COP3410
Assignment 3.1

Author: Jessica W.

Instructor: Dr. Taebi

Date: February 19, 2022

Chapter 3: Algorithm Analysis

Problem R-3.8
Order the following functions by asymptotic growth rate.

$$4n\log n + 2n,\ 2^{10},\ 2^{\log n},\ 3n + 100\log n,\ 4n,\ 2^n,\ n^2 + 10n,\ n^3,\ n\log n$$

Solution: Firstly, note $2^{\log n} = n$.
Now, note that the order of asymptotic growth of the seven common functions (increasing) is:

$$1,\ \log n,\ n,\ n\log n,\ n^2,\ n^3,\ 2^n$$

We have:

$$4n\log n + 2n \text{ is } \Theta(n\log n)$$
$$2^{10} \text{ is } \Theta(1)$$
$$2^{\log n} = n \text{ is } \Theta(n)$$
$$3n + 100\log n \text{ is } \Theta(n)$$
$$4n \text{ is } \Theta(n)$$
$$2^n \text{ is } \Theta(2^n)$$
$$n^2 + 10n \text{ is } \Theta(n^2)$$
$$n^3 \text{ is } \Theta(n^3)$$
$$n\log n \text{ is } \Theta(n\log n)$$

Now, we simply need to order the functions with equal big-Theta notation, as they grow asymptotically at the same rate, up to constant factors.
First, 2^{10} is $\Theta(1)$, so it is asymptotically the smallest.
Then, n, $3n+100\log n$, and $4n$ are $\Theta(n)$. Note that as n tends to infinity, the $100\log n$ term grows far slower than n, so is negligible. Thus, these functions in increasing order of growth are: $n, 3n + 100\log n, 4n$.
Next are $n\log n$ then $4n\log n + 2n$ in that order, as both are $\Theta(n\log n)$ and clearly the second function has a greater constant factor.
Then, $n^2 + 10n$ is the only function that is $\Theta(n^2)$, and n^3 is the only function that is $\Theta(n^3)$.
Lastly, 2^n has the highest asymptotic growth rate, being exponential and $\Theta(2^n)$.

Thus, the functions ordered by increasing asymptotic growth rate are:

- 2^{10}
- $2^{\log n} = n$
- $3n + 100\log n$
- $4n$
- $n\log n$
- $4n\log n + 2n$
- $n^2 + 10n$
- n^3
- 2^n

■

Problem R-3.19
Show n is $\mathcal{O}(n \log n)$

Solution: Let $f(n) = n$ $g(n) = n \log n$. We want n_0, c such that:

$$n \geq n_0 : f(n) \leq c * g(n)$$

$$n \geq n_0 : n \leq c * n \log n \tag{1}$$

Note that when $n \geq 2$, $\log n \geq 1$ *.
$-$ * under the assumption that $\log n$ is $\log_2 n$
So, when $n \geq 2$, $n \leq n * \log n$.

Taking $n_0 = 2, c = 1$ in (1):

$$n \geq 2 : n \leq 1 * n \log n$$

Thus, $\boxed{\text{n is } \mathcal{O}(n \log n)}$. ■

Problem R-3.20
Show n^2 is $\Omega(n \log n)$

Solution: Let $f(n) = n^2$ $g(n) = n \log n$. We want n_0, c such that:

$$n \geq n_0 : f(n) \geq c * g(n)$$

$$n \geq n_0 : n^2 \geq c * n \log n \tag{2}$$

Note that when $n \geq 2$, $n \geq \log n \equiv 2^n \geq n \equiv 2^n - n \geq 0$.
The above can be seen clearly from a graph of the functions 2^n and n, or by noting that $2^2 - 2 = 2$ and when $n \geq 2 : \frac{d}{dx}\left(2^x - x\right) = 2^x \ln 2 - 1 \geq 2^2 \ln 2 - 1 = 4 \ln 2 - 1 \geq 0$.
So, for $n \geq 2, n * n \geq n * \log n$.

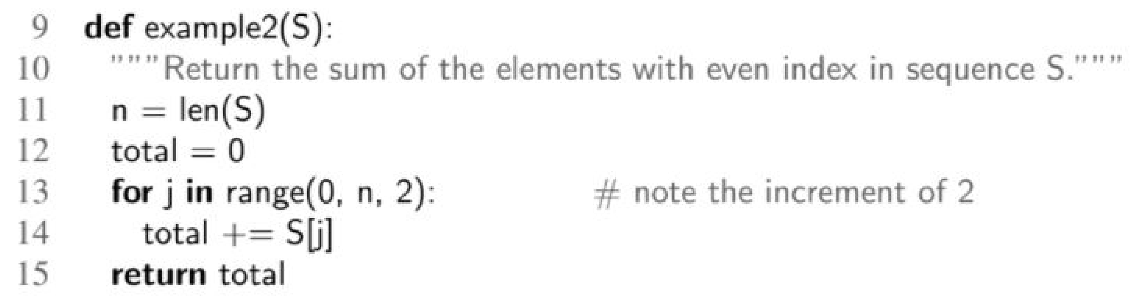
Taking $n_0 = 2, c = 1$ in (2):

$$n \geq 2 : n^2 \geq 1 * n \log n$$

Thus, $\boxed{\text{n}^2 \text{ is } \Omega(n \log n)}$. ■

> **Problem R-3.24**
> Give big-Oh characterization, in terms of n, of the running time of the example2 function in the code fragment below.

