# CSE 2331 Homework 2

1. Give the asymptotic running time of each function in asymptotic notation (i.e., as $\Theta$, $O$, $\Omega$, etc). Justify your answer.

(a)
```
def f1(n):
    s = 0
    for i in range(1, n//2 + 1):
        for j in range(i, n-i + 1):
            s = s*s
    return s
```

*Proof.* The outer loop runs $n/2$ times, and the inner loop runs $n-2i$ times in the $i$-th iteration of the outer loop. So

$$\begin{aligned}
c_1 + \sum_{i=1}^{n/2}\sum_{j=i}^{n-i} c_2 &= c_1 + c_2\sum_{i=1}^{n/2}(n-i-i+1) \\
&= c_1 + c_2\sum_{i=1}^{n/2}(n-2i+1) \\
&= c_1 + c_2\left[\sum_{i=1}^{n/2} n - 2\sum_{i=1}^{n/2} i + \sum_{i=1}^{n/2} 1\right] \\
&= c_1 + c_2\left[n\frac{n}{2} - 2\frac{n/2(n/2+1)}{2} + \frac{n}{2}\right] \\
&= c_1 + c_2\left[\frac{n^2}{2} - \frac{n^2}{2^2} - \frac{n}{2} + \frac{n}{2}\right] = c_1 + c_2\frac{n^2}{4}
\end{aligned}$$

Therefore, the running time is $\Theta(n^2)$. □

(b)
```
def f2(n):
    s = 0
    i = 1
    while (i < 5*n):
        for j in range(1, i**2 + 1):
            s = s*i + j
        i = i + 3
    return s
```

*Proof.* The inner loop runs $i^2$ times and takes constant time per iteration. So,

$$T(n) = c_1 + \sum_{\ell=1}^{n_\ell} c_2 i^2.$$

Now, $n_\ell$ depends on $n$ and $i(\ell)$. Observe that $i(\ell) = 1+3\ell$, and the loop terminates once $i(\ell) \geq 5n$, so once $\ell \geq (5n-1)/3$. It follows that $n_\ell = \lceil (5n-1)/3 \rceil$. Therefore,

$$T(n) = c_1 + c_2 \sum_{\ell=1}^{(5n-1)/3} (1+3\ell)^2.$$

Now, we can bound the sum from above. Observe,

$$\begin{aligned}
T(n) &\leq c_1 + c_2 \sum_{\ell=1}^{(5n-1)/3} (1+3\ell)^2 \leq c_1 + c_2 \sum_{\ell=1}^{(5n-1)/3} (4\ell)^2 \\
&= c_1 + 16c_2 \sum_{\ell=1}^{(5n-1)/3} \ell^2 \leq c_1 + 16c_2 \sum_{\ell=1}^{5n} \ell^2 \leq c_1 + 16c_2 \sum_{\ell=1}^{5n} (5n)^2 \\
&= c_1 + 16c_2(5n)^3.
\end{aligned}$$

So $T(n) \in O(n^3)$.
Now, we can bound the sum from below. Observe,

$$T(n) \geq c_2 \sum_{\ell=1}^{(5n-1)/3} \ell^2 \geq c_2 \sum_{\ell=1}^{n} \ell^2 \geq c_2 \sum_{\ell=n/2+1}^{n} \frac{n^2}{4} = c_2 \frac{n^3}{8}.$$

So $T(n) \in \Omega(n^3)$. Therefore, $T(n) \in \Theta(n^3)$. □

Another way to see this is using the fact that $\sum_{i=a}^{m} \Theta(i^p) = \Theta(m^{p+1})$.

