CPSC 320 2018W1, Midterm 2 Sample Solution

1 Greed is good

1. **[1.5 marks]** Which one(s) of the following algorithms, listed here in alphabetical order, is/are greedy? Circle all that apply.
   - [✓ Dijkstra's shortest paths algorithm]
   - ○ Gale Shapley's stable matching algorithm
   - [✓ Kruskal's minimum spanning tree algorithm]
   - ○ The Power algorithm from quiz 4
   - [✓ The Unweighted Interval Scheduling algorithm]
   - ○ The Weighted Interval Scheduling algorithm

2. **[2 marks]** A server has n customers waiting to be served. The service time required by each customer is known in advance: it is t_i minutes for customer i. So if, for instance, the customers are served in order of increasing i, then customer number i has to wait $\sum_{j=1}^{i} t_j$ minutes. We wish to minimize the total waiting time

$$T = \sum_{i=1}^{n} (\text{time spent waiting by customer } i).$$

   Describe succintly an efficient algorithm for computing the optimal order in which to process the customers.

   Solution: Order customers from fastest service time to slowest.

3. **[2.5 marks]** The following is the outline of a proof of correctness for the correct solution to question 1.2. Please fill in the holes we left in the proof outline with the appropriate information.

   First, we prove that there is an optimal solution that

   $\boxed{\text{has the customer with fastest serving time first}}$

   Then we use

   $\boxed{\text{induction}}$

   and the previous fact to prove that our greedy solution is at

   $\boxed{\text{at least as good as any optimal}}$

   solution.

2 Yet another spanning algorithm

Let $G = (V, E)$ denote a connected, undirected graph with $n \geq 2$ nodes and m weighted edges. Throughout this problem, *assume that all edges of E have distinct weights.* Let $\text{wt}(e)$ denote the weight of edge e of G. The following algorithm appeared on a recent quiz:

```
1.     Algorithm Spanning($G = (V, E)$, wt())
2.
3.         Let $G' = (V, E')$ where $E' = \emptyset$
4.         While $G'$ is not connected
5.             $E$-new $= \emptyset$
6.             For each connected component $C$ of $G' = (V, E')$
7.                 Find an edge $e = (u, v) \in E$ of minimum weight wt($e$) that
8.                 connects a node $u$ in $C$ to a node $v$ that is not in $C$
9.                 $E$-new $= E$-new $\cup \{e\}$
10.            $E' = E' \cup E$-new
11.        Return $G'$
```

1. **[2 marks]** As covered in the quiz, in the worst case, $\Theta(\log n)$ iterations of the While loop will be executed. Describe an input graph with n nodes on which this worst case behaviour can happen, where furthermore the most costly edge of E is added to E' in the *first* iteration of the While loop.

   Solution: Worst case behavior happens on a graph that forms a path. To have most costly edge of E added in first iteration: suppose the highest weight edge is from one of the end nodes of the path to the next node. Since it's the only edge from that component, the algorithm will choose it.

2. **[4 marks]** Explain why the algorithm always returns a tree on all inputs $G = (V, E)$, given our assumption that all edges of E have different weights.

   Solution: Since the algorithm continues until G' is connected, G' contains a tree. Suppose to the contrary G' is not a tree; then it must contain a cycle. Let i be the first iteration in which a cycle is introduced. Let the components connected by this cycle be $C_1, C_2, \ldots, C_k$, where we choose this ordering so that C_i chooses an edge to C_{i+1} and C_k chooses an edge to C_1. Then the edge e_1 must have weight greater than the edge e_2, since C_2 did not choose edge e_1. Generalizing, we have that:

$$wt(e_1) > wt(e_2) > \ldots > wt(e_k) < wt(e_1),$$

   a contradiction since $wt(e_1)$ cannot be greater than itself.

3 Recursive multiplication recurrence

[5 marks] The following algorithm, due to Karatsuba, multiplies two n-bit unsigned integers $x = x_n x_{n-1} \dots x_1$ and $y = y_n y_{n-1} \dots y_1$. For $n > 1$ the algorithm is based on the following observation. Let $x_H = x_n x_{n-1} \dots x_{\lfloor n/2 \rfloor + 1}$ and $x_L = x_{\lfloor n/2 \rfloor} \dots x_1$. For example, if $x = 11101$ then $x_H = 111$ and $x_L = 01$. Define y_H and y_L similarly, by breaking y in two. Then the product of x and y can be written as

$$\begin{aligned} xy &= x_H y_H 2^n + (x_H y_L + x_L y_H) 2^{n/2} + x_L y_L && (1) \\ &= z_2 2^n + z_1 2^{n/2} + z_0, && (2) \end{aligned}$$

