FDS Project

Performance Measurement

姓名:时奇旭

学号:3230104643

Date 2024-3-9

1 Introduction

What to be done:

The task is to implement both an iterative and a recursive version of "sequential search" and "binary search" algorithms,respectively.

- sequential search: scans through the list from left to fight
- binary search: each time divide the interval into tow parts and continue in the correct interval involving n since it's ordered.

Why:

Furthermore,it is necessary to measure the performances of these four functions when solving the problem of seeing whether N exists in a list consisting of N integers,numbered from 0 to N-1.

2 Algorithm Specification

- iterative binary search:

```
function iter_binary_search(list,n)
{
  while left<right do
  {
    mid = (left+right)/2
    if a[mid]>n then n is between left and mid so the right = mid-1
    if a[mid]<n then n is between mid and righet so the left = mid+1
    else return find!
  }
  return Notfind!
}
end function
```

- recursive binary search:

```
function rec_binary_search(list,n,left,right)
{
  if left>right then Notfind!
  if a[mid] > n then return rec_binary_search(list,n,left,mid-1)
  if a[mid] < n then return rec_binary_search(list,n,mid+1,right)
  else return find!
```

```
}
end function
```

- **iterative sequential search:**

```
function iter_sequential_search(list,n)
{
  Examine a[0] to a[n-1] to find N
  for i=0 upto n do
  {
    if a[i] == n then return find!
  }
  return Notfind!
}
end function
```

- **recursive sequential search:**

```
function rec_sequential_search(list,n,index)
{
  Examine a[0] to a[n-1] to find N
  if index == n then return Notfind!
  if a[index] == n then return find!
  else return rec_sequential_search(list,n,index+1)
}
end function
```

- **the main program:**

```
use the library time.h to check the performance of each function as shown in the diagram
```

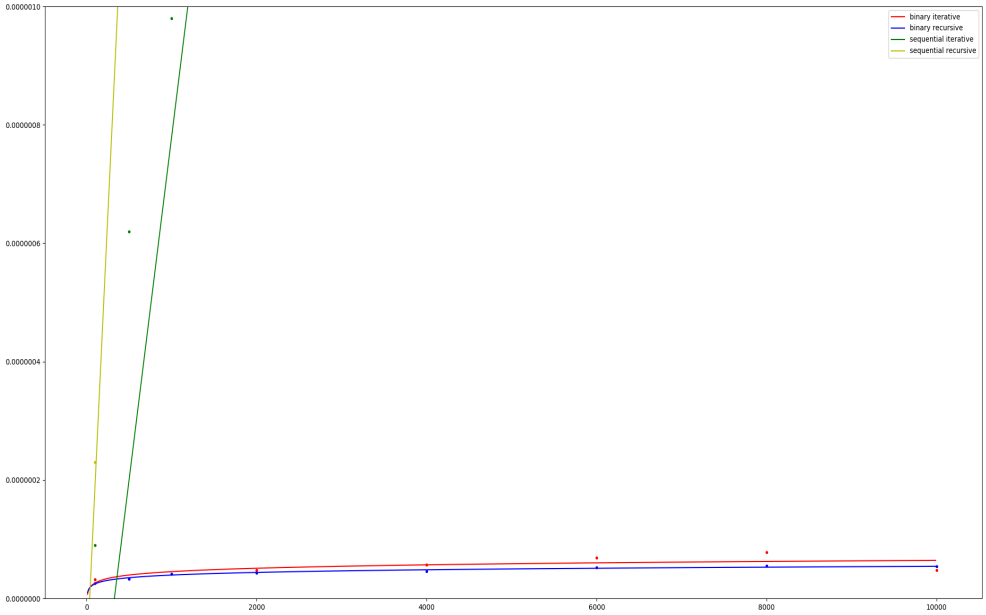
3 Testing Results

We stipulate inputs as follows:

Case	Scale	status
1	100	pass
2	500	pass
3	1000	pass
4	2000	pass
5	4000	pass
6	6000	pass
7	8000	pass
8	10000	pass

