CISC-235

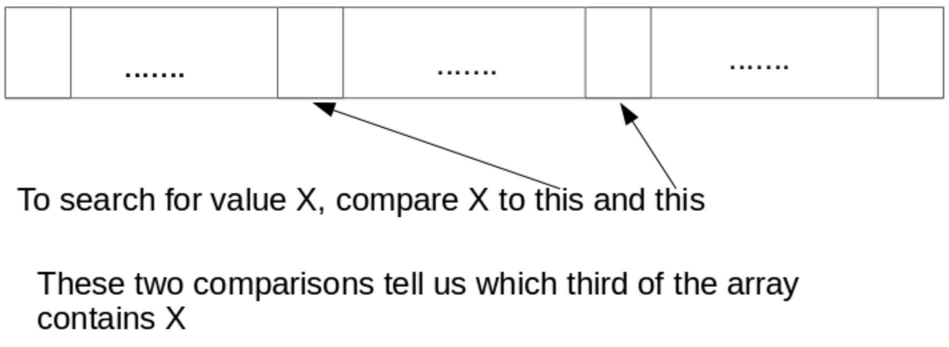
Assignment 1

January 6, 2020

The binary search algorithm for searching a sorted array is well known:

```
bin_search(A,first,last,target):
     # returns index of target in A, if present
     # returns -1 if target is not present in A
     if first > last:
          return -1
     else:
          mid = (first+last)/2
          if A[mid] == target:
               return mid
          else if A[mid] > target:
               return bin_search(A,first,mid-1,target)
          else:
               return bin_search(A,mid+1,last,target)
```

Binary search reduces the number of possible locations for the target value by (about) half each time, which makes it quite efficient. But we could eliminate **more** locations by looking at two values in the range A[first ... last]. If we look at the value 1/3 of the way from first to last, and the value 2/3 of the way from first to last, we can eliminate about 2/3 of the locations each time. We can call this algorithm **trinary search :**


....
....
....
To search for value X, compare X to this and this
These two comparisons tell us which third of the array contains X

Pseudo-code for the trinary search algorithm is on the next page:

```
trin_search(A,first,last,target):
     # returns index of target in A, if present
     # returns -1 if target is not present in A
     if first > last:
          return -1
     else:
          one_third = first + (last-first)/3
          two_thirds = first + 2*(last-first)/3
          if A[one_third] == target:
               return one_third
          else if A[one_third] > target:
               # search the left-hand third
               return trin_search(A,first,one_third-1,target)
          else if A[two_thirds] == target:
               return two_thirds
          else if A[two_thirds] > target:
               # search the middle third
               return trin_search(A,one_third+1,two_thirds-1,target)
          else:
               # search the right-hand third
               return trin_search(A,two_thirds+1,last,target)
```

Your assignment is to empirically compare the efficiency of these two search algorithms.

Preliminary Work:

The measure of efficiency we will use is the number of times the search function is called. Modify the given algorithms to return the total number of times the function is called during a search, instead of the location of the search value.

For example suppose the array A looks like this:

8	32	34	55	73	80	1432

and the target value is 55

Using `bin_search`, the target value is found immediately so the total number of calls to `bin_search` is 1.

Using `trin_search` , the first call to the function examines 34 and 73, and determines that the target value lies between them. The second call finds the target value, so the total number of calls to `trin_search` is 2

Now suppose the target value is 7

Using `bin_search`, the first call looks at 55, the second call looks at 32, the third call looks at 8, and the fourth call returns because the range of values to examine is empty. So this search requires 4 calls to `bin_search`.

Using `trin_search`, the first call looks at 34, the second call looks at 8, the third call returns because the range of values to examine is empty. So this search requires 3 calls to `trin_search`.

Thus we can see that in some situations `bin_search` is called fewer times than `trin_search`, and in other situations the reverse is true. Of course there may be situations in which they both require the same number of calls.

Experiment 1:

For this range of values of n: n = 1000, n = 2000, n = 4000, n = 8000, n = 16000 complete the following steps. It is possible that due to hardware limitations you may not be able to complete the experiment for the larger data sets. If so, that's ok – just do the largest cases that you can:

> Step 1: Generate an array (or list, if using Python) of n distinct integers. Sort the integers into ascending order. You should be able to write your own sorting function. (An acceptable alternative is to fill the array or list with an increasing sequence of numbers. This can be done by starting with 1 then repeatedly adding a

positive number to the current value.)

Step 2: Use binary search to search the array for each of the values in the array. Record the **total** number of calls to `bin_search` required to conduct the binary searches. Compute the **average** number of calls to `bin_search` required to conduct the binary searches.

Step 3. Use trinary search to search the array for each of the values in the array. Record the **total** number of calls to `trin_search` required to conduct the trinary searches. Compute the **average** number of calls to `trin_search` required to conduct the trinary searches.

Experiment 2:

Repeat Experiment 1, but this time search for n values that are **not present** in the array, but which are uniformly distributed across the range of values in the array. (One easy way to do this is to fill your array with even values, then search for the odd values found by adding 1 to each value in the array.)

Summarize your results by creating tables or graphs for the results of the two algorithms within the two experiments.

Based on the results of your experiments, answer the following questions:

1. Binary search and trinary search both fall into the O(log n) complexity class. Do your experiments show growth in the average number of function calls that is consistent with this? (Reminder: characteristic behaviour of O(log n) complexity is that when n doubles, the amount of work goes up by an approximately constant value.)

2. Does one of the algorithms seem to require fewer function calls than the other (on average) when searching for values that are in the array?

3. Does one seem to require fewer function calls (on average) when searching for values that are not in the array?

4. Can we say there is a value of n such that binary search is better on sets with $\leq$ n elements and trinary search is better on sets with > n elements ... or vice versa ... or does the size of the set not affect the relative efficiency?)

Logistics:

You may complete the programming part of this assignment in Python, Java, C or C++.

You must submit your source code, properly documented according to standards established in CISC-121 and CISC-124. You must also submit a PDF file containing the summary of your experimental results and your answers to the questions. Both files must contain your name and student number, and must contain the following statement: "I confirm that this submission is my own work and is consistent with the Queen's regulations on Academic Integrity."

Combine your files into a .zip file for uploading.

You are required to work individually on this assignment. You may discuss the problem in general terms with others and brainstorm ideas, but you may not share code. This includes letting others read your code or your written conclusions.

The due date for this assignment is 11:59 PM, January 19, 2020. Submission will be through onQ.