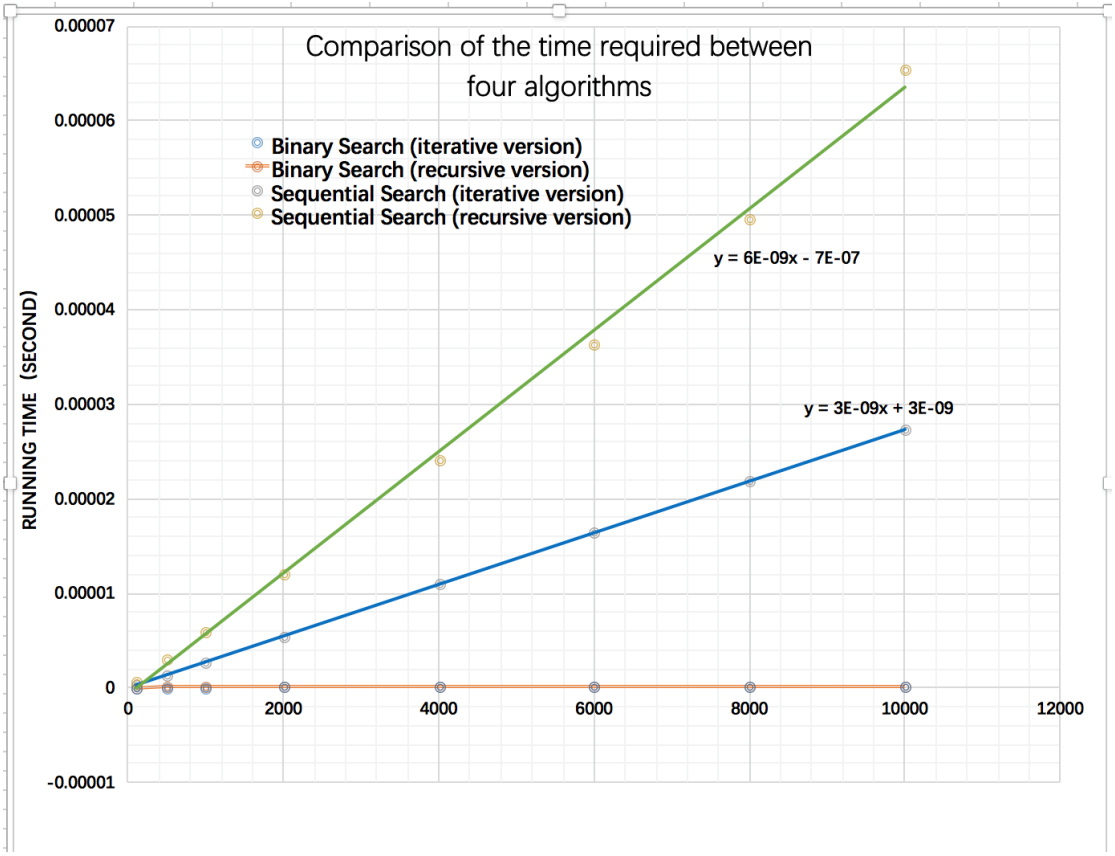

Figure 5: Comparison

It is easy to find that Binary Search is much quicker than Sequential Search at the worst run time of algorithms. And recursive version is a little bit more slowly than iterative version. It meets our expectation that "binary search" performs better than "sequential search" in O(log n) if the list is ordered. And recursion will take up a lot of stack space and will provide low operating efficiency mainly because a lot of function calls.

But, as you can see, the time line of the Sequential Search coincides with the X axis, which means we should analysis two algorithms independently to get deeper understanding of two algorithms.

4.2 The Analysis of Sequential search

Using the Excel's fitting function, we can also work out the linear equation of Sequential Search. The time complexity of the iterative version satisfies the equation:

$$y = 3E - 09x + 3E - 09$$

And the time complexity of the recursive version satisfies the equation:

$$y = 6E - 09x - 7E - 07$$

So, we can know that the time complexity of the two versions are both O(N), the coefficients of the recursive version is twice as much as the coefficients of the iterative version.