	N	100	500	1000	2000	4000	6000	8000	10000
Binary Search (iterative version)	**Iterations(K)**	**10000000**	**10000000**	**10000000**	**10000000**	**10000000**	**10000000**	**10000000**	**10000000**
	Ticks	**323**	**333**	**414**	**478**	**573**	**687**	**779**	**476**
	Total Time(sec)	**0.323**	**0.333**	**0.414**	**0.478**	**0.573**	**0.687**	**0.779**	**0.476**
	Duration(sec)	**3.2e-08**	**3.3e-08**	**4.1e-08**	**4.8e-08**	**5.7e-08**	**6.9e-08**	**7.8e-08**	**4.8e-08**
Binary Search (recursive version)	**Iterations(K)**	**10000000**	**10000000**	**10000000**	**10000000**	**10000000**	**10000000**	**10000000**	**10000000**
	Ticks	**262**	**325**	**410**	**430**	**459**	**516**	**552**	**540**
	Total Time(sec)	**0.262**	**0.325**	**0.41**	**0.430**	**0.459**	**0.516**	**0.552**	**0.540**
	Duration(sec)	**2.6e-08**	**3.3e-08**	**4.1e-08**	**4.3e-08**	**4.6e-08**	**5.2e-08**	**5.5e-08**	**5.4e-08**
Sequential Search (iterative version)	**Iterations(K)**	**100000**	**100000**	**100000**	**100000**	**100000**	**100000**	**100000**	**100000**
	Ticks	**11**	**62**	**98**	**171**	**349**	**572**	**915**	**1175**
	Total Time(sec)	**0.011**	**0.062**	**0.098**	**0.171**	**0.349**	**0.572**	**0.915**	**1.175**
	Duration(sec)	**1.1e-07**	**6.2e-07**	**9.8e-07**	**1.71e-06**	**3.49e-06**	**5.72e-06**	**9.15e-06**	**1.18e-05**
Sequential Search (recursive version)	**Iterations(K)**	**100000**	**100000**	**100000**	**100000**	**100000**	**100000**	**100000**	**100000**
	Ticks	**23**	**139**	**233**	**642**	**1327**	**1724**	**2306**	**3117**
	Total Time(sec)	**0.023**	**0.139**	**0.233**	**0.642**	**1.327**	**1.724**	**2.306**	**3.117**
	Duration(sec)	**2.3e-07**	**1.39e-06**	**2.33e-06**	**6.42e-06**	**1.32e-05**	**1.72e-05**	**2.3e-05**	**3.11e-05**

Figure 1 Performance on extreme cases



4 Analysis and Comments

- **iterative_binary_search:**
  - **Time complexity:**
    In each loop,the binary search halves the search space,so the time complexity is O($\log N$)
  - **Space complexity:**
    It only uses a fixed amount of space to store the start,middle and end,so the space complexity is O(1)
- **recursive_binary_search:**
  - **Time complexity:**
  - **In each recursive call,the binary search halves the search space,so the time complexity is O($\log N$)**
  - **Space complexity:**
  - **As mentioned above,the function is called recursively of $\log N$ times and additional space is required to store the funciont call stack frame,so the space conplexity is O($\log N$)**
- **iterative_sequential_search:**
  - **Time complexity:**
    The loop runs for N times and the complexity of each loop body is O(1),so the total time complexity is O(N)
  - **Space complexity:**
    It only uses a fixed amount of space to store the current position,so the space complexity is O(1)
- **recursive_sequential_search:**
  - **Time complexity:**
    The function is called for N times ,so the total time complexity is

O(N)

- Space complexity:
  The function is called recursively for N times,and additional space is required to store the function call stack frame,so the space complexity is O(N)

- Further possible comments:
  As is shown in the plot,the binary search is better because it can halve the internal every time.But in fact,a hash table can provide O(1) search time in the average case.

