CS 337: Algorithms: Design & Practice

Lab#1: Linear Search: Performance Comparison

Chapter 2 of Cormen describes four algorithms for Linear Search as follows:

Version 1: From page 13.

---

procedure linearSearch(A, n, x)

Inputs:

- A: An array (indexed from 0 onwards)
- n: the number of elements in A to search through
- x: the value being searched for.

Outputs: Either an index i for which A[i] = x, or a special value, -1

1. Set answer to -1.
2. For each index i=0 to n-1:
    a. If A[i] = x then set answer to i.
3. Return the value of answer

Version 2: From page 14 (Better Linear Search)

---

procedure betterLinearSearch(A, n, x)

Inputs & Outputs: Same as Version 1

1. For each index i=0 to n-1:
    a. If A[i] = x then return value of i.
2. Return -1

Version 3: From page 16: Sentinel Linear Search

---

procedure SentinelLinearSearch(A, n, x)

Inputs & Outputs: Same as Version 1

1. Last ← A[n-1]
2. A[n-1] ← x.
3. Set i ← 0.
4. While A[i] ≠ x:
    a. Increment i by 1
5. A[n-1] ← Last
6. If i < n-1 or A[n-1] = x then return value of i.
7. Otherwise return -1

Version 4: From page 24: Recursive Linear Search

---

procedure RecursiveLinearSearch(A, n, i, x)

Inputs: Same as Version 1 with an added parameter, i.
Outputs: The index of an element equaling x in the subarray from A[i] through A[n-1], or -1 if not found.

1. If i ≥ n then return -1.
2. Otherwise (i < n),
   if A[i] = x then return i.
3. Otherwise (i < n and A[i] ≠ x),
   return RecursiveLinearSearch(A, n, i+1, x).

Task#1: Implement each of the above algorithms in a programming language of your choice (C, Java, Go, Rust, or Python). Write a program that uses each of these using the following algorithm:

---

procedure main()

Inputs: None.
Outputs: For each search algorithm, it reports the following:

```
Searching array of size: N
Algorithm: linearSearch
Looking for element: x
Result: -1 or i

Algorithm: betterLinearSearch
Looking for element: x
Result: -1 or i

etc.
```

1. Initialize an array, A of integers of size 10 such that A[i] ← i.
2. Search for element, 7 in A using each of the four searches. It should report a result 7.
3. Search for element, 10 in A using each of the four searches. It should report a result -1.

Implement and test each search function one at a time. Once you have confirmed that each of your functions is working correctly proceed to the next task.

Task#2: Next, we want to compare the run time of each of these functions in the best, worst, and average cases. In order to do this, first modify the program from Task#1 to the following:

---

1. Initialize an array, A of random integers of size X. The integers should be chosen from the range 0…(10*X) so that repeats are relatively rare. X should be chosen so that the time required to find an item using algorithm Version 1 worse case is between 2 and 3 seconds.

2. Best case: Search for element at A[0] in A using each of the four searches.
3. Average case: Search for a random number in A using each of the four searches. The range of random numbers you search for should be at least 20X (so that some searches will fail).
4. Worst case: Search for a number that is NOT in A using each of the four searches. (How do you know that the number is not in A?)

Ensure that the programs are working correctly and that the programs exactly follow the above algorithms. Then insert timing commands to determine the execution time for each search. Run the program a sufficient number of times so as to get a good data set that will allow you to analyze and compare the performance of the searches against each other.

Note that Algorithm 4 may run into issues If it does, be sure you understand these issues and describe the issues and causes in your report. What are the issues? Are the issues consistent? (Be sure to precisely define "consistent".)

Extra Credit: (1) Implement the entire program in another programming language. Based on the performance data what conclusion(s) can you draw about the comparative speed of running programs in the different languages; (2) Run Version 2, timing the worst case for A starting at 4% of the size you determined for A and increasing by 4%. Ie. draw a performance plot. (3) Determine how many searches (average case) in 1 second?

Further Extra Credit: The recursive search algorithm is tail recursive. None of the Java, Go or Python implement tail call optimization. So, first, find a procedural language that implements tail call optimization (I can name at least 2 languages that do so). Implement recursive search in that language and explore the benefit of tail call optimization. For instance, does it make the speed equivalent to Algorithm 2?. While you are at it, merge in the ideas of sentinel linear search to recursive search.

Task#3: The Report

Write a formal report that uses the data obtained to compare the performance of the four searches. Rank the searches, based on your data from most efficient to least efficient, by run time. Does the performance match the expected efficiency as discussed in the text? Include in your report descriptions of how you determined X (and its value). Also, how you determined the size of the random strings you used.

The format for the report should be *at most* 5-pages using the formatting discussed in class on XXXX. In your report, clearly outline the purpose of the study, the methodology employed and a detailed presentation and analysis of the data obtained. End the report by summarizing your findings. Include a listing of all programs in the Appendix of the report.

What to hand in: A report as described above.

Helpful Notes

Generating a random number

In C, **stdlib** has the function **rand()** defined as follows:

```
int rand() – returns a random number between 0 and RAND_MAX
             RAND_MAX is defined in stdlib
```

In Java, you can use the **Random** class from **java.util** as follows:

```
import java.util.Random;
…
Random rand = new Random();    // Create a generator
…
int x = rand.nextInt(10);      // Generates a random number
                               // between [0..10)
```

In Python, the random module has the following function:

```
x = random.randrange(10)       // generates a random number
                               // between [0..10)
```

In Go:

```
    import "math/rand"
                               x = rand.Intn(10)
```

You can look up the language references for details on these and other ways of generating random numbers.

Timing pieces of code in C

```
#include<time.h>
…
long start = clock();
…code to be timed inserted here…
double duration = (clock() – start) / CLOCKS_PRE_SEC;
                  // duration is in seconds
```

See the lecture notes from last Thursday for timing in Python, Go and Java.

You can look up the language references for details on these and other ways of timing lines of code.