CS5050 Advanced Algorithms

Philip Nelson: worked with Raul Ramirez, Ammon Hepworth

Assignment 1: Algorithm Analysis

Due Date: 3:00 p.m., Thursday, Jan. 25, 2018 (**at the beginning of CS5050 class**)

1. **(10 points)** This exercise is to convince you that exponential time algorithms are not quite useful.

   (a) For the input size $n = 100$:

   $$\frac{2^{100}}{1.25 \cdot 10^{17} \cdot 3600 \cdot 24 \cdot 365 \cdot 100} = 3215.75 \text{ centuries}$$

   (b) For the input size $n = 1000$:

   $$\frac{2^{1000}}{1.25 \cdot 10^{17} \cdot 3600 \cdot 24 \cdot 365 \cdot 100} = 2.72 \cdot 10^{274} \text{ centuries}$$

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

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

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

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

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 $g(n) = \log n^3 + 56$. $f(n) = \Theta(g(n))$

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

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

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

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

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

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

   1. **Algorithm Description** The *fill_knapsack* algorithm can fill a knapsack of size K from an array A to at least $\frac{K}{2}$ in a single linear scan. During the linear scan, check each element $a_i : \frac{K}{2} \leq a_i \leq K$. If an a_i is found, immediately return it as a single element solution. If the element is not a single element solution, check if it is an $a_j : a_j < \frac{K}{2}$. Add elements a_j to the knapsack and check if it is at least $\frac{K}{2}$ full. If the knapsack is at least $\frac{K}{2}$ full, return it.

   2. **Pseudocode** fill_knapsack

```
array fill_knapsack(array A, K)
{
  array knap;
  int sum = 0;

  for (i = 0; A.size(); ++i) // single linear scan
  {
    if (K / 2 <= A[i] && A[i] <= K) // check for one element solution
      return {A[i]};

    if (A[i] < K / 2) // all elements < K/2
    {
      sum += A[i];
      knap.push_back(A[i]);

      if (sum >= K/2) // return knapsack when it is more than K/2 full
        return knap;
} } }
```

   3. **Correctness** The first way to fill the knapsack is obviously correct. The algorithm will put any single element $a_i : \frac{K}{2} \leq a_i \leq K$ in the knapsack. If no such single element exists, the knapsack will be filled with elements $a_j : a_j < \frac{K}{2}$. It is impossibe to overfill the knapsack using this method. If two elements a_1 and a_2 are put in the knapsack such that $a_1, a_2 < \frac{K}{2}$, the elements will be $\frac{k-1}{2}$ at the largest. Thus

$$a_1 + a_2 = \frac{K-1}{2} + \frac{K-1}{2} = \frac{2K-2}{2} = K - 1$$

   Therefore any $a_1 + a_2$, where a_1 and a_2 could be combinations of multiple a_j, will never be more than $K - 1$ therefore a solution will always be found.

   4. **Time Analysis** Please make sure that you analyze the running time of your algorithm. The algorithm *fill_knapsack* solves the knapsack problem, factor two approximation, in order $O(n)$. The order is $O(n)$ because there is one for loop which does a linear scan over the array A containing n elements. During the linear scan, a constant amount of work is done; this only affects the order by a constant ammount and therefore can be ignored. Thus, *fill_knapsack* is $O(n)$.

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