*Proof.* Consider

$$T(n) = c_1 + c_2 \sum_{\ell=1}^{(5n-1)/3} (1+3\ell)^2.$$

Now observe

$$\begin{aligned}
T(n) &= c_1 + c_2 \sum_{\ell=1}^{m} \Theta(\ell^2) = c_1 + c_2\Theta(m^3) \\
&= c_1 + c_2\Theta\left(\left(\frac{5n-1}{3}\right)^3\right) \\
&= c_1 + c_2\Theta(n^3) = c_1 + \Theta(n^3) \in \Theta(n^3).
\end{aligned}$$

where the second equality follows from $\sum_{i=a}^{m} \Theta(i^p) = \Theta(m^{p+1})$, and $m = (5n-1)/3$. Keep in mind that we can only use the absorbing property of $\Theta$ notation a finite number of times, because each time we do we change the function that it's standing in for, though not by more than a constant factor asymptotically. If we change it some variable number of times that depends on $n$, those changes can add up to be more than a constant factor.

□

(c)

```
def f3(n):
    s = 0
    i = 1
    while (i < 3*n**7):
        for j in range(1, i + 1):
            s = s - i*j
        i = i * 3
    return s
```

*Proof.* The inner loop runs $i$ times and takes constant time per iteration. So,

$$T(n) = c_1 + \sum_{\ell=1}^{n_\ell} c_2 i(\ell).$$

Now, $n_\ell$ depends on $n$ and $i(\ell)$. Observe that $i(\ell) = 3^{\ell-1}$, and the loop terminates once $i(\ell) \geq 3n^7$, so once $\ell \geq 1 + \log_3(3n^7)$. It follows that $n_\ell = \lceil 1 + \log_3(3n^7) \rceil$. Therefore,

$$T(n) = c_1 + c_2 \sum_{\ell=1}^{1+\log_3(3n^7)} 3^{\ell-1} = c_1 + \frac{c_2}{3} \sum_{\ell=1}^{1+\log_3(3n^7)} 3^\ell.$$

Recall the formula for a geometric series,

$$\sum_{i=0}^{m} r^i = \frac{1 - r^{m+1}}{1 - r}.$$

Observe that we have

$$r^0 + \sum_{i=1}^{m} r^i = \frac{1 - r^{m+1}}{1 - r} = \frac{r^{m+1} - 1}{r - 1},$$

so we can rewrite the sum as

$$\sum_{i=1}^{m} r^i = \frac{r^{m+1} - 1}{r - 1} - 1.$$

Plugging in for $r = 3$ and $n = 1 + \log_3(3n^7)$, we get

$$T(n) = c_1 + \frac{c_2}{3}\left[\frac{3^{2+\log_3(3n^7)} - 1}{2} - 1\right] = c_1 + \frac{c_2}{3}\left[\frac{9 \cdot 3^{\log_3(3n^7)} - 1}{2} - 1\right]$$
$$= c_1 + \frac{c_2}{6}(27n^7 - 3) \in \Theta(n^7).$$

□

(d)
```python
def f4(xs):
    n = len(xs)

    x = 0
    for i in range(n):
        for j in range(3*(i**2)):
            x += i*i - j
    return x
```

*Proof.* The inner loop runs $3i^2$ times and takes constant time per iteration. So,

$$T(n) = c_1 + \sum_{i=0}^{n-1} c_2 3i^2.$$

Now, we can bound the sum from above and below, but here let's just use $\sum_{i=a}^{m} \Theta(i^p) = \Theta(m^{p+1})$.
So,

$$T(n) = c_1 + \sum_{i=0}^{n-1} c_2 3i^2 = c_1 + c_2 3\sum_{i=0}^{m} i^2 = c_1 + c_2\Theta(m^3) = \Theta((n-1)^3) = \Theta(n^3).$$

□

(e)
```python
def f5(xs):
    n = len(xs)

    x = 0
    for i in range(n):
        u = int(4*(n**(1/2)))
        for j in range(u):
            x += i*i - j

    return x
```

*Proof.* The inner loop runs $4\sqrt{n}$ times and takes constant time per iteration. So,

$$T(n) = c_1 + \sum_{i=0}^{n-1} c_2 4\sqrt{i}.$$

Now, using $\sum_{i=a}^{m} \Theta(i^p) = \Theta(m^{p+1})$, with $m = n - 1$ and $p = 1/2$,

$$T(n) = c_1 + \sum_{i=0}^{n-1} c_2 4\sqrt{i} = c_1 + c_2\Theta(m^{3/2}) \in \Theta(n^{3/2}).$$