But, how does this happen?

```
//iterative version of sequential search
int seqsearch_iteration(int* a,int n,int m)
{
    int i;

    for(i=0;i<n;i++){
        if(a[i]==m){
            return i;
            break;
                terminates
        }
    }
    if(i==n)
    return -1;
}
```

Figure 6: Code

We can see that one loop body appears in this function, And for each loop, the time complexity is O (1). So the time complexity of the whole loop is O (n). And the space complexity is O (1).

```
//recursive version of sequential search
int seqsearch_recursion(int* a,int m)
{
    static int result=-1;

    if(*a==m)
    result=*a;
    else{
        while(*(a+1)>=0){
            result=seqsearch_recursion(++a,m);
            break;
        }
    }
    return result;
}
```

Figure 7: Code

For recursive version, there is a recursion in each loop, So, the time complexity is still O (1) for each loop. We can easily calculate the time complexity is O (n). And the space complexity is O (n).

4.3 The Analysis of Binary search

Though the regression equation, we know that the time complexity of the two versions are both O(logN). And the recursive version satisfies the equation:

$$y = 1E - 08ln(x) - 1E - 09$$

The iterative version satisfies the equation:

$$y = 1E - 08ln(x) - 7E - 09$$

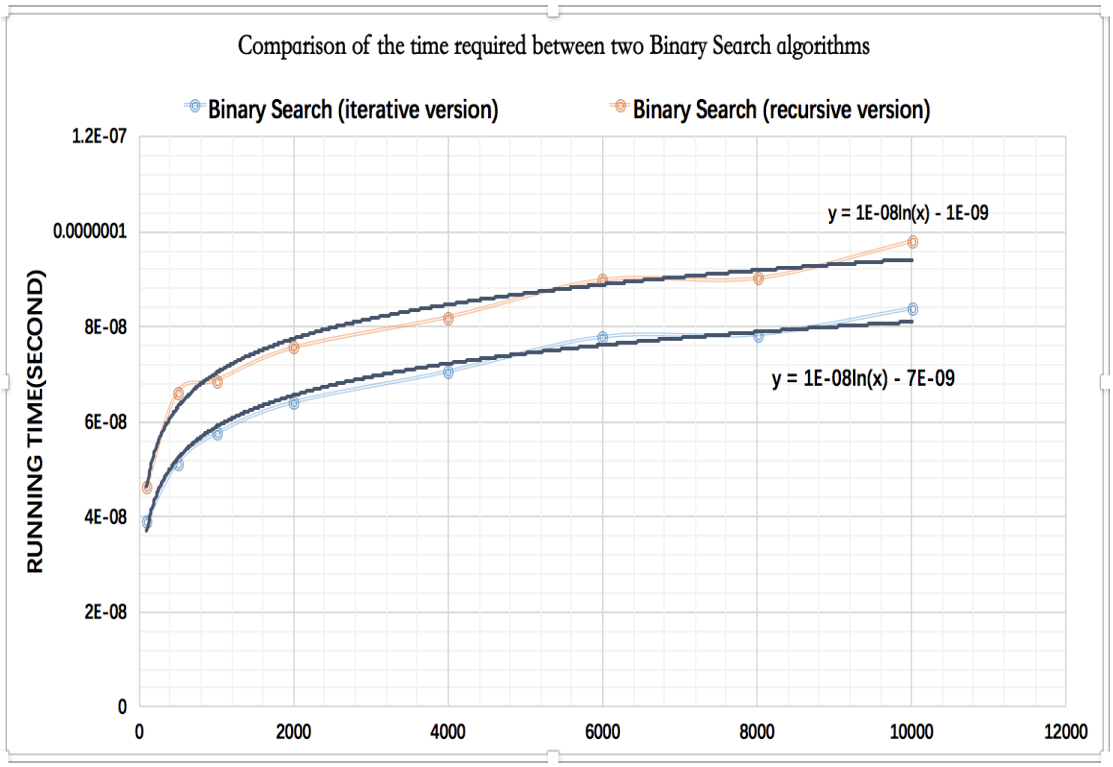

Figure 8: Comparison

The Binary search compares the keys with the middle elements. If the key is less than the middle key, the search continues in the left array, and if it is greater than the search on the right array, the middle key is the key we are looking for. At the worst case, the times(K) of search satisfies the equation: $2^K = N$ So, the time complexity is O (log(n)).