5 Appendix: Source Code

```
#include<stdio.h>
#include<time.h>

clock_t start,stop;
//clock_t is a built-in type for processor time(ticks)
double duration;
//records the run time(seconds) of a function

int iter_sequential_search(int a[],int n){
    for(int i = 0;i<n;i++){
        if(a[i] == n) return 1;
    }
    return 0;
}
int rec_sequential_search(int a[],int n,int i){
    //i is the index of the array
    if(i >= n) return 0;
    //base case
    if(a[i] == n) return 1;
    return rec_sequential_search(a,n,i+1);
    //search the next element
}
int iter_binary_search(int a[],int n){
    int left = 0;
    int right = n-1;
    while(left<right){
        int mid = (left+right)/2;
        if(a[mid] > n) right = mid-1;
        /*if the middle element is greater than n,
```

```
        then it implies that n is in the left half of the array*/
        else if(a[mid] < n) left = mid+1;
        else return 1;
        // if the middle element is equal to n,then return 1
    }
    return 0;
}
int rec_binary_search(int a[],int n,int left,int right){
  //left and right are used to define the range of the array
    if(left>right) return 0;
    int mid = (left+right)/2;
    if(a[mid]>n) return rec_binary_search(a,n,left,mid-1);
    else if(a[mid]<n) return rec_binary_search(a,n,mid+1,right);
    /*if the middle element is less than n,
    then it implies that n is in the right half of the array*/
    else return 1;
    // if the middle element is equal to n,then return 1
}

int main(){
    int n;
    scanf("%d", &n); //input the size of the array
    int a[n]; //declare an array of size n
    for(int i = 0;i<n;i++)  a[i] = i; //initialize the array
    int k = 100000; //k times to iterate to obtain a total run time
    double total_ticks = 0; //total ticks
    double total = 0; //total run time
    // measure iter_sequential_search function
    for(int i = 0;i<k;i++){
        start = clock();
        //record the ticks at the start of the function
        iter_sequential_search(a,n); // call the function
        stop = clock();
        //record the ticks at the end of the function
        duration = ((double)(stop-start))/CLK_TCK;
        //convert ticks to seconds
        total += duration;
        total_ticks += (double)(stop-start);
    }
    printf("iter_sequential_search:Ticks:%f,Total Time:%f,\
    Duration:%.9f\n",total_ticks,total,total/k);
```

```
// measure rec_sequential_search function
total = 0; // reset total
total_ticks = 0;//reset total ticks
for(int i = 0;i<k;i++){ //iterate k times
    start = clock();
    rec_sequential_search(a,n,0); //call the function
    stop = clock();
    duration = ((double)(stop-start))/CLK_TCK;
    total += duration;
    total_ticks += (double)(stop-start);
}
printf("rec_sequential_search:Ticks:%f,Total Time:%f,\
Duration:%.9f\n",total_ticks,total,total/k);
k = 10000000; // for binary_search
// measure iter_binary_search function
total = 0; // reset total
total_ticks = 0; // reset total ticks
for(int i = 0;i<k;i++){ // iterate k times
    start = clock();
    iter_binary_search(a,n);
    stop = clock();
    duration = ((double)(stop-start))/CLK_TCK;
    total += duration;
    total_ticks += (double)(stop-start);
}
printf("iter_binary_search:Ticks:%f,Total Time:%f,\
Duration:%.9f\n",total_ticks,total,total/k);
// measure rec_binary_search function
total = 0; // reset total
total_ticks = 0; // reset total ticks
for(int i = 0;i<k;i++){
    start = clock();
    rec_binary_search(a,n,0,n-1);
    stop = clock();
    duration = ((double)(stop-start))/CLK_TCK;
    total += duration;
    total_ticks += (double)(stop-start);
}
printf("rec_binary_search:Ticks:%f,Total Time:%f,\
Duration:%.9f\n",total_ticks,total,total/k);
```

```
    return 0;
}
```

6 Declaration

I hereby declare that all the work done in this project titled "Performance Measurement (Search)" is of my independent effort