□

(f)

```python
def f6(xs):
    n = len(xs)

    x = 1
    i = 32
    while i < n:
        x *= xs[i]
        i += 1

    return x
```

*Proof.* The loop runs $n - 32$ times (for $n > 32$) and takes constant time per iteration. So,

$$T(n) = c_1 + \sum_{i=0}^{n-32-1} c_2 = c_1 + c_2(n - 32) \in \Theta(n).$$

□

(g)

```python
def f7(xs):
    n = len(xs)

    x = 1
    i = 0
    while i < 64*n:
        x *= xs[i % n]
        i += 8            # note: addition

    return x
```

*Proof.* This is a simple single while loop where the body takes constant time per iteration, so

$$T(n) = \sum_{\ell=1}^{n_\ell} c_2 = \Theta(n_\ell).$$

So we only need to figure out only how many times the body runs. Observe that $n_\ell$ depends on $n$ and $i(\ell)$. Observe that $i(\ell) = 8\ell$ after the $\ell$-th time it's run, and the loop terminates once $i(\ell) \geq 64n$, so once $\ell \geq 8n$. It follows that $n_\ell = \lceil 8n \rceil$ and therefore $T(n) \in \Theta(n)$. □

(h)

```
def f8(xs):
    n = len(xs)

    x = 1
    i = 11
    while i < n**25:       # n^25 (n to the 25-th power)
        x *= xs[i % n]
        i *= 4             # note: multiplication

    return x
```

*Proof.* This is a simple single while loop where the body takes constant time per iteration, so

$$T(n) = \sum_{\ell=1}^{n_\ell} c_2 = \Theta(n_\ell).$$

So, again, we only need to figure out only how many times the body runs. Observe that $n_\ell$ depends on $n$ and $i(\ell)$. Observe that $i(\ell) = 4^\ell \cdot 11$ after the $\ell$-th time it's run, and the loop terminates once $i(\ell) \geq n^{25}$, so once $\ell \geq \log_4(n^{25}/11)$. It follows that $n_\ell = \lceil \log_4(n^{25}/11) \rceil$ and therefore $T(n) \in \Theta(\log(n))$.
If you aren't sure why $\log_4(n^{25}/11) \in \Theta(\log(n))$, recall that

$$\log_4(n^{25}/11) = \log_4(n^{25}) - \log_4(11) = 25 \log_4(n) - \log_4(11).$$

Since $\log_4(11)$ is a constant, and since $\log_4(n)$ is $\Theta(\log(n))$ by the change of base formula, we have that $25 \log_4(n) - \log_4(11) \in \Theta(\log(n))$. □

2. Consider the following functions. For each, give the asymptotic running time in $\Theta$ notation as a function of $n$. Justify your answer.

```
def reverse(ds):
    n = len(ds)
    for i in range(n//2):
        # swap digits at positions i and n-i-1
        ds[i], ds[n-i-1] = ds[n-i-1], ds[i]

def join(xs):
    # xs is an array of constant-sized integers
    # n = len(xs)

    res = ""
    for x in xs:
        res += str(x)
    return res

def asBase(n, b):
    if n == 0:
        return "0"

    digits = []
    while n > 0:
        # compute the remainder mod b
        digits.append(int(n % b))
        # (integer) divide n by b
        n //= b

    reverse(digits)
    return join(digits)
```

3. Consider the algorithm `f` which takes a positive integer and in turn calls functions `f_0(n)`, `f_1` where the running times of these correspond to $\Theta(n)$ and $\Theta(n^2)$, respectively.

*Proof.* The running time of `reverse` is $\Theta(n)$, since it runs a loop that runs $n/2$ times and takes constant time per iteration.

The running time of `join` is $\Theta(n)$, since it runs a loop that runs $n$ times and takes constant time per iteration.

