CSE 2331 Homework 3 Solutions

1. Give the best closed-form asymptotic bounds that you can on the following recurrence relations. You may assume that for any fixed constant, c, $T(c) \in \Theta(1)$. Show your work.

   (a) Given $T(n) = T(n-1) + \Theta(1)$, we find $T(n) \in \Theta(n)$.

   Proof. The problem statement says that there exists $c_1, c_2 \in \mathbb{R}^+$ such that, for sufficiently large n,

   $$T(n-1) + c_1 \le T(n) \le T(n-1) + c_2.$$

   Note, it suffices to solve $T'(n) = T'(n-1) + c$, as this recurrence relation occurs on both sides of the inequality (with $c = c_1$, $c = c_2$, respectively).
   After one substitution,
   $$T'(n) = T'(n-2) + 2c.$$

   After two substitutions,
   $$T'(n) = T'(n-3) + 3c.$$

   After $k-1$ substitutions, we have that

   $$T'(n) = T'(n-k) + k \cdot c.$$

   This terminates when $n - k = 1$. That is, when $k = n - 1$. So plugging back in,

   $$T'(n) = T'(1) + (n-1) \cdot c \in \Theta(n).$$

   Thus, applying this solution to the right hand side,

   $$T(n) \le T(1) + (n-1) \cdot c_2 \in O(n).$$

   At the same time, applying it to the left hand side gives

   $$T(n) \ge T(1) + (n-1) \cdot c_1 \in \Omega(n).$$

   We conclude $T(n) \in \Theta(n)$.

   □

   (b) Given $T(n) = T(n-1) + O(n)$, we find $T(n) \in O(n^2)$, $T(n) \in \Omega(n)$.

   Proof. The problem statement means that

   $$T(n) = T(n-1) + g(n)$$

   for some $g(n) \in O(n)$.

Therefore, there exists some $c \in \mathbb{R}^+$ such that, for sufficiently large n,

$$T(n) \leq T(n-1) + nc.$$

After one substitution,

$$T(n) \leq T(n-2) + (n-1)c + nc.$$

After two substitutions,

$$T(n) \leq T(n-3) + (n-2)c + (n-1)c + nc.$$

After $k-1$ substitutions, we have that

$$T(n) \leq T(n-k) + c\sum_{i=0}^{k-1}(n-i).$$

Now note

$$\sum_{i=0}^{k-1}(n-i) = \sum_{i=0}^{k-1} n - \sum_{i=0}^{k-1} i = k \cdot n - \frac{1}{2}(k-1)(k-2).$$

This terminates when $n - k = 1$. That is, when $k = n - 1$. So plugging back in,

$$T(n) \leq T(1) + c\left[(n-1)n - \frac{1}{2}(n-2)(n-3)\right] \in O(n^2)$$

At the same time, since $g(n)$ represents the running time of a (non-empty) body of an algorithm, $g(n) \in \Omega(1)$. Thus,

$$T(n) \geq T(n-1) + c.$$

By the previous solution, the RHS is in $\Theta(n)$, and thus it follows that $T(n) \in \Omega(n)$. □

(c) Given $T(n) = T(n/2) + O(1)$, we find $T(n) \in \Theta(\log n)$.

Proof. The problem statement means that

$$T(n) \leq T(n/2) + g(n)$$

for some $g(n) \in O(1)$. At the same time, since $g(n)$ represents the running time of a (non-empty) body of an algorithm, $g(n) \in \Omega(1)$. Therefore, there exists $c_1, c_2 \in \mathbb{R}^+$ such that, for sufficiently large n,

