CSIS385 Algorithms

Lab #4: Brute Force Algorithms

When in doubt, use brute force.

--- Ken Thompson, Turing Award winner and creator of the original Unix operating system

Names: _____

Learning goals:

- to use empirical methods to study the behavior of algorithms
- to review the brute force Sequential Search algorithm and its worst and best case running times
- to gain additional experience with exhaustive search algorithms

1. (65 Points) In class, we looked at the brute-force algorithm for computing convex hulls, and at the program in the BruteForceConvexHull example that computes it for the METAL highway mapping data graphs. Here you get the opportunity to run experiments and empirically measure its running time.
    - Download the project that includes this program and several METAL graphs from Blackboard.
    - (15 Points) Instrument the program so it measures the time taken to compute the convex hull, and so it counts the number of times it computes the checkVal variable in the computation loop. Be careful not to include the time taken to load the graph, or the time taken to print the results. The Java method System.currentTimeMillis() will come in handy here.
    - (15 Points) Modify the code so that when the second command-line parameter given is "timings" (as opposed to the existing "tmg" and "text" options) your program prints a single line containing the name of the graph file, the number of vertices it contains, the time it took to compute the convex hull for that graph, and the number of times checkVal was computed for that graph.
    - (20 Points) Run your program for each of the eleven provided TMG graphs using the new "timings" option. Load this data into a spreadsheet like Excel and create a plot of problem size ($|V|$) vs. the number of checkVal computations. Also compute (in the spreadsheet) and plot $|V|^2$ and $|V|^3$. If using Excel, be sure to sort your data by problem size, and plot using an XY Scatter chart. Attach to your lab submission a printout of your code, and the data and chart from the spreadsheet.
    - (5 Points) Do the numbers from this empirical study make sense given its Big-O (or Big-Theta) complexity? Explain.

    - (5 Points) What does this say about how this algorithm performs relative to its best and worst cases on this specific set of data?

    - (5 Points) Based on the trends you see in your data, about how many computations of checkVal and how long would you expect this program to take using the full "tm-master.tmg" graph (it contains about 290,000 vertices)? Show your work.

2. (4 Points) Another example brute force algorithm is **sequential search**. This algorithm searches for a *key* element k in an array of elements by starting at the beginning of the array and scanning through sequentially element by element until it finds k, or it reaches the end of the array without finding it. Below is pseudocode for sequential search with two parts missing. Study the code and then fill in the boxes with the missing pseudocode.

ALGORITHM SequentialSearch(A[0...n-1], k)
// Input: An array A of n elements and a search key k
// Output: The index of the first occurrence of k in A, or -1 if k is not in A.

i ← 0
while i < n and [ ] **do**
i = i + 1
if i < n return i
else return [ ]

3. (4 Points) For an n element array, exactly how many <u>key</u> comparisons does the algorithm make in the worst case? (A key comparison is any comparison made between the key k and an element of the array. The number of key comparisons often determines the asymptotic running time of searching algorithms, which is why we count them.)

4. (4 Points) The best case efficiency of an algorithm is not nearly as important as that of the worst-case efficiency. But it is not completely useless either. For example, some sorting algorithms perform best on nearly sorted data, which is a common situation in practice. For an n element array, exactly how many <u>key</u> comparisons does sequential search make in the best case?

5. (6 Points) A simple trick is often used in implementing sequential search: if the key k is added to the end of the array (assume there is space for it), then the search will find the key there (for an unsuccessful search), if not sooner (for a successful search). The advantage of this is that we can eliminate the check `i < n` in each iteration of the while loop. Write pseudocode for this version of sequential search below.

ALGORITHM SequentialSearch(A[0...n], k)
// Input: An array A of n elements (with an extra, unused entry A[n] at the end) and a search key k
// Output: The index of the first occurrence of k in A, or -1 if k is not in A.

6. (4 Points) This problem is from the Nintendo DS game Professor Layton's Diabolical Box, which is full of puzzles/problems of the type we encounter regularly in algorithms: In the figure below, collect all of the mushrooms in the forest as you pass through. Each circular clearing on the map contains mushrooms. You don't want to spend too long in this creepy forest though, so find a route through that visits each clearing only once. Stay on the roadways.

7) (5 Points) Suppose you were to solve the mushroom problem above using a *very* exhaustive search. I.e., for every possible ordering of the 18 mushrooms (not counting the start and goal mushrooms), check if the roadways exist that allow them to be visited in that order. In the worst case, how many different orderings would need to be checked?

8) (4 Points) If your computer can check 1 billion orderings per second, how long would this exhaustive search approach take in the worst case?

9) (4 Points) What well-known problem in computer science is illustrated in this puzzle?