|  |  |
| --- | --- |
| **CSCI 2100E Spring 2020** | Student Name: \_Tam Yi Ki\_\_\_\_ Student Id: \_1155126127\_\_ |

# Lab Exercise #5

*Lab exercises are always due 2**weeks after the day of the lab. Please always ask for help from tutors if you are stuck!*

*Please fill in the lab sheet and submit the completed Word doc file to blackboard.* *Places you need to fill in or work on are marked in red.*

## Demo

Today you will try to do some empirical work on searching algorithms! Below you will find a framework for generating random array of sorted integers. It is a bit crude but it has the advantage of being simple. Your instructor will explain how it works.

|  |
| --- |
| #include <stdio.h> #include <stdlib.h>  int fillArray(int a[], int size, double skipProb, int skipMagnitude) {  // a crude, far-from-perfect random sorted array generation function  // we use skipProb and skipMagnitude to simulate "not found" cases    int i;  int x = -1;  for (i=0;i<size;i++) {  x++; // x must increment  if ((rand() % 100) < (skipProb \* 100))  x += skipMagnitude; // a chance to advance a few numbers    a[i] = x;    }  return size + (int)(skipProb \* skipMagnitude \* size) - 1; }  int main() {  srand(0); // change random seed to get different numbers  int a[100] = {0};   int size = 20;  int max = fillArray(a,size,0.5,2);    // a demonstration of generated array  int i;  for (i=0;i<size;i++) {  printf("%d ",a[i]);   }  printf("\nSearch testing range: 0 to %d\n",max);  printf("Theoretical chance of number found in search test: %.2f\n", size/(max+1.0));   return 0; } |

## Problem 1 [Linear Search with Early Stop]

Below you will find an implementation of an algorithm - linear search with early stop.

A linear search is just a naive search - you go through each of the elements of a list one by one. Early stop works only on sorted list. Early stop means, instead of going through the whole list, we will stop when your number to search can no longer be possibly found in the rest of the list. For instance, if my array a is sorted in ascending order, my linear search for a number x can stop when we reach an element a[i] > x.

Notice if no match is found, the convention for search functions is to return -1.

|  |
| --- |
| #include <stdio.h>   // Linear search with early stop - just search one by one;  // It assumes list is sorted in ASCENDING ORDER // this function returns index of item found, or -1 if no match is found  int comparisonsUsed = 0;   int linearSearch(int a[], int lastIndex, int x) {  // lastIndex is the last index of the array  // x is the number to search for in the array    int i = 0;  while (i <= lastIndex && a[i] <= x) {  comparisonsUsed++;  if (x == a[i]) return i;  i++;  }  return -1;  }   int main(void) {  int a[] = {0,1,4,5,6,7,9,11,12,15};  int size = 10;  int x = 10;  int foundIndex = linearSearch(a, size-1, x);  printf("Comparisons used: %d\n",comparisonsUsed);  if (foundIndex == -1)  printf("\* Element x is not found in list\n");  else   printf("\* Element x is found at index %d\n",foundIndex);  return 0; } |

You will notice that I am tracking the number of comparisons (between x and array element) used. Let's assume we will adopt that as the **unit time cost** of the linear search function.

a) In the example program above, x is equal to 10. What is the exact time cost of the linear search function in this specific case?

|  |
| --- |
| 7 |

b) If x is a uniformly distributed random integer variable in range of 0 to 15, what is the expected time cost of the linear search function?

