CS 3510 | **Instructor: Prateek Bhakta**
Georgia Tech, Spring 2015 | Student: YOUR NAME HERE

Homework 1

This homework is due by the ***start of class on Friday, January 22nd***. You can either submit the homework via the course page on T-Square or hand it in at the beginning of class. Since this is a short homework, this is an ideal time to learn to use Latex.

Homework Policies:

- Your work will be graded on correctness and clarity. Write complete and precise answers.
- You may collaborate with up to *three* other classmates on this problem set. However, you must *write your own solutions* in your own words and *list your collaborators.*
- You may portions of our two textbooks that we have covered. The purpose of these homeworks is NOT only to teach you the algorithms that we cover, but also to teach you how to problem-solve. Therefore using *ANY* outside resources like the internet, previous students etc. is *not* allowed!
- **Optional** questions are for you to practice and learn the basics before you advance. They may ask you to follow the steps of an algorithm covered in class, or review material from an earlier class. Solutions to these problems will not be graded, but you must be are able to do them, and you are responsible for material that is addressed by these optional questions.
- **Basic** questions *must be solved and written up alone*; you may not collaborate with others. These will help you build the framework needed to solve the harder questions. Please do these first, before meeting in groups, for both you and your group's benefit.
- **Group** questions may be solved in collaboration with up to ***three*** other classmates. However, you must still *write up your own solutions* alone and in your own words. These will form the bulk of your homework questions.
- **Bonus** questions are like group questions, but they will be more challenging and will be graded more rigorously. Bonus points are tallied separately from your normal points, and will be applied to your grade *after* the curve. Bonus points are worth more than normal points, and can have a significant positive affect on your grade. Most importantly, bonus problems are fun!

1. (Optional) Please write a few sentences about your math background, and what you are hoping to get out this class.

2. (15 points) (Basic) For each pair of functions below, decide which of the following is true: (1) $f(n) = o(g(n))$, (2) $f(n) = \Theta(g(n))$, or (3) $g(n) = o(f(n))$. Justify briefly your choices.

   (a) (Optional) $f(n) = 3^{n+2}$, $g(n) = 3^n$.

   Solution: (2) $f(n) = \Theta(g(n))$. Since $f(n) = 9g(n)$, $f(n)/g(n) = 9$, which is a constant.

   (b) (Optional) $f(n) = 5^{3n+2}$, $g(n) = 5^n$.

   Solution: (3) $g(n) = o(f(n))$. Since $f(n)/g(n) = 5^{2n+2}$, which goes to ∞.

(c) (5 points) $f(n) = n^{5000} + 2^n$, $g(n) = 3^n$.

Solution: (1) $f(n) = o(g(n))$. In this case, $f(n)/g(n) = \frac{n^{5000}}{3^n} + \frac{2^n}{3^n}$. The first term approaches 0 as any linear exponential grows faster than a polynomial (you could also see this by repeated L'hopital's rule). The second term aproaches 0 as it is an exponential with $|2/3| < 1$. Thus the limit goes to 0.

(d) (5 points) $f(n) = \ln(n)$, $g(n) = \log_3(2n)$.

Solution: (2) $f(n) = \Theta(g(n))$. Using basic properties of logarithms, we can show that $\forall n, g(n) = \ln(2n)/\ln(3) = \frac{\ln(2)+\ln(n)}{\ln(3)}$. Thus $\frac{f(n)}{g(n)} = \frac{\ln 3 \cdot \ln n}{\ln n + \ln 2} = \ln 3 \frac{1}{1+\frac{\ln 2}{\ln n}} \to \ln 3$. Thus $f(n) = \Theta(g(n))$.

(e) (5 points) $f(n) = n^3$, $g(n) = 9^{3\log_5 n}$
(Hint: First convert $g(n)$ into a more useful form).

Solution: Using properties of logs, we see that $g(n) = (5^{\log_5 9})^{3\log_5 n} = (5^{\log_5 9 \cdot 3\log_5 n} = (5^{\log_5 n \cdot 3\log_5 9} = n^{3\log_5 9}$. As $\log_5(9) > 1$, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{1}{n^{3(\log_5(9)-1)}} \to 0$, thus $f(n) = o(g(n))$.

3. (45 points) (Group) Describe a reasonable greedy algorithm for each of the following problems. Then find an input for each problem to show that the greedy algorithm is NOT optimal.

(a) (15 points) InterstellarTravel: Say there are n planets in a line, you are on planet 1 and are travelling to plant n. On each planet, you can find flights only to the other planets that have a larger index, but the cost of each flight is completely arbitrary.

Find the cheapest way to get to planet n.