$$T(n/2) + c_1 \leq T(n) \leq T(n/2) + c_2.$$

Note, it suffices to solve $T'(n) = T'(n/2) + c$, as this recurrence relation occurs on both sides of the inequality (with $c = c_1$, $c = c_2$, respectively).
After one substitution,
$$T'(n) = T'(n/2^2) + 2c.$$
After two substitutions,
$$T'(n) = T'(n/2^3) + 3c$$
After $k-1$ substitutions,
$$T'(n) = T'(n/2^k) + kc$$
This terminates when $n/2^k \leq 1$. That is, when $k \geq \log_2(n)$. So,
$$T'(n) = T'(1) + c \cdot \log_2(n) \in \Theta(\log(n)).$$
Thus, applying this solution to the right hand side,
$$T(n) \leq T(1) + c_2 \cdot \log_2(n) \in O(\log(n)).$$
At the same time, applying it to the left hand side gives
$$T(n) \geq T(1) + c_1 \cdot \log_2(n) \in \Omega(\log n).$$
We conclude $T(n) \in \Theta(\log n)$

□

(d) Given that $T(n) = T(n/2) + \Theta(n)$, we find $T(n) \in \Theta(n)$.

Proof. The problem statement means that
$$T(n) = T(n/2) + g(n)$$
for some $g(n) \in \Theta(n)$. Therefore, there exists $c_1, c_2 \in \mathbb{R}^+$ such that, for sufficiently large n,
$$T(n/2) + c_1 n \leq T(n) \leq T(n/2) + c_2 n.$$
We first solve $T'(n) = T'(n/2) + cn$, since this appears on the left and right sides.
After one substitution,
$$T'(n) = T'(n/2^2) + cn/2 + cn.$$
After two substitutions,
$$T'(n) = T'(n/2^3) + cn/2^2 + cn/2 + cn$$

After $k-1$ substitutions,

$$T'(n) = T'(n/2^k) + cn\sum_{i=0}^{k-1}\left(\frac{1}{2^i}\right).$$

Now note that this terminates when $n/2^k \le 1$. That is, when $k \ge \log_2(n)$. We could plug this in to the upper bound of the sum to evaluation the solution. However, we can see by the n multiplying the sum that the upper bound is already linear for $k=1$. Further, this sum is a geometric series which, in the limit of large k, sums to 2. Thus,

$$T'(1) + cn\left(\frac{1}{2^0}\right) \le T'(n) \le T'(1) + 2cn.$$

Therefore, regardless of the choice of c,

$$T'(n) \in \Theta(n).$$

It follows that $T(n) \in \Theta(n)$ since it is sandwiched between two copies of $T'(n)$ which differ only by the choice of constant. □

(e) Given $T(n) = 2T(n/2) + O(n)$, we find that $T(n) \in O(n\log(n))$, and $T(n) \in \Omega(n)$.

Proof. The problem statement means that

$$T(n) = 2T(n/2) + g(n)$$

for some $g(n) \in O(n)$. At the same time, since $g(n)$ represents the running time of a (non-empty) body of an algorithm, $g(n) \in \Omega(1)$. Therefore, there exists $c_1, c_2 \in \mathbb{R}^+$ such that, for sufficiently large n,

$$2T(n/2) + c_1 \le T(n) \le 2T(n/2) + c_2 n.$$

First we handle the right hand side. After one substitution,

$$\begin{aligned} T(n) &\le 2\left[2T(n/2^2) + c_2(n/2)\right] + c_2 n \\ &= 2^2 T(n/2^2) + 2c_2 n. \end{aligned}$$

After two substitutions,

$$\begin{aligned} T(n) &\le 2^2\left[2T(n/2^3) + c_2(n/2^2)\right] + 2c_2 n \\ &= 2^3 T(n/2^3) + 3c_2 n. \end{aligned}$$

After $k-1$ substitutions,

$$T(n) \le 2^k T(n/2^k) + kc_2 n.$$

Now note that this terminates when $n/2^k \leq 1$. That is, when $k \geq \log_2(n)$. Observe,

$$T(n) \leq 2^{\log_2(n)} T(1) + \log_2(n) c_2 n = nT(1) + c_2 n \log_2(n).$$

Therefore,

$$T(n) \in O(n \log(n)).$$

We can apply the next problem to the LHS to find that it is $\Theta(n)$. Therefore, $T(n) \in \Omega(n)$. □

(f) Given $T(n) = 2T(n/2) + O(1)$, we find that $T(n) \in \Theta(n)$.

