Assignment 2 Solutions

1 Long integers [10 marks]

(a) Consider the following implementation of the Parse algorithm.

Algorithm 1: Parse($s[0 \dots n-1]$)

```
S ← LongInt(0);
M ← LongInt(1);
for i = 0, ..., n − 1 do
    S ← Add(S, MulWithInt(M, s[i]));
    M ← MulWithInt(M, 10);
return S;
```

In terms of n, give a big-Θ bound for the time complexity of the Parse algorithm. Note: arithmetic operations on primitive types like array indices and loop variables all have time complexity $\Theta(1)$.

Solution. At the start of the ith iteration of the loop, both $\log M$ and $\log S$ are $\Theta(i)$ since $M = 10^i$ and $0 \leq S < 10^i$. Thus, each `Add` and `MulWithInt` operation in loop iteration i has cost bounded by $\Theta(i)$ bit operations. The total cost in bit operations of the loop is given by $\sum_{i=0}^{n-1} \Theta(i)$, which we know to be $\Theta(n^2)$.

(b) Use the divide and conquer approach to design a different Parse algorithm that solves the same problem but has time complexity $\Theta(n^{\log_2 3})$.

Solution. There are a number of possible solutions. They are all similar in that they use two recursive calls on problems of about half the size, and a combining cost of $\Theta(n^{\log 3})$ bit operations. Here's one solution.

We may assume without loss of generality that n is a power of two. (Technically, this requires a proof but it's ok if students don't include it.) With this assumption, we have the following divide and conquer algorithm.

```
parse := proc(s[0..n-1])

  # BASE CASE
```

```
  if n = 1 then return DigitToInteger(s[0]) fi;

  # DIVIDE
  s1[0...n/2-1] := s[0...n/2-1];
  s2[0...n/2-1] := s[n/2...n-1];

  # CONQUER
  S1 := parse(s1);
  S2 := parse(s2);

  # COMBINE
  B := Init(10);
  for i from 1 to (lg n) - 1 do
     B := Mul(B,B)
  od;

  return Add(S1,Mul(B,S2))

end:
```

Proof of correctness. The key identity is

$$S = \sum_{i=0}^{n-1} s[i] \times 10^i = \overbrace{\left(\sum_{i=0}^{n/2-1} s[i] \times 10^i\right)}^{S_1} + \overbrace{\left(\sum_{i=0}^{n/2-1} s[n/2+i] \times 10^i\right)}^{S_2} \times 10^{n/2}.$$

The for loop in the combine step computes

$$B = 10^{2^{\log n - 1}} = 10^{n/2}$$

so that the expression on the right-hand side of the identity is exactly what is returned by the algorithm.

Running time analysis First we show that the combine phase has cost $\Theta(n^{\lg 3})$ bit operations. The computation in the final return statement has cost $\Theta(n^{\lg 3})$ bit operations since both $\log S_1$ and $\log S_2$ are $\Theta(n)$. Now consider the for loop to compute B. At the start of the ith iteration, $\log B$ is $\Theta(2^i)$ since $B = 10^{2^{i-1}}$. Thus, the `Mul` operation in the ith iteration has cost $\Theta((2^i)^{\lg 3})$ bit operations. Noting that $(2^i)^{\lg 3} = 3^i$, we arrive at a total cost for the loop of $\sum_{i=1}^{(\lg n)-1} \Theta(3^i)$ bit operations, a truncated geometric series with solution $\Theta(3^{\lg n})$. Finally, note that $3^{\lg n} = n^{\lg 3}$.

Now, let $T(n)$ be the running time in bit operations of the algorithm, recalling that n is a power of two. Then $T(1) = \Theta(1)$ and $T(n) = 2T(n/2) + \Theta(n^{\lg 3})$, the solution of which satisfies $T(n) \in \Theta(n^{\lg 3})$ using the master theorem.

2 Diameter of a tree [10 marks]

The *diameter* of a tree is the length of the longest simple path between nodes of the tree. Design an efficient divide and conquer algorithm to compute the diameter of a rooted binary tree. Your algorithm may use the following operations on trees that each have time complexity $\Theta(1)$.

- Root(T) — Returns the root node of the tree T.
- LeftChild(T, v) — Returns the node that is the left child of v in T; or $\emptyset$ if v does not have a left child.
- RightChild(T, v) — Returns the node that is the right child of v in T; or $\emptyset$ if v does not have a right child.

Solution. For any node Y in a binary tree T, let its left and right children be denoted Y_L and Y_R. Let $\mathsf{depth}(Y)$ denote the depth of the binary (sub)-tree with root node Y (i.e., the length of a longest path from Y to a leaf node of the given (sub)-tree), and let $\mathsf{diameter}(Y)$ denote the diameter of the binary (sub)-tree with root node Y.

The following recurrence holds:

$$\mathsf{diameter}(Y) = \max\{\mathsf{diameter}(Y_L), \mathsf{diameter}(Y_R), \mathsf{depth}(Y_L) + \mathsf{depth}(Y_R) + 2\}$$

and

$$\mathsf{depth}(Y) = \max\{\mathsf{depth}(Y_L) + 1, \mathsf{depth}(Y_R) + 1\},$$

provided that base cases are handled appropriately.