We can also see that the time spent in opening space is constant. So, the recursive version of the binary search is also considerable for this question.

```
//interative version of binary search
int binsearch_iteration(int* a,int n,int m)
{
    int high,low,mid;

    low=0;
    high=n-1;
    while(low<=high){
        mid=(high+low)/2;
        if(a[mid]==m)
        break;
            terminates
        else if(a[mid]<m)
        low=mid+1;
        else
        high=mid-1;
    }
    return low>high?-1:mid;
}
```

Figure 9: Code

5 Appendix: Source Code in C

Main:

```
int main()
{
   int repeat,ri;
   int choice;
   int m,n,i,k;
   int result;
   int *a;

   printf("Choose_the_fuction\n"); //suggest users to
      choose a fuction to test
   printf("1-seqsearch_iteration_2-seqsearch_recursion
      \n"); //4 functions to choose
```

```
11      printf("3-binsearch_iteration_4-binsearch_recursion
           \n");
12      scanf("%d",&choice);
13
14      printf("\nInput_testing_times:\n"); //suggest users
            to input testing times
15      scanf("%d",&repeat);
16
17      for(ri=1;ri<=repeat;ri++){
18         printf("\nInput_size_of_the_array:\n");  //
              suggest users to input size of the array
19         scanf("%d",&n);
20         a=(int*)malloc((n+1)*sizeof(int*));
21         for(i=0;i<n;i++){
22            a[i]=i;       //given a list of ordered n
                 integers, numbered from 0 to n-1
23         }
24         a[i]=-1;  //assign a[i] a negative value to end
              recursion
25         printf("\nInput_the_number_you_want_to_search:\n
              ");   //suggest users to input the nunber to
              be searched
26         scanf("%d",&m);   //input the number to be
              searched
27         printf("\nInput_cycle_times_of_calling_function
              :\n");  //suggest users to input the cycle
              times
28         scanf("%d",&k);  //repeat the function calls for
               k times to obtain a total run time
29         start=clock();    //records the ticks at the
              beginning of the function call
30
```

```
        for(i=0;i<k;i++){
            switch (choice){
                case 1:result=seqsearch_iteration(a,n,m);
                    break;
                case 2:result=seqsearch_recursion(a,m);
                    break;
                case 3:result=binsearch_iteration(a,n,m);
                    break;
                case 4:result=binsearch_recursion(a,n-1,0,
                    m);break;
            }
        }           //call for function

        stop=clock();  //records the ticks at the end of
            the function call

        if(result==-1)
        printf("\nNOT_FOUND\n");  //m is not in the list
        else
        printf("\n%d\n",result);  //m is in the list

        free(a);
        a=NULL;
        Ticks=(double)(stop-start);
        Duration=Ticks/CLK_TCK;
        printf("Ticks=%f\n",Ticks);
        printf("Duration=%f\n",Duration);
    }
}
```

Functions:

```
//iterative version of sequential search
int seqsearch_iteration(int* a,int n,int m)
{
   int i;

   for(i=0;i<n;i++){  //iterate through the list
      if(a[i]==m){
         return i;
         break;  //once m is founded in the list, the
            loop terminates
      }
   }
   if(i==n)
   return -1;  //m is not founded in the list
}


//recursive version of sequential search
int seqsearch_recursion(int* a,int m)
{
   static int result=-1;   //if m is not founded,
      result will remain -1

   if(*a==m)
   result=*a;
   else{
      while(*(a+1)>=0){  //*(a+1) is in the ordered n
         integers
         result=seqsearch_recursion(++a,m);  //check
            the next integer
         break;
```

```
28          }
29      }
30      return result;
31  }
32
33
34  //interative version of binary search
35  int binsearch_iteration(int* a,int n,int m)
36  {
37      int high,low,mid;
38
39      low=0;
40      high=n-1;
41      while(low<=high){
42          mid=(high+low)/2;   //divide the list into two
            parts
43          if(a[mid]==m)
44          break;        //once m is founded in the list,
            the loop terminates
45          else if(a[mid]<m)
46          low=mid+1;    //search the latter part
47          else
48          high=mid-1;   //search the fisrt part
49      }
50      return low>high?-1:mid;  //if low>high, m is not
        founded in the list
51  }
52
53
54  //recursive version of binary search
55  int binsearch_recursion(int* a,int high,int low,int m)
56  {
```

```
57      int mid;
58
59      while(low<=high){
60          mid=(high+low)/2;          //divide the list into
            two parts
61          if(a[mid]==m)
62          break;   //once m is founded in the list, the
            loop terminates
63          else if(a[mid]<m)
64          return binsearch_recursion(a,high,mid+1,m);
            //search the latter part
65          else
66          return binsearch_recursion(a,mid-1,low,m);
            //search the fisrt part
67      }
68      return low>high?-1:mid;      //if low>high, m is not
         founded in the list
69  }
```

Declaration:

We hereby declare that all the work done in this project titled "Performance Measurement" is of our independent effort as a group.

Duty Assignments:

Programmer: **Jiahui Shi**

Tester: **Yingchu Sun**

Writer: **Zecheng Qian**