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hint: There are total 16 possible value of x, and each of them have equal probability of 1/16   |  |  | | --- | --- | | Value of x | Time cost (# of comparisons) | | 0 | 1 | | 1 | 2 | | 2 | 3 | | 3 | 3 | | 4 | 3 | | 5 | 4 | | 6 | 5 | | 7 | 6 | | 8 | 7 | | 9 | 7 | | 10 | 8 | | 11 | 8 | | 12 | 9 | | 13 | 10 | | 14 | 10 | | 15 | 10 |   Expected time cost: 6 |

c) If a[] is now an array of size n with distinct elements in range of [0,max] in ascending order (max > n), and x is a uniformly distributed random variable in [0,max], which of the following will be a good estimate of the expected time cost of the linear search function?

|  |
| --- |
| A. 0 B. 1 C. n/2 D. max E. n |

d) **(optional)** With your knowledge in probabilities, try to derive the expected time cost in c) to confirm your guess.

|  |
| --- |
| *Please write your derivations and assumptions below:* |

## Problem 2 [Verify Your Guess/Derivation]

Using the random generation framework provided in the demo, let's try to verify your answer in Problem 1 c)/d)!

You can generate a sufficiently-good-random array using the fillArray function in demo, like this:

|  |
| --- |
| int a[100] = {0}; int size = 20; int max = fillArray(a,size,0.5,2); |

As in the demo, this setup will always generate an array of size 20 with elements often in range of [0,40] (with some minor error, which we will ignore as noise). Please perform the following:

1. Generate a random array of size 20 using above method
2. Search the array using linearSearch in Problem 1 with a random number x in range of [0,max]
3. Record the number of comparisons used (as in global variable comparisonsUsed)
4. Repeat 1 to 3 1000 times and find the average

Does the empirical value fit with your formula?

|  |
| --- |
| Average value found in experiment:  10.198  My guess/derived formula in Problem 1 c)/d):  (20+1)/2  Does it fit? What could explain the differences, if any? Not exactly fit, as the average value may be affected by the random x.  #include <stdio.h>  #include <stdlib.h>  #include<time.h>  int fillArray(int a[], int size, double skipProb, int skipMagnitude) {  // a crude, far-from-perfect random sorted array generation function  // we use skipProb and skipMagnitude to simulate "not found" cases    int i;  int x = -1;  for (i=0;i<size;i++) {  x++; // x must increment  if ((rand() % 100) < (skipProb \* 100))  x += skipMagnitude; // a chance to advance a few numbers    a[i] = x;    }  return size + (int)(skipProb \* skipMagnitude \* size) - 1;  }  int comparisonsUsed = 0;  int linearSearch(int a[], int lastIndex, int x)  {  // lastIndex is the last index of the array  // x is the number to search for in the array    int i = 0;  while (i <= lastIndex && a[i] <= x) {  comparisonsUsed++;  if (x == a[i]) return i;  i++;  }  return -1;  }  int main()  {  // change random seed to get different numbers  int a[100] = {0};  int size = 20;    int x;  int foundIndex;  srand(time(NULL));    for(int i = 0; i < 1000; i++){  int max = fillArray(a,size,0.5,2);  x = (rand()%max +1);  foundIndex = linearSearch(a, size-1, x);    }  printf("Average used = %lf", comparisonsUsed / 1000.0);    return 0;  } |

*(Actually if it is very different you should check your answer in Problem 1…)*

## 

## Problem 3 [Binary Search]

Below we have a simple implementation of binary search that is WRONG. Please help debug and fix it. You MAY NOT rewrite more than 10% of the whole program. (HINT: There are only two major bugs)

|  |
| --- |
| #include <stdio.h>    // Binary search function using recursion - this function  // should return -1 if no match is found  // assume a is always sorted in ascending order  int binarySearch(int a[], int left, int right, int x)  {  printf("Using binary search from left=%d to right=%d \n",left,right);  if (right >= left)  {  int i;  printf("Sublist considered: ");  for (i=left;i<=right;i++) {  printf("%d ",a[i]);  }  printf("\n");  int mid = left + (right - left)/2;    if (a[mid] == x)  return mid;    if (a[mid] > x) {  printf("Choosing left half to continue search...\n");  return binarySearch(a, left, mid-1, x);  } else {  printf("Choosing right half to continue search...\n");  return binarySearch(a, mid+1, right, x);  }      } else {  printf("No sublist to consider anymore...\n");  }    // not found!  return -1;  }    int main(void)  {  int a[] = {23, 31, 40, 52, 62, 71, 87, 91, 100, 101, 120, 131};  int size = 12;  int x = 91;  int result = binarySearch(a, 0, size-1, x);  if (result == -1)  printf("\* Element is not found in list\n");  else  printf("\* Element is found at index %d\n",result);  return 0;  } |