The above recurrence is justified as follows. Suppose that P is the longest path in the (sub)-tree with root node Y. P must be a path between two leaf nodes, say A and B. There are three cases to consider:

(i) A and B are both in the left subtree of node Y. In this case, P has length $\mathsf{diameter}(Y_L)$.

(ii) A and B are both in the right subtree of node Y. In this case, P has length $\mathsf{diameter}(Y_R)$.

(ii) A is in the left subtree of node Y and B is in the right subtree of node Y. Then P passes through node Y. The portion of P from A to Y must be a path of length $\mathsf{depth}(Y_L) + 1$ and the portion of P from Y to B must be a path of length $\mathsf{depth}(Y_R) + 1$. The length of P is therefore $\mathsf{depth}(Y_L) + \mathsf{depth}(Y_R) + 2$.

The formula considers all three cases, and chooses the largest answer to be $\mathsf{diameter}(Y)$.

In order to evaluate the formula for $\mathsf{diameter}(Y)$, we need to know the values of $\mathsf{depth}(Y_R)$ and $\mathsf{depth}(Y_L)$. These are also computed recursively, as described above.

The base cases occur when one or both subtrees are empty; we do one or zero recursive calls in this case, as indicated in the following pseudocode.

Algorithm 2.1: BinaryTreeDiameter(X, di, de)

```
if leftchild(X) = nil
    then { if rightchild(X) = nil
             then { de ← 0
                    di ← 0
                    return (di, de)
           else { X_R ← rightchild(X)
                  BinaryTreeDiameter(X_R, di_R, de_R)
                  de ← de_R + 1
                  di ← max{di(X_R), de(X_R) + 1}
                  return (di, de) }
    else { if rightchild(X) = nil
             then { X_L ← leftchild(X)
                    BinaryTreeDiameter(X_L, di_L, de_L)
                    de ← de_L + 1
                    di ← max{di(X_L), de(X_L) + 1}
                    return (di, de)
           else { X_L ← leftchild(X)
                  X_R ← rightchild(X)
                  BinaryTreeDiameter(X_L, di_L, de_L)
                  BinaryTreeDiameter(X_R, di_R, de_R)
                  di ← max{di(X_L), di(X_R), de(X_L) + de(X_R) + 2}
                  de ← max{de(X_L) + 1, de(X_R) + 1}
                  return (di, de) }
```

Let n denote the number of nodes in a rooted binary tree T, let n_L denote the number of nodes in the left subtree and let n_R denote the number of nodes in the right subtree. Note that $n = n_L + n_R + 1$. Then we obtain the running time recurrence

$$t(n) = \begin{cases} 0 & \text{if } n = 0 \\ t(n_L) + t(n_R) + \Theta(1) & \text{if } n > 0. \end{cases}$$

There is a positive constant d such that $t(n) \leq t(n_L) + t(n_R) + d$ for all $n > 0$. Then it is easy to prove by induction that $t(n) \leq dn$. Clearly this is true for $n = 0, 1$, which we take to be base cases. Now assume that $t(j) \leq dj$ for $0 \leq j \leq n - 1$. Then we have that

$$\begin{aligned} t(n) &\leq t(n_L) + t(n_R) + d \\ &\leq d\,n_L + d\,n_R + d \qquad \text{by induction} \\ &= d(n_L + n_R + 1) \\ &= dn, \end{aligned}$$

as desired. This proves that $t(n) \in O(n)$.

To prove that $t(n) \in \Omega(n)$, observe that there is one recursive call done for every node in the tree. This already takes time $\Omega(n)$.

Finally, because $t(n) \in O(n)$ and $t(n) \in \Omega(n)$, we have that $t(n) \in \Theta(n)$.

3 Making change [10 marks]

Prove or disprove the following statements regarding the greedy coin changing algorithm GreedyChange seen in class. In the case of statements that are true, you should include a complete proof by induction. And for statements that are false, you should include a counter-example.

(a) The GreedyChange algorithm always returns an optimal solution to coin changing problem when the set of denominations $d_1 > d_2 > ... > d_n = 1$ are such that d_i divides d_{i-1} for every $i = 2, 3, ..., n$.

Solution. True

Proof. There are a few different possible proofs (by induction or contradiction). One proof that I like starts with the observation that there is a unique representation of v as a sum

$$v = \sum_{i \leq n} a_i d_i$$

that satisfies $0 \leq a_i < \frac{d_{i-1}}{d_i}$ for each $i \geq 2$.

We first claim that the numbers $a_1, \ldots, a_n$ in the above representation correspond to the number of coins of each denomination returned by the greedy algorithm. That's because if it returned any value $a_i \geq \frac{d_{i-1}}{d_i}$, then this would contradict the fact that it maximized the number a_{i-1} of coins of denomination d_{i-1} it returned since it could have added at least one more.