Solution:



```
 9  def example2(S):
10    """Return the sum of the elements with even index in sequence S."""
11    n = len(S)
12    total = 0
13    for j in range(0, n, 2):            # note the increment of 2
14      total += S[j]
15    return total
```

Figure 1: Function example2

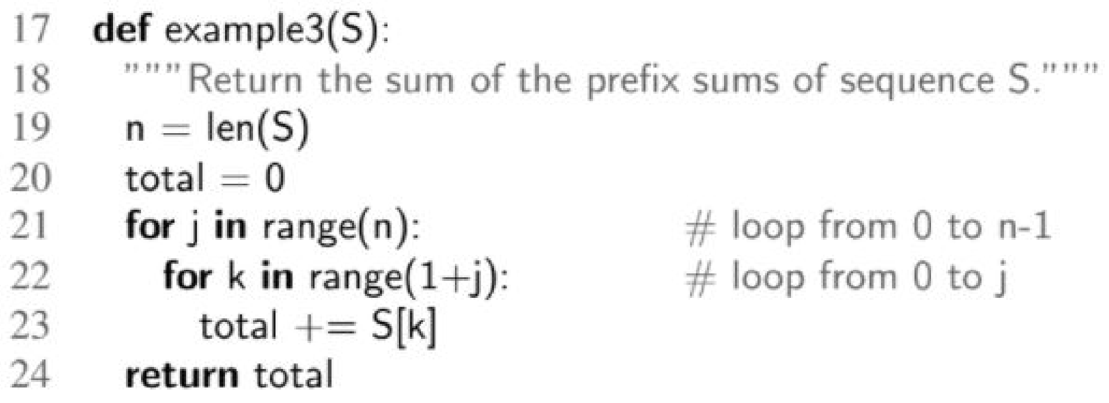
Algorithm analysis:

- On line 9, we define the function, **example2**, that returns the sum of elements with even index in S.
- Then, the statements, n = len(S) and total = 0 on lines 11 and 12, execute in constant, $\mathcal{O}(1)$, time.
- Then, we loop over even indices j from 0 to $n-1$. The statement, total + = S[j], requires constant operations, and is executed $\lceil \frac{n}{2} \rceil$ times, for $\mathcal{O}(n)$ run-time. The increments of j also require $\mathcal{O}(n)$ primitive operations. Thus, the for-loop on lines 13-14 runs in $\mathcal{O}(n)$ time.
- Finally, we return total on line 15; this operation is $O(1)$.

The run-time of **example2** is the sum of the run-times of the above steps; as a linear function is asymptotically greater than a constant function, and the steps have running time of $\mathcal{O}(1)$, $\mathcal{O}(n)$, $\mathcal{O}(n)$ respectively, the function has running time of order $\boxed{\mathcal{O}(n)}$. ■

Problem R-3.25
Give big-Oh characterization, in terms of n, of the running time of the example3 function in the code fragment below.

Solution:



```
17 def example3(S):
18   """Return the sum of the prefix sums of sequence S."""
19   n = len(S)
20   total = 0
21   for j in range(n):                # loop from 0 to n-1
22     for k in range(1+j):            # loop from 0 to j
23       total += S[k]
24   return total
```

Figure 2: Function example3

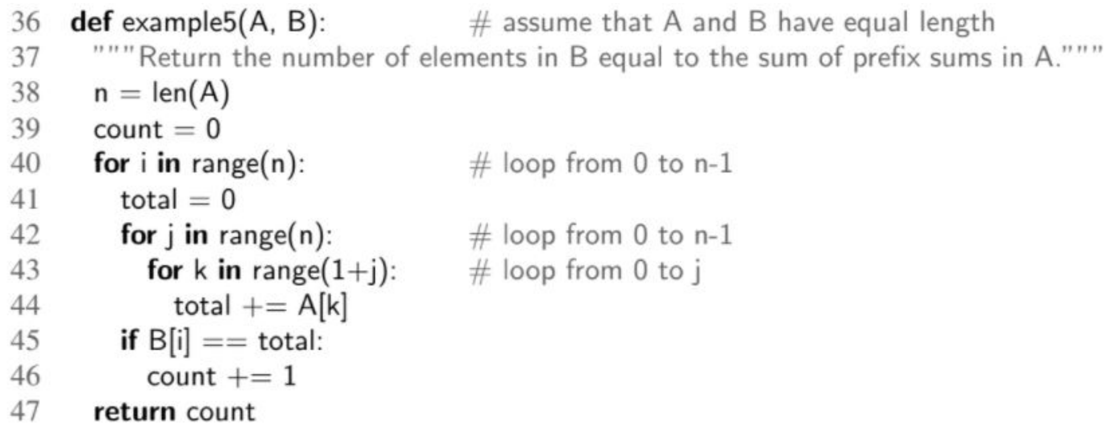
Algorithm analysis:

- On line 17, we define the function, **example3**, that returns the sum of the prefix sums of S.
- The statements, n = len(S) and total = 0 on lines 19 and 20, execute in constant, $\mathcal{O}(1)$, time.
- In the two nested for-loops, we have indices $j = 0, ..., n-1$ and $k = 0, ..., j$. The statement, total += S[k], requires constant operations and is executed $1 + ... + n = \frac{n(n+1)}{2}$ times ($j+1$ times for each index j), for $\mathcal{O}(n^2)$ running time on line 23.
- Additionally, initializing and incrementing j, k require $\mathcal{O}(n)$, $\mathcal{O}(n^2)$ operations, respectively.
- Thus, altogether, the nested for-loop on lines 21-23 runs in $\mathcal{O}(n^2)$ time.
- Finally, we return total on line 24; this operation is $O(1)$.

The run-time of **example3** is the sum of the run-times of the above steps; as a quadratic function is asymptotically greater than a constant or linear function, the function has running time of order $\boxed{\mathcal{O}(n^2)}$. ■

Problem R-3.27
Give big-Oh characterization, in terms of n, of the running time of the example5 function in the code fragment below.

Solution:



```
36 def example5(A, B):              # assume that A and B have equal length
37   """Return the number of elements in B equal to the sum of prefix sums in A."""
38   n = len(A)
39   count = 0
40   for i in range(n):             # loop from 0 to n-1
41     total = 0
42     for j in range(n):           # loop from 0 to n-1
43       for k in range(1+j):       # loop from 0 to j
44         total += A[k]
45     if B[i] == total:
46       count += 1
47   return count
```

Figure 3: Function example3

Algorithm analysis:

- On line 36, we define the function, **example5**, that returns the number of elements in B sum of the prefix sums of S.
- The statements, n = len(S) and count = 0 on lines 38 and 39, execute in constant, $\mathcal{O}(1)$, time.
- Note that inside the outermost for-loop on line 40, lines 41-44 run in $\mathcal{O}(n^2)$ time (from function example3)
- Then lines 45-46 check whether B[i] == total, and, if so, increments count; this require constant operations for $\mathcal{O}(1)$ running time.
- Thus, the code inside the loop from $i = 0, ..., n-1$ requires $\mathcal{O}(n^2)$ operations. As it is run n times, the loop on lines 40-46 altogether will have $\mathcal{O}(n^3)$ running time.
- Finally, we return count on line 47; this operation is $O(1)$.

The run-time of **example5** is the sum of the run-times of the above steps; as the run-time is a polynomial in n with greatest degree 3, the function has running time of complexity $\boxed{\mathcal{O}(n^3)}$. ■

Problem R-3.33
Al and Bob are arguing about their algorithms. Al claims his $\mathcal{O}(n \log n)$-time algorithm is *always* faster than Bob's $\mathcal{O}(n^2)$-time method. To settle the issue, they perform a set of experiments. To Al's dismay, they find that if $n < 100$, the $\mathcal{O}(n^2)$-time algorithm runs faster, and only when $n \geq 100$ is the $\mathcal{O}(n \log n)$-time one better. Explain how this is possible.

Solution:

By the definition of big-Oh notation, (and assuming that Al and Bob used accurate, simplest terms to characterize their algorithms*), Al's function will run proportional to $c_1 * n^2$ time, for some constant c_1. Bob's function will run proportional to $c_2 * n \log n$ time, for some constant c_2.

An $\mathcal{O}(n \log n)$-time algorithm is asymptotically faster than an $\mathcal{O}(n^2)$-time algorithm; ie. Al's algorithm will require less time than Bob's when $n > n_0$ for some n_0. This explains how Al's algorithm will be faster than Bob's algorithm for all $n \geq 100$.

However, constant factors may cause Al's algorithm to be slower than Bob's for $n < 100$. For example, if Al's algorithm requires $50n \log n$ operations while Bob's only requires n^2, then when $2 < n < 100$, $\frac{n}{\log n} < 50 \equiv n^2 < 50n \log n$, and so Bob's algorithm may run faster.

Thus, it's possible for Al's $\mathcal{O}(n \log n)$ algorithm to be faster than Bob's $\mathcal{O}(n^2)$ algorithm only when $n \geq 100$. ■

(*Note: If Al or Bob did not use the most accurate big-Oh notation, then it is possible that the algorithms have different asymptotic growth rates than the ones assumed above. In this case, their findings may still result from Al's solution being asymptotically faster, but with constant factors that produce slower results for $n < 100$.)