The program output, if the main function remains unchanged, should be like this after all bugs are fixed:

|  |
| --- |
| Using binary search from left=0 to right=11  Sublist considered: 23 31 40 52 62 71 87 91 100 101 120 131  Choosing right half to continue search... Using binary search from left=6 to right=11  Sublist considered: 87 91 100 101 120 131  Choosing left half to continue search... Using binary search from left=6 to right=7  Sublist considered: 87 91  Choosing right half to continue search... Using binary search from left=7 to right=7  Sublist considered: 91  \* Element is found at index 7 |

# Problem 4 [Empirical Search Performances]

Using the testing framework in Demo, as well as your results in Problem 2 and 3, please conduct the following experiment:

|  |
| --- |
| int max = fillArray(a,size,0.5,2); |

1. Fill up a random array of size = 20 using the above setup again
2. Search for a random number x in range of [1, max] using both binary search and linear search
3. Count the number of comparisons (search term with array elements) needed for each case, again as our **unit time cost** (you may need to create a few global variables and/or modify the linear and binary search functions a bit).
4. Repeat step 1 to step 3 1000 times and find out the **best, average and worst time costs** for binary search and linear search
5. Repeat step 1 to step 4 for size = 20, 40, 60, 80 and 100 and record the data. Put the data in the following table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Linear Search w/ Early Stop | | | Binary Search | | |
|  | Best | Average | Worst | Best | Average | Worse |
| size = 20 | 1 | 10.141 | 20 | 1 | 4 | 5 |
| size = 40 | 1 | 20.252 | 40 | 1 | 6 | 6 |
| size = 60 | 1 | 30.088 | 60 | 1 | 6 | 6 |
| size = 80 | 1 | 38.451 | 80 | 1 | 6 | 7 |
| size = 100 | 1 | 49.279 | 100 | 1 | 7 | 7 |

Please plot a graph (using Excel or similar programs) with the above data and show the empirical time complexity of linear search with early stop v.s. binary search against length of input (the array). Just copy as picture into the document.

|  |
| --- |
|  |

Does your result agree with the theoretical results for Binary Search, as mentioned in our lecture notes? What do you think are the flaws of our empirical method used above, if any?

|  |
| --- |
| I don’t agree of that, as the range of the size is too small. |

# 

# Problem 5 [Performance in Different Conditions]

Repeat the experiment in Problem 4 now with the following setup:

|  |
| --- |
| int max = fillArray(a,size,0.5,30); |

Here, the generated random arrays will have elements coming from a much larger range of numbers, i.e. **max >> size**. Please fill the new experiment data in the following tables.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Linear Search w/ Early Stop | | | Binary Search | | |
|  | Best | Average | Worst | Best | Average | Worse |
| size = 20 | 1 | 9.878 | 20 | 1 | 4 | 5 |
| size = 40 | 1 | 19.214 | 40 | 1 | 6 | 6 |
| size = 60 | 1 | 30.039 | 60 | 1 | 6 | 6 |
| size = 80 | 1 | 40.579 | 80 | 1 | 6 | 7 |
| size = 100 | 1 | 50.638 | 100 | 1 | 7 | 7 |

And the graph:

|  |
| --- |
|  |

Is there a part of the results substantially different from the results you get in Problem 4? If there is, what do you think is the reason? If not, why not? Anyhow, what do you think the implication is?

|  |
| --- |
| The averages of Linear Search of size 20 and 40 are smaller than before. It is because the range is larger but the array size does not change. |