Q1(a)+(b)

Write a function with the signature `insert(L, e)` which takes in a list of `floats` `L`, which is sorted in non-decreasing order, and returns a new list which is also sorted in non-decreasing order, and contains all the elements of `L` as well as the float `e`. Here are some examples:

```
insert([3.0, 4.0, 5.0], 3.5) should return [3.0, 3.5, 4.0, 5.0].
insert([2.0, 5.0], 7.0) should return [2.0, 5.0, 7.0].
insert([], 42.0) should return [42.0].
```

In [1]:
```python
# Complexity: O(n log n)
def insert(L, e):
    return sorted(L + [e])
```

In [2]:
```python
# Complexity: O(n)
def insert(L, e):
    i = 0

    while L[i] < e and i < len(L):
        i += 1

    L.insert(e, i)
```

In [3]:
```python
# Complexity: O(n)
def insert(L, e):
    i = 0

    while i < len(L) and L[i] < e :
        i += 1

    copyL = L[:] #not necessary
    copyL.insert(i, e)

    return copyL
```

In [4]:
```python
# Complexity: O(n)
def insert(L, e):
    copyL = [e] + L
    for i in range(len(copyL)-1):
        if copyL[i+1] < copyL[i]:
            copyL[i], copyL[i+1] = copyL[i+1], copyL[i]
        else:
            return copyL
    return copyL
```

Marking scheme:

- 12 for correct solution, no matter how obtained
- 6 points for finding the right place to insert, 6 points for inserting. Award part marks for good attempts:
  - 2/6: the person knows the right Python construct but not how to use it
  - 4/6: the person knows how to use the construct, but didn't manage to
  - 5/6: minor mistake
- 3 points for correct complexity, no points for incoreect complexity (unless there are very special circumstances.)

Q2

Santa wants to select gifts for an EngSci student. Santa has two dictionaries: a dictionary that records the rating of how good a gift would be for the student (on a scale of 1-5), and a dictionary that records the rating of how much the student wants the gift (on a scale of 1-5). The rating of a gift which is not in either of the dictionaries is considered to be 0. Santa wants to select the gifts with the maximal possible combined rating, where the combined rating is the sum of the rating of how good the gift would be for the student and the rating of how much the student wants the gift. For example, the dictionaries can be:

```
good_ratings = {"Calc textbook": 5, "iPhone": 1, "Alarm clock": 4, "Notebooks": 4}
want_ratings = {"iPhone": 4, "A+ in CSC": 5, "Calc textbook": 4, "Notebooks": 5}
```

Here, the gifts Santa wants to select are `"Calc textbook"` and `"Notebooks"`, since the combined rating for them is $5+4=9$, larger than any other one. The combined rating of `"Alarm clock"` is $4+0=4$.

Write a function with the signature `select_gifts(good_ratings, want_ratings)` that returns a list of all the gifts which have the highest combined rating of all the gifts, sorted in alphabetical order. For example, for `good_ratings` and `want_ratings` as defined above, `select_gifts(good_ratings, want_ratings)` should return `["Calc textbook", "Notebooks"]`.

In [5]:
```python
def select_gifts(good_ratings, want_ratings):

    combined_ratings = {}

    for gift in good_ratings:
        combined_ratings[gift] = good_ratings.get(gift, 0) + want_ratings.get(gift, 0)

    for gift in want_ratings:
        combined_ratings[gift] = good_ratings.get(gift, 0) + want_ratings.get(gift, 0)

    max_combined = max(combined_ratings.values())

    res = []

    for gift, rating in combined_ratings.items():
        if rating == max_combined:
            res.append(gift)

    return res
```

Marking scheme:

- 8 points for computing the combined ratings (5 general + 3 for dealing with 0's)
  - 2/5 (general): The person knows the correct Python construct but not how to use it
  - 3/5 (general): The person is on the right track, but made a major mistake
  - 4/5 (general): minor mistake
- 4 points for computing the max ratings
  - 1/4 for an attempt to compute the max of something
  - 2/4 major mistake
  - 3/4 minor mistake
- 3 points for getting the resultant list.

Q3

In Python, you can use a *list of lists* to store a matrix, with each inner list representing a row. For example, you can store the matrix

$$\begin{pmatrix} 5 & 6 & 7 \\ 0 & -3 & 5 \end{pmatrix}$$

by storing each row as a list: `M = [[5,  6, 7], [0, -3, 5]]`.

Complete the following function. The function takes in a matrix `M` in a list-of-lists format. The matrix returns the transposed version of `M`, in a list-of-lists format. For example,

`transpose([[5,  6, 7], [0, -3, 5]])` should return `[[5, 0], [6, -3], [7, 5]]`.