Solution: A reasonable greedy solution would be to chose a flight that takes you the farthest for the least amount of price at each point along the way. Assume we have access to cost to travel to planet by indexing .cost.

```
function InterstellarTravel(Array $Planets[P_1 \dots P_n]$)
    $currentPlanet = P_1$
    $totalCost = 0$
    $visitedPlanets = [P_1]$
    while $currentPlanet \neq P_n$ do
        $accessiblePlanets = [P_i \dots P_n]$ where $Pi > currentplanet$
        Sort $accessiblePlanets$ by increasing (index-of-planet/cost-of-flight-to-planet)
        $totalCost += accessiblePlanets[end].cost$
        $visitedPlanets.add(accessiblePlanets[end])$
        $currentPlanet = accessiblePlanets[end]$
    return $visitedPlanets, totalCost$
```

Input for which the above algorithm won't work :
Suppose there are 4 Planets, P_1, P_2, P_3 and P_4. Let the cost to travel to P_2, P_3 and P_4 from P_1 be 2,1, and 5 respectively . Let the cost to travel to P_3, P_4 from P_2 be 2 and 1

respectively. Let the cost to travel to P_4 from P_3 be 5.
Algorithm's output: The algorithm would chose to travel to P_3 first as $3/1 > 2/2 > 4/5$. It would then travel to P_4 from P_3, for a total cost of $1 + 5 = 6$.
Optimal Output: The path would the least cost would involve travelling to P_2 first and then travelling to P_4, for a total cost of $2+1 = 3$.

(b) (15 points) FactoryAssignment: You are deciding how to allocate the work of n tasks to m machines. Your input is a set of times $t_1, \dots t_n$ that each task takes, and a number m of machines.

Your goal is to find an assignment of tasks to machines that minimizes the *throughput*, which is the longest total time that *any* machine takes.

Solution: A greedy solution would be to keep track of the time of tasks assigned to each machine and assign the longest tasks to machines which have shortest tasks assigned. Each machine would keep track of the array of tasks assigned to it and the total time to complete these assigned tasks.

```
function FactoryAssignment(Array $times[t_1 \dots t_n]$, int $numMachines$)
    $machinesList$ tracks all assigned tasks for each machine.
    for all $i$, Set $machinesList[i].timeAssigned$ to 0
    Sort times in decreasing order of time.
    while $times \neq \emptyset$ do
        Sort $machinesList$ in increasing order of $machinesList[i].timeAssigned$
        $machinesList[0].tasksAssigned.add(times[0].originalTaskIndex)$
        $machinesList[0].timeAssigned += times[0]$
        $times.remove(times[0])$
    return $machinesList$
```

Input for which the above algorithm won't work : Assume there are 4 tasks and 2 machines. Let t_1 take 5 unit of time, t_2 take 4 units of time, t_3 take 3 units of time, t_4 take 3 units of time and t_5 take 3 units of time
Algorithm's output: The algorithm would assign the first task to Machine1 (timeAssigned for M1 =5) and the second task to Machine 2 (timeAssigned for M2 = 4). After this, it would assign the third task to Machine2, the fourth task to Machine1 and the fifth task to Machine2, Resulting in the following assignment: M1 = t_1, t_4 while M2= t_2, t_3, t_5. This results in M1 taking 8 units of time and M2 taking 10 units of time, resulting in a throughput of 10.
Optimal Output: A possible optimal assignment of tasks would be M1 = t_3, t_4, t_5 and M2 = $t_1 andt_2$, resulting in M1 taking 9 units of time and M2 taking 9 units of time, resulting in a throughput of 9.

(c) (15 points) JobHiring: You need to hire workers for your company. Your company needs to do a set of n tasks $T = \{t_1, t_2, \dots t_n\}$, and you have m available workers, each of which is able to do some subset of the tasks, $W_1, W_2, \dots W_m$ where each $W_i \subset T$.

Your goal is to hire the fewest number of workers so that all the tasks can be completed.

Solution: Keep a list of tasks that need to get done. Each time a worker is chosen, remove all the tasks that the worker can do from the list of tasks that need to get done. Also remove the tasks that the chosen worker can do from the tasks that other workers (not chosen as of yet) can do. At each point, pick a worker that can do the maximum number of remaining tasks.

```
function JobHiring(Array $T[t_1 \dots t_n]$ , Array $workers[W_1 \dots W_m]$ )
    $hiredWorkers = []$
    while $T \neq \emptyset$ do
        Sort $workers$ by increasing number of tasks they can perform
        $hiredWorkers$.add($workers[end].index$)
        $workers.remove(workers[end])$
        $tasksOfChosenWorker = workers[end]$
        for each worker in workers do
            remove from worker tasks also in $tasksOfChosenWorker$
        for each task in T do
            if $task$ in $tasksOfChosenWorker$ then
                remove $task$ from T
    return $hiredWorkers$
```

Input for which the above algorithm won't work :
Suppose there are 5 tasks to be completed: t_1, t_2, t_3, t_4 and t_5. Let there be 3 Workers who can complete tasks W_1, W_2, and W_3 respectively. Let $W_1 = [t_1, t_2, t_3]$. Let $W_2 = [t_1, t_2, t_4]$ and let $W_3 = [t_3, t_5]$
Algorithm's output: The algorithm would chose worker1 who can complete W_1 tasks as 5 tasks need to be completed in total and this particular worker can perform 4 of them. However, the algorithm would then have to chose both worker2 (as this is the only one who can complete t_4) and worker3 (as this is the only worker who can complete t_5).
Optimal Output: The optimal choice would be to directly select worker2 and worker3. Together, they can complete all 5 tasks.