Proof. The problem statement means that

$$T(n) \leq 2T(n/2) + g(n)$$

for some $g(n) \in O(1)$. At the same time, since $g(n)$ represents the running time of a (non-empty) body of an algorithm, $g(n) \in \Omega(1)$. Therefore, there exists $c_1, c_2 \in \mathbb{R}^+$ such that, for sufficiently large n,

$$2T(n/2) + c_1 \leq T(n) \leq 2T(n/2) + c_2.$$

We first solve $T'(n) = 2T'(n/2) + cn$, since this appears on the left and right sides. After one substitution,

$$\begin{aligned} T'(n) &= 2\left[2T'(n/2^2) + c\right] + c \\ &= 2^2 T'(n/2^2) + 2c + c. \end{aligned}$$

After two substitutions,

$$\begin{aligned} T'(n) &= 2^2\left[2T'(n/2^3) + c\right] + 2c + c \\ &= 2^3 T'(n/2^3) + 2^2 c + 2c + c. \end{aligned}$$

After $k-1$ substitutions,

$$T'(n) = 2^k T'(n/2^k) + c\sum_{i=0}^{k-1} 2^i = 2^k T'(n/2^k) + c(2^k - 1)$$

This terminates when $n/2^k \leq 1$. That is, when $k \geq \log_2(n)$. So,

$$T'(n) \leq 2^{\log_2(n)} T'(1) + c(2^{\log_2(n)} - 1) = nT'(1) + c(n-1).$$

So, regardless of the value of c,

$$T'(n) \in \Theta(n).$$

It follows that $T(n) \in \Theta(n)$ since it is sandwiched between two copies of $T'(n)$ which differ only by the choice of constant. □

2. Write a recurrence relation describing the worst case running time of each algorithm. Solve the recurrence relation to determine the asymptotic complexity. In each, you may assume that $0 \le n \le \text{len}(xs)$. Also, the double-divison notation $a//b$ evaluates to the integer $\text{floor}(a/b)$.

   (a) Listing 1.

```
 1  def f1(xs, n):
 2      if n < 10:
 3          return xs[n-1]
 4
 5      x = 0
 6      for i in range(10):
 7          for j in range(n):
 8              xs[j] = xs[j] - xs[i]
 9
10          x = x + f1(xs, n//10)
11      return x
```

Proof. There is no asymptotically-relevant conditional code. Therefore the worst-case running time is given by

$$T(n) = \sum_{i=0}^{9} \left[\left(\sum_{j=0}^{n-1} c' \right) + T(n/10) \right] = 10c'n + 10T(n/10),$$

where c' is the number of instructions in one execution of the for-loop on line 7. For simplicity we'll let $c = 10c'$, so that we solve $T(n) = 10T(n/10) + cn$.
After one substitution,

$$T(n) = 10 \left[10T\left(\frac{n}{10^2}\right) + cn\frac{1}{10} \right] + cn = 10^2 T\left(\frac{n}{10^2}\right) + 2cn.$$

After two substitutions,

$$T(n) = 10^2 \left[10T\left(\frac{n}{10^3}\right) + cn\frac{1}{10^2} \right] + 2cn = 10^3 T\left(\frac{n}{10^3}\right) + 3cn.$$

After $k-1$ substitutions,

$$T(n) = 10^k T\left(\frac{n}{10^k}\right) + kcn.$$

This terminates when $n/10^k \le 1$. That is, when $\log_{10}(n) \le k$. Plugging back in,

$$T(n) = 10^{\log_{10}(n)} T(1) + \log_{10}(n)cn = nT(1) + cn\log(n) \in \Theta(n\log(n)).$$

$\square$

(b) Listing 2.

```
1  def f2(xs, n):
2      if n < 5:
3          return xs[n-1]
4
5      if xs[0] < xs[n-1]:
6          return xs[1] + f2(xs, (2*n)//3)
7      else:
8          return xs[2] - f2(xs, n//5)
```