In [6]:
```python
def transpose1(M):
    res = []
    for ncol in range(len(M[0])):
        row = []
        for nrow in range(len(M)):
            row.append(M[nrow][ncol])
        res.append(row)
    return(res)


def transpose2(M):
    width = len(M[0])
    height = len(M)

    res = []
    for i in range(width):
        res.append([0]*height)

    for i in range(height):
        for j in range(width):
            res[j][i] = M[i][j]

    return res
```

Marking scheme

- 5 pts for the idea that we need to use `M[i][j]` to update `res[j][i]` somehow.
  - Not many opportunities for part marks, but watch for people with other ideas, and make sure to mention the ideas to figure out how to grade those
- 3 pts for looping through every entry of `M`
  - Not many opportunities for part marks, probably
- 7 pts for implementation
  - 2/7: a very flawed attempt: the person didn't come up with either the idea of appending row-by-row, or the idea of initializing the matrix and then filling it in
  - 5/7: an $\mathrm{ncols} \times \mathrm{nrows}$ matrix is being built, but there is a significant mistake somewhere
  - 6/7: a minor mistake

Q4

Write a recursive function with the signature `max_rec(L)` which takes in a list of `ints` `L`, and returns the largest element in the list. You may not use loops, global variables, or Python's `max()`, `sorted()`, and `sort()` functions. You may use slicing.

For example, `max_rec([103, 180, 101, 102, 180])` should return `180`.

In [7]:
```python
def max_rec(L):
    if len(L) == 1:
        return L[1]

    res1 = max_rec(L[1:])

    if res1 > L[0]:
        return res1
    else:
        return L[0]
```

Marking scheme

- 2 pts for the base case
- 2 pts for the recursive step
- 1 pt for computing max(res1, L[0])

Q5

Write a recursive function with the signature `is_fib(L)` which takes in a list of `ints` `L`, and returns `True` if `L` is the start of the Fibonacci sequence, and `False` otherwise. For example:

```
is_fib([1, 1, 2, 3, 5]) should return True.
is_fib([1, 1, 2, 3, 5, 8, 13]) should return True.
is_fib([5, 8, 13]) should return False.
is_fib([1, 1, 1]) should return False.
is_fib([]) should return True.
```

You may not use helper functions, loops or global variables. You may use slicing. Reminder: $fib(n+2) = fib(n+1) + fib(n), fib(1) = fib(2) = 1$.

In [8]:
```python
def is_fib(L):
    if len(L) == 0:
        return True
    if len(L) == 1:
        return L == [1]
    if len(L) == 2:
        return L == [1, 1]

    return L[-1] == L[-2] + L[-3] and is_fib(L[:-1])
```

Marking scheme

- Base case: 5 pts (2 pts for something, 3 pts for lengths of 0, 1, 2)
- Recursive step: 10 pts
  - At most 2/10 if a non-working attempt is made to go from the front of the list to the back (and not as in the solution)

Q6(a)

In [9]:
```python
A = [[1, 2], [3, 4]]
A[0] = A[1]
B = A[:][0]
B[0] = 5
print(A)
```

```
[[5, 4], [5, 4]]
```

Q6(b)

In [10]:
```python
def f():
    L[0] = 5

L = [1, 2]
print(f(L))
print(L)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-01fe5e8f91bd> in <module>()
      3
      4 L = [1, 2]
----> 5 print(f(L))
      6 print(L)

TypeError: f() takes no arguments (1 given)
```

- Marking scheme: 1 pt for `[5, 2]`

Q6(c)

In [11]:
```python
def f(L, M):
    L = M
    L[0] = 3

M = [1, 2]
L = [3, 4]
f(L, M)
print(M[0])
```

```
3
```

Q6(d)

In [12]:
```python
s1 = "HO HO HO"
s2 = s1
s1 = "Happy Holidays!"
print(s2)
```

```
HO HO HO
```

Q7(a)

In [13]:
```python
n = 0
#O(n^2)
total, i = 0.0, 0
for i in range(n):
    for j in range(i//2):
        total += i
```

In [14]:
```python
#O(n^2)
i, j, sum = 1, 1, 0
while i < n**3:
    while j < n:
        sum = sum + i
        j += 1
    i += n
```

In [15]:
```python
#O(n)
def f(n):
    if n == 0:
        return 1
    return f(n//2) + f(n//2)

if __name__ == "__main__":
    f(n)
```

In [16]:
```python
#O(1)
def f(n):
    i, total = 0, 0.0
    while (i < n) and ((i % 10000) != 0):
        total += i
        i += 1

if __name__ == "__main__":
    f(n)
```

Q8

In [17]:
```python
def mystery_helper(L, k):
    p = max(L[0], L[-1])
    L1 = []
    L2 = []
    for e in L:
        if e < p:
            L1.append(e)
        else:
            L2.append(e)

    if len(L1) > k:
        return mystery_helper(L1, k)
    elif len(L1) < k:
        return mystery_leper(L2, k-len(L1))
    else:
        return p

def mystery(L):
    return mystery_helper(L, len(L)//2)
```

Q8(a)

State clearly and concisely what mystery_helper(L, k) returns.