The running time of `asBase` is $\Theta(\log(n))$. Here, the running time is

$$T(n) = c_1 + \sum_{i=0}^{n_\ell} c_2 + \Theta(n_\ell),$$

where the second term is the time to run the while loop and the third term is the time to run `reverse` since the number of digits in the digits array is equal to the number of iterations of the while loop.

Observe that $n_\ell$ depends on $n(\ell) = n/b^\ell$ after the $\ell$-th time it is run, and the loop terminates once $n(\ell) < 1$, so once $\ell \geq \log_b(n)$. It follows that $n_\ell = \lceil \log_b(n) \rceil$ and therefore $T(n) \in \Theta(\log(n))$. □

```
def f(xs):
    n = len(xs)

    i = xs[n-1]             # i is last item of xs
    if i % 2 == 0:
        return f_0(n)       # Theta(n), if i is even.

    return f_1(n)           # Theta(n^2), otherwise.
```

In the above function, `i % 2` denotes the remainder after dividing $i$ by 2. It is 0 when $i$ is even and 1 when $i$ is odd.

(a) What is the best case asymptotic running time for the above function? Note that since I am asking for the best case, you may assume $n$ is restricted to the class of input that makes the above algorithm run the fastest. For this class of input, the algorithm runs in time $\Theta(\textit{something})$. Explain your answer.

*Proof.* Since the question is asking for *best-case* runtime, we know we are given an increasing sequence of instances which *minimize* the number of executed instructions in the RAM model.

The code consists of two mutually exclusive branches (in the sense that no instance will cause both to run), which correspond to the parity (even-or-oddness) of the last element of the instance. We call lists with even last element *even* instances, and lists with odd last element, *odd* instances.

*Even* instances take $\Theta(n)$ time due to the $f_0 \in \Theta(n)$. In other words, for sufficiently large $n$, *even* instances cause at least $c_1 n$ instructions, and at most $c_2 n$ to be executed for some $c_1, c_2 \in \mathbb{R}^+$. On the other hand, any *odd* instance requires executing at least $c_3 n^2$ instructions, but not any more than $c_4 n^2$ for some $c_3, c_4 \in \mathbb{R}^+$. In other words, odd instances take $\Theta(n^2)$-time. Therefore, the best-case instances correspond to even instances, and the number of instructions executed is linear in its value. That is, the best-case running-time is $\Theta(n)$. □

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

*Proof.* Since the question is asking for *worst-case* runtime, we know we are given an increasing sequence of instances which *maximize* the number of executed instructions in the RAM model.

The code consists of two mutually exclusive branches which correspond to the parity of the instance. By the analysis of the previous part, *odd* instances maximize the number of executed instructions, ultimately taking $\Theta(n^2)$-time, compared to $\Theta(n)$-time for even instances. Therefore, the worst-case instances correspond to odd instances, and the number of instructions executed is quadratic in its value. That is, the worst-case running-time is $\Theta(n^2)$. □

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

*Proof.* The question does not ask us to perform *best* or *worst* case analysis, so we cannot assume the instances are selected either to maximize or minimize the number of executed instructions. Instead, we are given an increasing-in-size (but otherwise arbitrary) sequence of problem instances. We know from previous analysis that any sufficiently large instance causes at most $c_4n^2$ instructions to be executed. Therefore, there is an $O(n^2)$-runtime upper bound on general instances. We know it is not possible to lower this upper bound because some instances (in particular, the odd ones) *require* quadratic time. □

(d) Analogously, give the best possible lower bound on the asymptotic running time for arbitrary $n$, as a function of $n$.

*Proof.* The question does not ask us to perform *best* or *worst* case analysis, so we cannot assume the instances are selected either to maximize or minimize the number of executed instructions. Instead, we are given an increasing-in-size (but otherwise arbitrary) sequence of problem instances. We know from previous analysis that any sufficiently large instance causes at least $c_1n$ instructions to be executed. Therefore, there is an $\Omega(n)$-runtime lower bound on general instances. We know it is not possible to improve (increase) this lower bound because some instances (in particular, the even ones) are solved in linear time. □

(e) Suppose, instead, that all you know is that the running time of $f_0(n)$ is in $\Omega(n)$, while the running time of $f_1(n)$ remains $\Theta(n^2)$. What are the best lower and upper bounds you can give on the asymptotic running time for best case, worst case, and general $n$, as a function of $n$?