Proof. Since we are performing worst-case analysis we consider what kind of input maximizes the running time. There are two possible (but mutually-exclusive) recursive calls: one which processes roughly two-thirds of the input, the other which processes one-fifth. If we assume that we always recurse on Line 6 then solving $T(n) \leq T\left(\frac{2}{3}n\right) + c$ will provide an upper-bound on the over-all running time. At the same time, any array xs with distinct entries which has its smallest entry first will be among the worst-case instances for its size and, in particular when they contain at least 5 elements, only recurse on Line 6. Therefore, $T(n) \geq T\left(\frac{2}{3}n\right) + c$. Therefore, the worst-case running time is given by

$$T(n) = T\left(\frac{2}{3}n\right) + c.$$

After one substitution,

$$T(n) = \left[T\left(\left(\frac{2}{3}\right)^2 n\right) + c\right] + c = T\left(\left(\frac{2}{3}\right)^2 n\right) + 2c.$$

After two substitutions,

$$T(n) = \left[T\left(\left(\frac{2}{3}\right)^3 n\right) + c\right] + 2c = T\left(\left(\frac{2}{3}\right)^3 n\right) + 3c.$$

After $k-1$ substitutions,

$$T(n) = T\left(\left(\frac{2}{3}\right)^k n\right) + kc.$$

This terminates when $\left(\frac{2}{3}\right)^k n \leq 1$. That is, when $k \geq \log_{3/2}(n)$. Pluggin in,

$$T(n) = T(1) + c\log_{3/2}(n) \in \Theta(\log(n)).$$

$\square$

(c) Listing 3.

```
 1  def f3(xs, n):
 2      if n < 5:
 3          return xs[n-1]
 4
 5      i = n - 3
 6
 7      x = 0
 8      while i > n//2:
 9          for j in range(n//2):
10              xs[j] = xs[2*j] - xs[i]
11          x += f3(xs, i)
12          i -= 12
13      return x
```

Proof. There is no asymptotically-relevant conditional code. Therefore, the worst-case running time is given by

$$T(n) = \sum_{\ell=1}^{k} \left[T(i(\ell)) + \sum_{j=0}^{n/2-1} c \right] = \left(\sum_{\ell=1}^{k} T(i(\ell)) \right) + kcn/2,$$

where $i(\ell)$ is the value of i prior to Line 12 in the ℓ-th execution of the while-loop body. Here, the outer sum is taken over the $\ell = 1, 2, \ldots, k$ executions of the while-loop body for some yet-to-be-determined k.

To fully express the running time in terms of n, must figure out i, and k. To figure out $i(\ell)$ we can make a chart. Initially, $i = n-3$, so $i(1) = n-3$. Each subsequent execution subtracts 12, thus $i(\ell) = n - 3 - 12(\ell - 1)$. To figure out k note that the outer-loop terminates once $i(\ell) \leq n/2$. That is, once $n - 3 - 12(\ell - 1) \leq n/2$. Solving for ℓ, we find that it stops once $l \geq (n+18)/24$. That is, $k = (n+18)/24$. Plugging in,

$$T(n) = \sum_{\ell=1}^{(n+18)/24} T(n - 3 - 12(\ell - 1)) + cn(n + 18)/48.$$

Okay lets stop and think, because this got ugly. The formula says something like we make linearly many recursive calls where we subtract a constant, and do $\Theta(n^2)$ work in the body.

Recall, from the Fibonacci example, that even two recursive calls with a $\Theta(1)$-time body had an exponential-time lower-bound. In this class, it is enough to give an exponential-time lower-bound only, since this meant we'd never be able to use this in practice.