Answer: the (k+1)-st smallest element of L

Marking scheme

- 3/4 for k-th smallest, (k+1)-st largest, etc.

Q8(b)

What is the tight asymptotic upper bound on the worst-case runtime complexity of `mystery(L)`, where `n = len(L)`? Use Big O notation. Explain how you got your answer to this subquestion. You may assume that `L` is a list of `floats`.

Answer:

In the worst case, we keep calling `mystery(L1)`, with `L1` only getting shorter by one element every time.

$n + (n-1) + (n-2) + \ldots + (n-k)$ is $O(n^2)$ for $k = n/2$

Also acceptable: infinite loop for `[1, 1, 1, 1, 1]`

Marking scheme

- No marks for wrong complexity
- No explanation => 1/3 for $O(n^2)$
- No mention of worst case w/ something reasonable about it => 1/3
- Somewhat sensible discussion of the worst-case + some sensible calculuation of $O(n^2)$ => 3/3

Q9

A timestamp is a tuple consisting of two integers, with the first one denoting the hour in the day (between 0 and 23), and the second one denoting the minute (between 0 and 59). The timestamp (5, 10) corresponds to 5:10AM, the timestamp (13, 25) corresponds to 1:25PM, and so on. Write a function with the signature sorted_timestamps(timestamps) that takes in a list of timestamps, and returns a sorted version of that list, with the sorting done from earlier to later timestamps. The function must run in O(n) time, where n = len(timestamps). For example,

```
sorted_timestamps([(5, 10), (2, 40), (22, 59), (5, 10)])
```

should return `[(2, 40), (5, 10), (5, 10), (22, 59)]`

In [18]:
```python
def sorted_timestamps(timestamps):
    counts = [0]*60*24
    for t in timestamps:
        counts[t[0]*60+t[1]] += 1

    res = []
    for m in range(60*24):
        res.extend( [(m//60, m%60)]*counts[m])

    return res
```

Marking scheme

- 3 points for bucketsort idea (if progress made)
- 2 points for the hash function
- 2 points for figuring out how to convert minutes (or whatever the hashfunction value is) back to timestamp
- 3 points for general implementation

Q10

We can use a dictionary to record who is friends with whom by recording the lists of friends in a dictionary.

For example:

```
friends = {"Carl Gauss": ["Isaac Newton", "Gottfried Leibniz", "Charles Babbage"],
           "Gottfried Leibniz": ["Carl Gauss"],
           "Isaac Newton": ["Carl Gauss", "Charles Babbage"],
           "Ada Lovelace": ["Charles Babbage", "Michael Faraday"],
           "Charles Babbage": ["Isaac Newton", "Carl Gauss", "Ada Lovelace"],
           "Michael Faraday"" ["Ada Lovelace"] }
```

Here, Carl Gauss is friends with Isaac Newton, Gottfried Leibniz, and Charles Babbage. Assume that friendships are symmetric, so that if X is friends with Y, then it's guaranteed that Y is friends with X. A clique is defined as a group of friends where everyone is friends with everyone. For example, Carl Gauss, Isaac Newton, and Charles Babbage form a clique in the example above, since all three are friends with each other. Ada Lovelace and Michael Faraday also form a clique. Write the function max_clique(friends), which takes in a dictionary in the format above, and returns the largest clique that can be found, as a list. (If there are several such cliques, return one of them.) For example, the largest clique in the example above is `["Carl Gauss", "Isaac Newton", "Charles Babbage"]`, since there is no clique of size larger than 3.

In [19]:
```python
def is_clique(group, friends):
    for f in group:
        for f1 in group:
            if f != f1:
                if f1 not in friends[f]:
                    return False
    return True


def get_all_subsets(L):
    if len(L) == 0:
        return [[]]
    all0 = get_all_subsets(L[1:])
    res = []
    res.extend(all0)
    for subset in all0:
        res.append([L[0]] + subset)

    return res

def max_clique(friends):
    all_subsets = get_all_subsets(list(friends.keys()))

    max_sz = 0
    clique = []
    for subset in all_subsets:
        if is_clique(subset, friends):
            if len(subset) > max_sz:
                max_sz = len(subset)
                clique = subset[:]
    return clique
```

In [20]:
```python
 friends = {"Carl Gauss": ["Isaac Newton", "Gottfried Leibniz", "Charles Babbage"],
           "Gottfried Leibniz": ["Carl Gauss"],
           "Isaac Newton": ["Carl Gauss", "Charles Babbage"],
           "Ada Lovelace": ["Charles Babbage", "Michael Faraday"],
           "Charles Babbage": ["Isaac Newton", "Carl Gauss", "Ada Lovelace"],
           "Michael Faraday": ["Ada Lovelace"] }
max_clique(friends)
```

Out[20]:
```
['Isaac Newton', 'Carl Gauss', 'Charles Babbage']
```

Marking scheme

- 2 points for idea of enumerate + check (with an attempt to implement)
- 5 points for enumerate