- This lab will focus on more asymptotic analysis and additional searching algorithms.
- It is assumed that you have reviewed chapters 1 and 2 of the textbook. You may want to refer to the text and your lecture notes during lab as you solve the problems.
- When approaching the problems, think before you code. Doing so is good practice and can help you lay out possible solutions.
- Think of any possible test cases that can potentially cause your solution to fail!
- You must stay for the duration of the lab. If you finish early, you may help other students. If you don't finish by the end of the lab, we recommend you complete it on your own time.
- Your TAs are available to answer questions in lab, during office hours, and on Piazza

---

Vitamins (maximum 30 minutes)

---

1. For each of the following $f(n)$, write out the summation results, and provide a tight bound, $\Theta(f(n))$, using the Θ *notation*.

   Given n numbers:

   $1 + 1 + 1 + 1 + 1 \ldots + 1 =$ _____ $= \Theta($ _____ $)$

   $n + n + n + n + n \ldots + n =$ _____ $= \Theta($ _____ $)$

   $1 + 2 + 3 + 4 + 5 \ldots + n =$ _____ $= \Theta($ _____ $)$

   Given $log(n)$ numbers, where *n* is a power of *2*:

   $1 + 2 + 4 + 8 + 16 \ldots + n =$ _____ $= \Theta($ _____ $)$

   $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} \ldots + 1 =$ _____ $= \Theta($ _____ $)$

2. For each of the following code snippets, find $f(n)$ for which the algorithm's time complexity is $\Theta(f(n))$ in its **worst case** run and explain why.

a)
```
def func(lst):
    for i in range(len(lst)):
        if (lst[i] % 2 == 0):
            print("Found an even number!")
            return
```

b)
```
def func(lst):
    for i in range(len(lst)):
        if (lst[i] % 2 == 0):
            print("Found an even number!")
        else:
            print("No luck.")
            return
```

c)
```
def func(lst):
    for i in range(0, len(lst), 2):
        for j in range(i):
            print(lst[j], end = " ")
```

d)
```
def func(n):
    for i in range(n):
        j = 1
        while j <= 80:
            print("i = ",i,", j =", j)
            j *= 2
```

e)
```
def func(n):
    for i in range(n):
        j = 1
        while j <= n:
            print("i = ",i,", j =", j)
            j *= 2
```

Coding

In this section, it is strongly recommended that you solve the problem on paper before writing code.

1. Write a function to convert a string containing only digits to an int and returns it. Your solution must run in $\Theta(n)$, where n is the number of digits in the string. You may only use the `int( )` to convert a single digit to int such as `"1"` to `1` but not the entire string.

   ex.) `str_to_int("1134")` returns `1134`

```
def str_to_int(int_str):
"""
: int_str type: string
: return type: int
"""
```

2. Write a function that takes in a string as input and returns a new string with its vowels reversed. For example, an input of “tandon” would return “tondan”. Your function must run in $\Theta(n)$ and you may assume all strings will only contain lowercase characters.

   Hint: you may want to use:
   1. The **list constructor** to convert the string into a list in linear time.
   2. The **.join() string method**, which is guaranteed to run in linear time.
      The join() method is a string method that can take in a list of string values and returns a string concatenation of the list elements joined by a str separator.
      For example: `",".join(["a", "b", "c"])` will return `"a,b,c"`.

```
def reverse_vowels(input_str):
"""
: input_str type: string
: return type: string
"""

    list_str = list(input_str) #list constructor guarantees
Theta(n)
```

3.

For this question, you will write a new searching algorithm, *jump search*, that will search for a value in a **sorted list**. With jump search, you will separate your list into n/k groups of k elements each. It then finds the group where the element you are looking for should be in, and makes a linear search in this group only.

For example, let's say k = 4 for the following list of n = 20 elements:

[1, 3, 6, 7 , 10, 12, 15, 20 , 22, 24, 29, 33 , 39, 55, 61, 64 , 99, 101, 134, 150]

Here, we have a list of n/k = 20/4 = 5 groups. If we were to check for val = 15, we would start with the first element (index 0) of the first group and check if we've found our value.

Since, $1 \; ! = 15 \;\; and \; 1 \; < \; 15$, we will jump k = 4 elements and move to 10 (at index 4).

[1, 3, 6, 7 , 10, 12, 15, 20 , 22, 24, 29, 33 , 39, 55, 61, 64 , 99, 101, 134, 150]

Since $10 \; ! = \; 15 \; and \; 10 \; < \; 15$, we jump another k = 4 steps and move to 22 (at index 8).

[1, 3, 6, 7 , 10, 12, 15, 20 , 22, 24, 29, 33 , 39, 55, 61, 64 , 99, 101, 134, 150]

Now, we have $22 \; ! = \; 15 \; and \; 22 \; > \; 15$, so we don't need to jump further. Instead, we will hop back k elements because we know our val has to be somewhere between 10 and 22, (index 4 and index 8). We jump back up to k = 4 elements until we find our val = 15.