Our expression should also have an exponential lower-bound by comparison to the Fibonacci example. Let's just find an easy lower-bound here by throwing away all but two recursive calls, and lower the running-time of the body to be constant. In other words, note that

$$T(n) \geq \sum_{\ell=1}^{2} T(i(\ell)) + c = T(n-3) + T(n-15) + c \geq 2T(n-15) + c.$$

Let's consider $T(n) \geq 2T(n-15) + c$. After one substitution,

$$T(n) \geq 2[2T(n-30) + c] + c = 2^2 T(n-30) + 2c + c.$$

After two substitutions,

$$T(n) \geq 2^2[2T(n-45) + c] + 2c + c = 2^3 T(n-45) + 2^2 c + 2c + c.$$

After $k-1$ substitutions,

$$T(n) \geq 2^k T(n-15k) + c\sum_{i=1}^{k-1} 2^i = 2^k T(n-15k) + c(2^k - 1).$$

This stops once $n - 15k \leq 1$. That is, once $k \geq \frac{1}{15}(n-1)$. Plugging in,

$$T(n) \geq 2^{\frac{1}{15}(n-1)} T(1) + c(2^{\frac{1}{15}(n-1)} - 1).$$

So $T(n) \in \Omega((2^{1/15})^n)$. Recall, a lower bound of $\Omega(b^n)$ is exponential provided $b > 1$.

□

3. Consider the depth-first search algorithm.

```python
visited = set()

def dfs(graph, node):
    visited.add(node)
    for neighbor in graph[node]:
        if neighbor not in visited:   # assume O(1) to test for membership
            dfs(graph, neighbor)
```

Here, `graph` is a dictionary mapping nodes to sets of neighbors. For example,

```python
graph = [
    # node: set of neighbors
    'A': {'B', 'C'},
    'B': {'A', 'D', 'E'},
    ...
]
```

(a) What is the best-case asymptotic running time for the above function?

Proof. The best-case occurs when the graph contains n nodes say and no edges. In this case, the function will run in $\Theta(1)$-time as we visit only the initial node and then return. $\square$

(b) Analogously, what is the worst-case asymptotic running time?

Proof. The worst-case occurs when the graph is the complete graph on n nodes. In this case, the function will run in $\Theta(n^2)$-time as we visit each node and then check each of its neighbors. Let's see why this is the case.

The running time of the loop is proportional to the number of neighbors of the current node. However, for the graph we have picked, every node has $n - 1$ neighbors. Therefore, the running time of the loop is proportional to $n - 1$.

Each time we visit a node we add it to the set of visited nodes. Therefore, the set of nodes that we will visit decreases by one each time. It's a little tricky to write down a recurrence because while the number of nodes we need to visit will shrink, the number of neighbors we examine in the loop remains constant. We don't want to write $T(n) = T(n-1) + c(n-1)$ because this would imply the number of nodes we are checking in the loop, i.e., the number of neighbors, is shrinking. Instead, let's write $T(n) = T(n-1) + c(|V|-1)$ where $|V|$ is the number of nodes in the graph and n is the number of nodes we need to visit. We'll substitute this back for the number of nodes in the graph at the end.

Starting from $T(n) = T(n-1) + c(|V|-1)$, after one substitution,

$$T(n) = T(n-2) + c(|V|-1) + c(|V|-1) = T(n-2) + 2c(|V|-1).$$

After two substitutions,

$$T(n) = T(n-3) + 3c(|V|-1).$$

After $k-1$ substitutions,

$$T(n) = T(n-k) + kc(|V|-1).$$

This terminates when $n - k = 1$. That is, when $k = n - 1$. So plugging back in for k and n for $|V|$,

$$T(n) = T(1) + (n-1)c(|V|-1) = c_2 + c(n-1)^2 \in \Theta(n^2).$$

$\square$

(c) Give the best possible upper bound on the asymptotic running time for arbitrary n. This should be denoted with O, Ω, or Θ (as appropriate). Explain your answer.

Proof. A generic graph has no restrictions on the number of edges it may contain. Therefore, by part (b), the running time must be $O(n^2)$. At the same time, by part (a), any run must take at least $\Omega(1)$-time. $\square$