4. (Optional) Consider the ActivityScheduling problem from lecture. We are given a list of activities, each with a (start, end) timestamp pair, where $0 \leq start \leq end$. We seek to find the maximum number of non-overlapping activities.

   Prove that the following greedy algorithm discussed in class, which always takes the available activity that will end first, is optimal.

```
function Schedule(Array $Activities[A_1, \dots A_n]$)
    Sort $Activities$ by increasing end times
    $curend = 0$
    $totevents = 0$
    for $i \in [1, \dots, n]$ do
        if $A_i.start \geq curend$ then                ▷ If TRUE, we are "choosing" this activity
            $curend = A.end$
            $totevents++$
    return TotEvents
```

Solution: We'll prove this by induction on the number of activities.

- For the Base case, say there was only one activity. Then our algorithm would choose that one activity on the one and only pass through the for loop. This is clearly optimal.
- For the Inductive step, say that we have $n > 1$ activities total, and that this algorithm is optimal for up to $[1, \ldots, n-1]$ activities.

  We note that our algorithm will begin by choosing A_1, the activity with the *earliest* end time.

  - First, we argue that A_1 MUST be in an optimum solution. Consider an optimum solution that does not include A_1, but whose first element is some A_k. Since A_1 ends before A_k, we could replace A_k with A_1 without interfering with any of the activies that come after A_k. Therefore, we may as well choose A_1 as the first element!
  - Since A_1 is in an optimum solution, our solution and this optimal solution agree on the first element. The rest of this optimal solution is an maximal non-overlapping subset of the remaining $< n$ activities that start after $A.end$. This is exactly what our algorithm computes, and by the induction hypothesis, our algorithm will find that optimally.

Note: The induction could also have worked by considering the last element A_n of the list.

5. (40 points) (Group) Consider the ChangeMaker algorithm from lecture. PROVE that if the denominations are all the powers of 2, [1,2,4,8,16,etc.], then the greedy algorithm will return the optimal solution.

   Note: There are a few different ways to prove this, including proof by induction.

Solution:

We provide two potential solutions, though others are out there.

Proof 1: Direct

Lemma 5.1: The optimal solution will use at most one of each denomination

PROOF: We will prove by contradiction. Suppose the optimal solution uses at least 2 coins of some denomination 2^i. Remove two of these coins, and add a coin $2^{i+1} = 2^i + 2^i$ to get a valid set of change with one fewer coin. This is a contradition, since the original solution was supposed to be optimal. ∎

Lemma 5.2: The greedy algorithm will produce a solution that uses at most one of each denomination

PROOF: We will again prove by contradiction. Suppose the greedy algorithm chooses a coin of size 2^i at least twice. When the coin 2^i was first chosen, the amount of change still needed must have been at least $2(2^i) = 2^{i+1}$. This is a contradiction, as the greedy algorithm would have chosen a coin of greater denomination. ■

By Lemma 5.1 and 5.2, for any denomination 2^i, the optimal solution and the greedy algorithm will have 1 or 0 of that coin. We can therefore view both as a binary representation of the initial value. There is only one way to represent any given number in binary, which implies that the optimal solution and the greedy algorithm must be equivalent.

Proof 2: Induction

We induct on the value n that we are making change for.

Base case: If $n = 1$, we will output 1 coin, which must clearly be optimal, as we could not make change with 0 coins.

Inductive step: Assume the greedy algorithm behaves correctly for all inputs less than n.

Consider the largest coin in an optimal solution for n, where $2^i \leq n < 2^{i+1}$. We know that the greedy algorithm will choose 2^i as it's first step.

Suppose for the sake of cantradction that the optimal solution does not contain coin 2^i. Since 2^{i+1} is larger than n, that means that the optimal solution must only use coins of size $\leq 2^{i-1}$. However, even if you chose one of every possible coin, $1, 2, 4, \ldots, 2^{i-1}$, the sum value would be $1 + 2 + 4 \ldots 2^{i-1} = 2^i - 1 < n$. Thus this solution would have to use some coins more than once. By Lemma 5.1, this is impossible. Hence, the optimal solution must contain the coin 2^i.

Thus the greedy and optimal solution agree on the largest coin. The remaining coins in both solutions add to $n - 2^i$. By the inductive hypothesis, the greedy algorithm will find change for the remaining $n - 2^i$ optimally, and will therefore coincide with the rest of the optimal solution. Therefore, the greedy algorithm will produce the optimal solution for n.