[1, 3, 6, 7 , 10, 12, 15, 20 , 22, 24, 29, 33 , 39, 55, 61, 64 , 99, 101, 134, 150]
← ← ←

With the sample list above, we jump from 1 to 10, and 10 to 22. Then we linear search back between 22 and 10 and found 15.

Recap:

1. Divide the list into `n/k` groups of `k` elements.
2. Jump `k` steps each time until either `val == lst[i]` or `val < lst[i]`.
3. if `val == lst[i]`, return the index i.
4. if `val < lst[i]`, jump back one step each time for a maximum of `k` steps.
5. If `val` is somewhere in those `k` steps, return the index.
6. Otherwise, return `None`.

3a.

Write a function that performs jump search on a sorted list, given an additional parameter, k. This parameter will let the user divide the list into k groups for the search.

Consider some edge cases such as if n isn't divisible by k (there will be a group with fewer than k elements) or if val > all elements in the list?

Analyze the worst case run-time of your function in terms of n, the size of the list, and k :

```
def jump_search(lst, val, k):
"""
: lst type: list[int]
: val, k type: int
: return type: int (if found), None(if not found)
"""
```

3b.

Let's now optimize our jump search algorithm by defining what our k value should always be. That is, we will no longer have k be passed in as a parameter.

Hint: The jump search algorithm is actually slower than binary search but faster than linear search. You may use functions from the math library for this algorithm.

Analyze the run-time of jump search in terms of n, the length of the list:

```
def jump_search(lst, val):
"""
: lst type: list[int]
: val type: int
: return type: int (if found), None(if not found)
"""
```

4. In class, you learned about *binary search*, which has a run-time of $O(log(n))$ for searching through a **sorted list**. With binary search, the lower bound begins at index 0 while the upper bound is the last index, len(list) - 1.

   You will write a modification of the binary search called, *exponential search* (also called doubling or galloping search). With exponential search, we start at index i = 1 (after checking index 0) and we double i until it becomes the upper bound.

   Let's try to find `15` in the given sample list by checking index `i = 0` first:

   `i = 0, (lst[0] = 1) != (val = 15):`
   [1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

   From there, you will make a comparison to see if the value is at index, i = 1. If not, you will double i until the index is out of range or until the value at the index is larger than the value you're searching for. **It is important for the two conditions to be in that order!**

   `i = 1, (lst[i] = 3) < (val = 15):`
   [1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

   `i = 2, (lst[i] = 6) < (val = 15):`
   [1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

   `i = 4, (lst[i] = 10) < (val = 15):`
   [1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

   `i = 8, (lst[i] = 22) > (val = 15):`
   [1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

   If the value is less than i or is out of bound, then you know it has to be between `i//2` and `i` (or `i//2` and `len(list)-1`). After confirming your bounds, you perform a binary search (in that range only).

   `(lst[4] = 10) < val < (lst[8] = 22):`
   [1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

   With the sample list above, we confirm that val must be between i = 4 and i = 8.

Recap:

1. Start with `i = 0` and check if `val == lst[0]` (if list is not empty).
2. If `val != lst[0]`, start the exponential search (doubling process) at `i = 1`.
3. If `val != lst[1]`, check if `val > lst[i]`. If so, double `i`.
4. Keep doubling while `val != lst[i] and val < lst[i]`.
5. Confirm that lower bound is `i//2` and upper bound is `max(i, len(lst))`
6. Perform binary search between the lower and upper bounds.

What is the run-time for *exponential search* and what advantage might this algorithm have over binary search?

```
def exponential_search(lst, val):
"""
: lst type: list[int]
: val type: int
: return type: int(if found), None(if not found)

"""

     if len(lst) > 0:           #check if list is not empty
          if lst[0] == val:     #check index 0 first
               return 0
          else:
               i = 1 #start at 1 for the exponential search
               ...
     return None
```

Here is a simple test code to verify that your searching algorithm works.

```
#TEST CODE

lst = [-1111, -646, -818, -50, -25, -3, 0, 1, 2, 11, 33, 45, 46, 51,
58, 72, 74, 75, 99, 110, 120, 121, 345, 400, 500, 999, 1000, 1114,
1134, 10010, 500000, 999999]

lst2 = [-val for val in lst if val != 0]

print("TESTING VALUES IN LIST:\n")

for val in lst:
     if exponential_search(lst, val) is None:
          print(val, "FAILED")
     #else:
          #print(val, "PASSED")


print("TESTING VALUES NOT IN LIST:\n")

for val in lst2:
     if exponential_search(lst, val) is not None:
          print(val, "FAILED")
     #else:
          #print(val, "PASSED")


#just to make sure you’re not stuck in an infinite loop
print("TEST COMPLETED")
```

EXTRA

Implement a function that calculates an approximation of the square root of a number with two-decimal points accuracy that runs in $\Theta(\sqrt{n})$.

```
def square_root(num):
    """
    : num type: positive int
    : return type: float
    """
```

Implement the square root function that runs in $\Theta(\log(n))$.