TA Lab Weeks 5 & 6 – Searching

The linear and binary search algorithms for searching a sorted array are well known. A linear search will typically look at half the list - $O(n/2)$, and binary search will look at much less - $O(\log\ n)$. Can we do better than that, by looking at the values stored at two positions in the array (e.g., at 1/3 and 2/3) instead of position ½ only, as in binary search?

The goal of this lab is to implement linear, binary and trinary search and empirically compare the efficiency of the three search algorithms. We will also tackle the problem of efficiently operating on sparse matrices by means of care use of symbol tables.

Exercise 1 – Implement the Search Algorithms

Implement linear search, binary search, and trinary search.

The measure of efficiency we will use in the remaining exercises is the number of times the search function is called. Once you know your search algorithms are working, modify the given algorithms to also return the total number of times the function is called during a search, in addition to the location of the search value.

Hint

The pseudocode for a recursive binary search is as follows:

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

Trinary search may be similarly implemented if we compute the values at 1/3 and 2/3 instead of the middle. Then, we check each of the conditions at 1/3 and 2/3 instead of just at the middle of the list.

Exercise 2 – Search for Values in the List

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.

For this range of values of N: {1000, 2000, 4000, 8000, 16000} complete the following steps:

1. Generate a list of N distinct integers. Sort the integers into ascending order. You should be able to write your own sorting function, use a Python built-in sort, or use an implementation from a previous lab.

2. Use linear search to search the array for each of the values in the array. Record the total number of iterations/calls/elapsed time required to conduct the searches. Compute the average number of iterations/calls required to conduct the searches.
3. Use binary search to search the array for each of the values in the array. Record the total number of iterations/calls/elapsed time required to conduct the searches. Compute the average number of iterations/calls required to conduct the searches.
4. Use trinary search to search the array for each of the values in the array. Record the total number of iterations/calls/elapsed time required to conduct the searches. Compute the average number of iterations/calls required to conduct the searches.

Exercise 3 – Search for Values not in the List

Repeat the previous experiments, this time searching for values that are not present in the array, but which are uniformly distributed across the range of values in the array. Summarize your results by creating tables or graphs for the results of the algorithms within the experiments in this and the previous exercise.

Hint

One easy way to create the searching list is to fill your array with even values, then search for the odd values found by adding 1 to each value in the array.

Exercise 4 – Analysis of Search Algorithms

Based on the results of your experiments, try and 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 / elapsed time that is consistent with this?
2. Does one of the algorithms seem to require fewer function calls / elapsed time than the other (on average) when searching for values that are in the array?
3. Does one seem to require fewer function calls / time taken (on average) when searching for values that are not in the array?
4. Can we say there is a value to trinary search over binary search? Consider: is there a value of N such that binary search is better on sets with 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?

Exercise 5 – Sparse Matrices

A simple way to represent a matrix is via a 2D array. However, that might result in a significant waste of space when representing matrices whose dimension is very large, yet they are very sparse (i.e., most elements are zeros). Design and implement a more space-efficient data structure to represent sparse matrices. Implement an efficient API to support matrix sums and multiplications. Experimentally compare the performance of this representation with a 2D array one.

Hint

Represent the matrix more compactly as a symbol table of symbol tables. For a non-zero entry M[x][y], x is key to the first symbol table; y is key to the symbol table (i.e., value) associated to x, and M[x][y] is the value associated to key y.