where $z_0 = x_L y_L, z_1 = x_H y_L + x_L y_H$, and $z_2 = x_H y_H$. Furthermore, in equation (2), the multiplications by 2^n and $2^{n/2}$ can be done using bit-shift operations in $\Theta(n)$ time, and also the additions can be done in $\Theta(n)$ time.

```
Algorithm Multiply($x = x_1 x_2 \dots x_n$, $y = y_1 y_2 \dots y_n$)
    // $x$ and $y$ are unsigned $n$-bit numbers, where $n \geq 1$
    If $n == 1$ then // Base case
        If ($x_1 == 1$) and ($y_1 == 1$) then
            Return 1
        Else
            Return 0
    Else
        $x_H = x_n x_{n-1} \dots x_{\lfloor n/2 \rfloor + 1}$
        $x_L = x_{\lfloor n/2 \rfloor} \dots x_1$
        $y_H = y_n y_{n-1} \dots y_{\lfloor n/2 \rfloor + 1}$
        $y_L = y_{\lfloor n/2 \rfloor} \dots y_1$
        $z_0 = $ Multiply($x_L, y_L$)
        $z_2 = $ Multiply($x_H, y_H$)
        $z_1 = $ Multiply($x_L + x_H, y_L + y_H$) $- z_0 - z_2$ // This quantity is $x_H y_L + x_L y_H$
        Return $z_2 2^n + z_1 2^{n/2} + z_0$ // This step can be done in $\Theta(n)$ time
```

Let $M(n)$ be the running time of this algorithm on two n-bit integers. Give a recurrence that provides a good upper bound for $M(n)$. You can ignore floors and ceilings.

$$M(n) \leq \begin{cases} \boxed{c} & \text{if } n = 1 \\ \boxed{3M(n/2) + cn} & \text{if } n \geq 2 \end{cases}$$

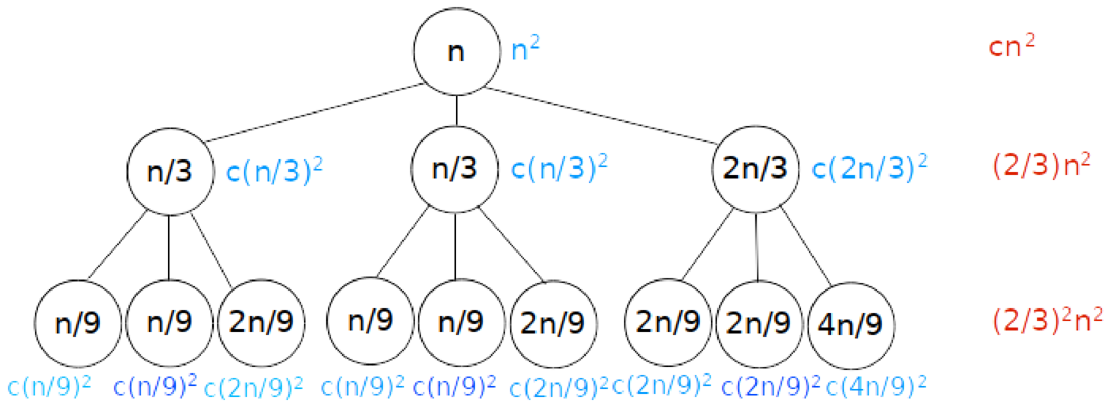
4 Recursive running times

[8 marks] While trying to come up with questions to ask on a midterm, Anne and Patrice discovered an amazingly novel algorithm to predict whether or not it will snow on the day of the final exam, based on n days worth of weather data. The time complexity of their algorithm is given by the recurrence relation

$$T(n) = \begin{cases} 2T(n/3) + T(2n/3) + cn^2 & \text{if } n \geq 2 \\ c & \text{if } n = 1. \end{cases}$$

Using a method of your choice, prove upper and lower bounds on the function $T(n)$. Your grade will depend partly on the quality of the bound you provide (so, proving that $T(n) \in \Omega(1)$ and $T(n) \in O(100^n)$, while true, will not give you many marks).

Solution: One approach to obtain a tight upper bound is to draw a recursion tree. The following pictures shows the first three levels of the tree: the expressions in each node represent the number of items the recursive call receives, the amount of work done at each node is in blue next to the node, and the amount of work done on each row is in red.



