CS5050 Advanced Algorithms

Spring Semester, 2021

Assignment 1: Algorithm Analysis

Due Date: 11:59 p.m., Thursday, Feb. 4, 2021

Note: Please turn in your homework on Canvas. You may either type or write your solutions on paper and then scan it. The Canvas submission will be automatically closed at the due time on the due day (this applies to all assignments in this semester), so please make your submission on time. **Note:** If not specified, the base of log is 2. This applies to all assignments in this semester.

1. **(10 points)** This exercise is to convince you that exponential time algorithms should be avoided if possible.

   Suppose we have an algorithm A whose running time is $O(2^n)$. For simplicity, we assume the algorithm A needs 2^n instructions to finish, for any input size of n (e.g., if $n = 5$, A will finish after $2^5 = 32$ instructions).

   According to Wikipedia, as of November 2020, the fastest supercomputer in the world is the Japanese Fugaku (located in in Kobe, Japan) and can perform about 4.0×10^{17} instructions per second.

   Suppose we run the algorithm A on Fugaku. Answer the following questions.

   (a) For the input size $n = 100$ (which is a relative small input size), how much time does Fugaku need to finish the algorithm? Give the time in terms of **centuries** (you only need to give an approximate answer).

   Around 1,005 centuries.

   (b) For the input size $n = 1000$, how much time does Fugaku need to finish the algorithm? Give the time in terms of **centuries** (you only need to give an approximate answer).

   Around 8.49 * 10^273 centuries.

   Note: You may assume that a year always has exactly 365 days.

2. **(20 points)** Order the following list of functions in asymptotically increasing order (i.e., from small to large).

$$\begin{array}{cccccc} \log n & n! & 2^{500} & 2^n & \log(\log n)^2 & 2^{\log n} \\ \log^3 n & n\log n & \log_4 n & n^3 & \sqrt{n} & n^2\log^5 n \end{array}$$

$2^{500}, \log n, log_4 n, \log(\log n)^2, log^3 n, \sqrt{n}, n\log n, n^2 log^5 n, n^3, 2^{\log n}, 2^n, n!$

3. **(30 points)** For each of the following pairs of functions, indicate whether it is one of the three cases: $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, or $f(n) = \Theta(g(n))$. For each pair, you only need to give your answer and the proof is not required.

   (a) $f(n) = 7\log n$ and $g(n) = \log n^3 + 56$.

   f(n)=logn < g(n)= logn^3 so f(n) = $O(g(n))$

   (b) $f(n) = n^2 + n\log^3 n$ and $g(n) = 6n^3 + \log^2 n$.

   f(n)= n^2 < g(n)= n^3 so f(n) = $O(g(n))$

   (c) $f(n) = 5^n$ and $g(n) = n^2 2^n$.

   f(n) = 5^n > g(n) = 2^n so f(n) = $\Omega(g(n))$

   (d) $f(n) = n\log^2 n$ and $g(n) = \frac{n^2}{\log^3 n}$.

   f(n) = n < g(n) = n^2 so f(n) = $O(g(n))$

   (e) $f(n) = \sqrt{n}\log n$ and $g(n) = \log^8 n + 25$.

   f(n) = sqrt(n) > g(n) = log^8 n so f(n) = $\Omega(g(n))$

   (f) $f(n) = n\log n + 6n$ and $g(n) = n\log_3 n - 8n$

   f(n) = n = g(n) so f(n) = $\Theta(g(n))$

4. **(20 points)** This is a "warm-up" exercise on algorithm **design** and **analysis**.

   The *knapsack problem* is defined as follows: Given as input a knapsack of size K and n items whose sizes are $k_1,k_2,...,k_n$, where K and $k_1,k_2,...,k_n$ are all **positive real numbers**, the problem is to find a full "packing" of the knapsack (i.e., choose a subset of the n items such that the total sum of the sizes of the items in the chosen subset is *exactly* equal to K).

   It is well known that the knapsack problem is NP-complete, which implies that it is very likely that efficient algorithms (i.e., those with a polynomial running time) for this problem do not exist. Thus, people tend to look for good **approximation algorithms** for solving this problem. In this exercise, we relax the constraint of the knapsack problem as follows.

   We still seek a packing of the knapsack, but we need not look for a "full" packing of the knapsack; instead, we look for a packing of the knapsack (i.e., a subset of the n input items) such that the total sum of the sizes of the items in the chosen subset is *at least* $K/2$ (but no more than K). This is called a *factor of* 2 *approximate solution* for the knapsack problem. To simplify the problem, we assume that a factor of 2 approximate solution for the knapsack problem always exists, i.e, there always exists a subset of items whose total size is at least $K/2$ and at most K.

   For example, if the sizes of the n items are {9,24,14,5,8,17} and $K = 20$, then {9,5} is a factor of 2 approximate solution. Note that such a solution may not be unique. For example, {14} is also a solution.

Design a **polynomial time** algorithm for computing a factor of 2 approximate solution, and analyze the running time of your algorithm (using the big-O notation). Note that although there may be multiple solutions, your algorithm only needs to find one solution.

If your algorithm runs in $O(n)$ time and is correct, then you will get **5 bonus points**.

Note: I would like to emphasize the following, which applies to the algorithm design questions in all assignments in this semester.

1. **Algorithm Description** You are required to clearly describe the main idea of your algorithm.
2. **Pseudocode** The pseudocode is optional. However, pseudocode is very helpful for explaining algorithms. Therefore, you are strongly encouraged to provide pseudocode for your algorithms. Also, although this is quite subjective, if your solution is not correct, then usually you will receive slightly more partial points if pseudocode is provided.
3. **Correctness** You also need to explain why your algorithm is correct. For example, for this knapsack problem you need to explain why the solution produced by your algorithm is a factor of 2 approximate solution.
4. **Time Analysis** Please make sure that you analyze the running time of your algorithm.

1. The main idea of my algorithm is a linear sort that checks the first value, k_1, to see if it is greater than K/2 and less than K and if so return that value. It then checks if k_1 is less than K/2 and if so, adds the value to the variable value and then check the variable value to see if it is between or equal to K/2 and K. If k_1 is greater than K the algorithm will move on to the next item on the list. This will repeat through a for loop until a solution is found.
2. {22 ,4 ,21 , 5, }

```
float value = 0
for (i = 0; i < n; i++)
{
if ($k_i \geq \frac{K}{2}$ and $k_i \leq K$)
{ value = $k_i$; return value }
if ($k_i < \frac{K}{2}$)
{ value += $k_i$ ;if (value $\geq \frac{K}{2}$ and value $\leq K$){return value} }
}
```

3. This algorithm specifically searches for a value that is at least K/2 or at the most K, either by finding a value already in that range or through adding up values until that range is reached. This is correct because there always exists a subset of items whose total size is at least $K/2$ and at most K, so this algorithm is a factor of 2 approximate solution.

4. The running time for this algorithm is O(n) because it is linear search. The worst case would be the for loop running through every single item, which would give it a running time of O(n).

Total Points: 80 (not including the five bonus points)