*Proof.* In this case our knowledge about the problem is weakened so that we no longer have an upper bound on the running time of even instances. In other words, we know that even instances takes time at least $c_1n$ but could take much more. Further, since all we have is a lower bound, we are not guaranteed that *any* even instance be solvable with only $c_2n$ instructions for some $c_2$. This means that we no longer know that the worst case instances are *odd* and that the best case instances are *even*. Instead, we must give ranges.

The best-case instances could be as low as $\Omega(n)$ since linear-time even instances might exist. On the other hand the won't be higher than quadratic since there is always an even instance not taking more than $O(n^2)$-time.

The worst-case instances could be arbitrarily bad. We do not know how to characterize the runtime of the long-running even instances, so we cannot justify any upper-bound on the worst case. For any runtime bound we might impose we can imagine that the even branch might take more time. On the other hand, we do know, for sure, that the odd instances require quadratic time, and so they are at least quadratic in the event that the missing upper-bound on the even branch is something low. Therefore the cheapest possible worst-case instances are at least quadratic $\Omega(n^2)$. This is all we can say.

The general case is always sandwiched between the best lower bound on the best-case and the worst upper-bound on the worst case. Therefore a general instance takes at least $\Omega(n)$-time but there is no justifiable upper bound. □

4. A *Mersenne prime* is a prime number with one less than a power of two. For example, $3 = 2^2 - 1$, $7 = 2^3 - 1$, $31 = 2^5 - 1$ are all Mersenne primes, but the prime numbers 2, 5, 11, 13, 17, 19, 23, 29 are not. The Lenstra–Pomerance–Wagstaff conjecture (LPWC) asserts that there are infinitely many Mersenne primes. We (human-kind) do not know whether or not LPWC is true.

   We can test for primes in something like time $O(\log(n)^{12})$. Assume, perhaps unrealistically for anything but the RAM model, that we can compute $2^n - 1$ for any $n$ in time $O(\log(n))$. Assume `f_0(n)` takes time $O(n^{24})$. Consider the following algorithm, which takes a positive integer as input:

```
def function(n):
    if is_prime(2**n - 1):
        return f_0(n)
```

   What is the best upper and lower bound you can give on the asymptotic runtime of the above algorithm,

   (a) assuming LPWC is true? (Use $O, \Omega, \Theta$ as appropriate.)
   (b) assuming LPWC is false? (Use $O, \Omega, \Theta$ as appropriate.)
   (c) assuming LPWC remains unknown? (Use $O, \Omega, \Theta$ as appropriate.)

   *Proof.* First, we only have upper bounds for each of the functions. However, we can assume that they execute at least one instruction, meaning there is a default constant-time lower bound for all parts.

   We now compute the upper bound. It takes $O(\log n)$-time to compute $2^n - 1$ and another $O(\log(2^n - 1)^{12}) = O(\log(2^n)^{12}) = O(n^{12})$-time to test it for being prime. Thus, in total, it takes $O(n^{12})$-time to evaluate the condition in the *if* statement. Now

(a) if LPWC is true then there are infinitely-many Mersenne primes. Thus, no matter how large $n$ may be, it is possible to run line 3. The result is an upper bound of $O(n^{12}) + O(n^{24}) = O(n^{24})$.

(b) if LPWC is false then there are finitely-many Mersenne primes. Thus, there is an $n_0$ such that $2^{n_0} - 1$ is the *last* Mersenne prime. Therefore, in the limit of large $n$, we only perform the $O(n^{12})$-time check. The result is an upper bound of $O(n^{12})$.

(c) if LPWC remains unknown, then indeed it may be true, and thus $O(n^{24})$ remains the lowest justifiable upper bound.

□