We see that the amount of work done on row i is $c(2/3)^i n^2$, and so if we let the number of levels of the recursion go to infinity and use the formula for the sum of an infinite geometric series, we see that

$$T(n) = \sum_{i=1}^{\infty} c(2/3)^i n^2 = 3cn^2$$

is an upper bound on the amount of work done.

Alternately, we can use mathematical induction to prove that $T(n) \leq 3cn^2$. The base case is when $n = 1$. Then $T(1) = c \leq 3c1^2$, so the claim holds.

For the inductive step, suppose that the claim is true for all $i, 1 \leq i \leq n-1$. We show it is also true for n:

$$\begin{aligned} T(n) &= 2T(n/3) + T(2n/3) + cn^2 \\ &\leq 2 * 3c(n/3)^2 + 3c(2n/3)^2 + cn^2 \\ &= 6cn^2/9 + 12cn^2/9 + 9cn^2/9 \\ &= 27cn^2/9 = 3cn^2. \end{aligned}$$

This completes the upper bound, and so we have that $T(n) \in O(n^2)$.

For the lower bound, we observe from the recurrence that $T(n) \geq cn^2$ for all n. Therefore we also have that $T(n) \in \Omega(n^2)$, which means that $T(n) \in \Theta(n^2)$.

5 Pell numbers

Recall the Pell numbers, defined by the following recurrence relation, and the algorithm Pell(n) which requires exponential time to compute the nth Pell number:

$$P_n = \begin{cases} 0, & \text{if } n = 0, \\ 1, & \text{if } n = 1, \\ 2P_{n-1} + P_{n-2}, & \text{otherwise.} \end{cases}$$

```
Algorithm Pell($n$)
// Returns the $n$th Pell number
    If $n = 0$ then
        Return 0
    ElseIf $n = 1$ then
        Return 1
    Else
        Return 2 * Pell($n - 1$) + Pell($n - 2$)
```

1. **[3 marks]** Use memoization to obtain a more efficient recursive algorithm for calculating Pell numbers.

```
Memo-Pell(n):  // n is nonnegative
  Create a new array Soln[0,1,... n] of length n+1

  Initialize Soln[0] to:  0

  If n >= 1 : Initialize Soln[1] to:  1

  If n >= 2 : Initialize each element Soln[i] for 2 <= i <= n to -1
              // Here, -1 is a flag indicating that Soln[i] is not yet computed

  Return PellHelper(n, Soln)


PellHelper(n, Soln):

  If (Soln[n] == -1):   // Soln[n] is not yet computed
    // Recursively compute and store the answer
    Soln[n] = 2*PellHelper(n-1,Soln) + PellHelper(n-2,Soln)

  // By this point, Soln[n] is computed
  Return Soln[n]
```

2. **[1 mark]** What is the running time of your Algorithm Memo-Pell from part 1? Check one.

   [✓ $\Theta(n)$] ○ $\Theta(n \log n)$ ○ $\Theta(n^2)$ ○ $\Theta(2^n)$ ○ $\Theta(3^n)$

3. **[3 marks]** Rewrite your Memo-Pell algorithm without using recursion.

   Solution: `Memo-Pell(n):  // n is nonnegative`

```
If n = 0 : Return 0
Else If n = 1 : Return 1
Else
   Create a new array Soln[0,1,... n] of length n+1
   Initialize Soln[0] to 0
   Initialize Soln[1] to 1
   For i from 2 to n :
       Soln[i] = 2* Soln[i-1] + Soln[i-2]
   Return Soln[n]
```

4. **[1 mark]** What is the running time of your algorithm from part 3? Check one.

   [✓ $\Theta(n)$] ○ $\Theta(n \log n)$ ○ $\Theta(n^2)$ ○ $\Theta(2^n)$ ○ $\Theta(3^n)$

5. **[2 marks]** The previous algorithms use array Soln, which has $n + 1$ entries and thus uses $\Theta(n)$ memory. Briefly describe how to obtain an algorithm with the same running time that uses $O(1)$ memory.

   Solution: To compute Soln[i], only Soln[i-1] and Soln[i-2] are needed, and so entries Soln[j] for $j < i$ need not be stored. Instead, we only need two variables: `currentValue` and `prevValue` that will contain the values of `Soln[i-1]` and `Soln[i-2]` respectively.

```
Memo-Pell(n):  // n is nonnegative

  If n = 0 : Return 0
  Else If n = 1 : Return 1
  Else
     Create a new array Soln[0,1,... n] of length n+1
     Initialize prevValue to 0
     Initialize currentValue to 1
     For i from 2 to n :
         nextValue = 2* currentValue + prevValue
         prevValue = currentValue
         currentValue = nextValue
     Return currentValue
```