Brigham Michaelis
A00364825
September 12th 2017

Assignment #1

Problem 1.
This exercise is to convince you that exponential time algorithms are not quite useful. Suppose we have an algorithm A whose running time is O(2^n). For simplicity, we assume that 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 June 2017, the fastest supercomputer in the world is "Sunway TaihuLight" (which is located in Wuxi, China) can perform roughly 10^17 instructions per second. Suppose we run the algorithm A on Sunway TaihuLight. Answer the following questions.

a.) For the input size n = 100 (which is a relatively small input), how much time does Sunway TaihuLight need to finish the algorithm? Give the time in terms of centuries.

We would need to run 2^{100} instructions to finish.
We need to provide runtime in centuries, for 10^{17} instructions per second
There are 3.154 e^{9} seconds in a century

The SunwayTaihuLight can do
$3.154e^{26}$ instructions per century.

So…
2^{100} divided by $3.154e^{26}$ which gives us about 4019 centuries.

b.) For the input size n = 1000, how much time does Sunway TaihuLight need to finish the algorithm? Give the time in terms of centuries.

Following the same method from above;
2^{1000} divided by $3.154e^{26}$ we get 3.397301e+274 centuries.

Problem 2.
Suppose you have algorithms with the five running times given below. (Assume these are the exact running times.) How much slower does each of these algorithms become when you double the input size (i.e., n becomes 2n)?

Double the input size how much does it grow?

(a) n^2 => 4
(b) n^3 => 8
(c) $100n^2$ => 4
(d) nlog(n) => ((2/logn) + (2))

The answer to the following question was discovered while working with Josh Dawson.
(e) 2^n => 2^n

Problem 3.

Note* Some of these answers were discovered while working with Josh Dawson.

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.

For each pair indicate the relationship

(a) $f(n) = 100n + \log n$ and $g(n) = 6n + \log^2 n$. => $f(n) = \Theta(g(n))$
(b) $f(n) = 20 \log n + 4$ and $g(n) = \log n^2 - 100$. => $f(n) = \Theta(g(n))$
(c) $f(n) = n^2/(\log n)$ and $g(n) = n\log^2 n$. => $f(n) = \Omega(g(n))$
(d) $f(n) = \mathrm{sqrt}(n)$ and $g(n) = \log^5 n$. => $f(n) = \Omega(g(n))$
(e) $f(n) = n2^n$ and $g(n) = 3^n$ => $f(n) = O(g(n))$
(f) $f(n) = 4n \log n$ and $g(n) = n\log_3 n$ => $f(n) = \Theta(g(n))$

4. 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 approximation solution for the knapsack problem.
To simplify the problem, we assume that a factor of 2 approximation 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 approximation solution. Note that such a solution may not be unique. For example, {9, 8} is also a solution.

Design a polynomial time algorithm for computing a factor of 2 approximation solution, and analyze the running time of your algorithm (in the big-O notation). (20 points)

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 of this course.

1. **Algorithm Description** You are required to clearly describe the main idea of your algorithm.

2. **Pseudocode** The pseudocode is optional. However, you may also give the pseudo-code if you feel it is helpful for you to explain your algorithm. (The reason I want to see the algorithm description instead of only the code or pseudo-code is that it would be difficult to understand another person’s code without any explanation.)

3. **Correctness** You also need to briefly explain why your algorithm is correct, i.e., why your algorithm can produce a factor of 2 approximation solution.

4. **Time Analysis** Please make sure that you analyze the running time of your algorithm.

1. Algorithm Description

Here is the outline of my algorithm which will run in O(n) time.

Assumptions:
1. We know how big the knapsack is. K is given and it is a constant
2. The solution is obtained as soon as the knapsack is filled to at least K/2. After that happens the algorithm terminates.
3. There exists a solution.

D
We are going to traverse the set just once. And at each item we will ask;

1. Is this item greater than K?
   If it is this item has no use to us and we discard and move to the next thing in line. ==> continue

If that first condition doesn't happen then we move to step 2.

2. Is this item greater than or equal to K/2 and less than or equal to K?
   If it is we are done and the knapsack is filled to at least K/2. ==> terminate.

If that second condition doesn't happen then we move to step 3.

3. Because we are at this step we know that every item we are looking at here is smaller than K/2.
   We are going to take every item we encounter in this step and place them in a "possible" knapsack. ==> continue.

3a. If our "possible" knapsack has a size greater than or equal to K/2. We are finished and the knapsack is filled to at least K/2. ==> terminate.

2. **Pseudocode**

Not really needed.

3. **Correctness**

This is correct for two reasons.

1. If an item is greater than or equal to K/2 and less than or equal to K, this algorithm will find it and terminate once it does.

2. If reason 1 is not satisfied then it means that a solution is to be found from some combination of items that are less than K/2. The algorithm described above will collect all the items of this size that it encounters and place them in a "possible" knapsack. Once this knapsack reaches size of at least K/2. We are done.

2a. This works because you can never add an item to the running sum that will tip it over K. Because all the values are smaller than K/2.

4. **Time Analysis**

As was mentioned before we are traversing the set just once and we are performing multiple operations at every item. But we are only traversing once, So O(n).