Then we claim that the numbers $a_1, \ldots, a_n$ also correspond to the optimal number of coins that can be used to make change for v. That's because any solution $(b_1, \ldots, b_n)$ that has any value $b_i \geq \frac{d_{i-1}}{d_i}$ can be improved by replacing at least $\frac{d_{i-1}}{d_i} \geq 2$ coins of denomination d_i with a single coin of denomination d_{i-1}. So the optimal solution must satisfy $0 \leq b_i \leq \frac{d_{i-1}}{d_i}$ for each $i \geq 2$ and, by the uniqueness of the representation above, we must have $(b_1, \ldots, b_n) = (a_1, \ldots, a_n)$. □

(b) The GreedyChange algorithm always returns an optimal solution to coin changing problem when the set of denominations $d_1 > d_2 > ... > d_n = 1$ are such that $d_{i-1} \geq 2d_i$ for every $i = 2, 3, ..., n$.

Solution. False

Proof. Consider the set of denominations $d_1 = 100$, $d_2 = 49$, $d_3 = 1$ and the value $v = 147$. The optimal solution is a set of 3 coins each of denomination $d_2 = 49$, but the greedy algorithm will instead choose 1 coin of denomination d_1 and 47 coins of denomination d_3, for a total of 48 coins. □

4 Making the deadline [10 marks]

In the Deadline problem, an instance is a set of n tasks that each take 1 unit of time to complete and a set of deadlines $d_1, \ldots, d_n$. When a task is not completed by its deadline, a fixed penalty of 100$ is applied. A valid solution to the Deadline problem is an ordering of the tasks that minimizes the total penalty incurred when the tasks are completed one after another in the order provided starting at time 0.

Design a greedy algorithm that solves the Deadline problem.

Solution. The idea of the algorithm is to sort the tasks by increasing deadlines. We then go through the tasks in that order and complete the next task whenever we have not passed its deadline yet. All the jobs whose deadlines already passed by the time we consider them are left to the end, at which point they are completed in any order.

Algorithm 2: TaskScheduler$(d_1, \ldots, d_n)$

```
Sort the tasks by deadlines so that d_1 ≤ d_2 ≤ ··· d_n;
J ← ();
L ← ();
for t = 1, 2, ..., n do
    if d_t ≤ t then
        J.Append(t);
    else
        L.Append(t);
return J.Append(L);
```

Proof of correctness. We can use an exchange argument. Assume without loss of generality (by relabeling the tasks if necessary) that the greedy algorithm returns the order $1\,2\,\cdots\,n$. Consider now any other ordering π of the tasks. We want to show that we can transform that ordering π into the greedy algorithm's ordering without increasing the number of tasks that are performed before their deadline (and, therefore, without decreasing the penalty paid).

Let i be the first task that is completed at time $t_i > i$ in the ordering π, and let j be the task completed at time $t_j = i$. Let π' be the order obtained by exchanging the positions of tasks i and j. There are four cases to consider:

1. Tasks i and j are both completed before their deadline in π. Then the design of the greedy algorithm is such that we must have $d_i \leq d_j$; this means that in π' task i is completed by time $t^* < t^* + 1 \leq d_i$ and task j is completed by time $t^* + 1 \leq d_i \leq d_j$ so both tasks are completed before the deadline in π' as well.

2. Tasks i and j are both completed after their deadline in π. Then exchanging the order of both tasks cannot decrease the number of tasks completed before their deadline since all other tasks are completed at the same time in π and π'.

3. Task i is completed before its deadline in π but task j is not. This case cannot occur: if j is completed after its deadline, we have $t_i > t_j > d_j \geq d_i$ so that i is also completed after its deadline.

4. Task j is completed before its deadline in π but task i is not. The greedy algorithm schedules task i after tasks $1, 2, \ldots, i-1$ only if (i) it is the task with smallest remaining deadline that has not passed, or (ii) all tasks have already passed their deadline. In the first case, then in π' task i is completed by its deadline and we again do not increase the penalty by the exchange; the second case cannot occur without contradicting the assumption that task j can still be completed in time.

Time complexity analysis. The sorting of the tasks by deadlines takes time $\Theta(n \log n)$. We can implement lists so that each Append operation takes constant time, so that the loop itself takes time $\Theta(n)$. The total time complexity of the algorithm is therefore $\Theta(n \log n)$.

5 Efficient polynomial multiplication [20 marks]

For this problem, you will design, analyze, and implement an algorithm that solves the following problem. The description, pseudocode, proof of correctness, and time complexity analysis will be submitted through Crowdmark. Your implementation will be submitted through Marmoset.

The problem. Given two polynomials with integer coefficients

$$P = \sum_{i=0}^{n} p_i x^i = p_n x^n + p_{n-1} x^{n-1} + \cdots + p_1 x + p_0,$$

$$Q = \sum_{i=0}^{m} q_i x^i = q_m x^m + q_{m-1} x^{m-1} + \cdots + q_1 x + q_0$$

and a nonzero integer number A, determine if the equality

$$P^2 = A \cdot Q$$

is true or false.

Solution. The idea of the solution is to follow the template for fast integer multiplication. For any questions about the specific details of this solution, visit any of the office hours.