CS220 A4

name: Ray Zhao

upi: lzha641

Problem 1

(a)

function **erase(a,i)**:

```
Function erase(a, i):
    temp ← a[i]
    for index ← i to n-1 do:
        a[i] ← a[i+1]
    a.removelast()
    return temp
```

How it works:

All the items between index **i** (index of element we want to remove) and the last element in the list (index **n-1**) get shifted to the left by 1 place; the current index will now contain the element of the next index. This results in the index i getting replaced by the next item and all the elements after i stay the same. Providing the erase functionality. Then we return the item we have removed.

actual implementation using python:

```python
def erase(a, i):
    temp = a[i]

    for index in range(i, len(a)-1):
        a[index] = a[index+1]

    a.pop()
    return temp
```

function **insert(a, j, t)**:

```
Function insert(a, j, t):
    temp = a.getLast()
    for index ← n-1 to j do:
        a[index] ← a[index - 1]
    a.add(temp)
    a[j] = t
```

shifts element from index ‘j’ to the end of the list to the right to make space for the new element to be inserted. Adds the last element stored in *temp* to the list and insert the element - t to the list at index j.

actual implementation using python:

```
def insert(a, j, t):
    last = a[len(a)-1]

    for index in range(len(a)-1, j, -1):
        a[index] = a[index-1]

    a.append(last)
    a[j] = t
```

(b)

In the case of our algorithm, memory write operation only occurs in erase() and insert() functions where the elements inside the list are being modified as erase() removes the element at index i and insert() adds an element at index j.

Therefore, the best case for the least number of memory write operations would be 0 when the list is already sorted.
example: [1,2,3,4,5,6]

as we go through the list that is already sorted in an increasing order, a[i] will always strictly be greater than a[j]:

$a[i] > a[j]$

so, this line of code will never run and j will not decrement by 1

```
while $j \geq 1$ and $a[i] < a[j]$ do
    $j \leftarrow j - 1$
```

which means the erase() and insert() functions will never be called:

```
if $j < i$ then
    $t \leftarrow$ erase($a, i$)
    insert($a, j, t$)
```

Therefore, we can conclude that there will be 0 memory operations happening when the list is sorted.

(c)

The worst case for the number of memory write operations would happen when the list is sorted strictly in **decreasing order** (reversed). It will cause every element of list - **a[i]** to be inserted at the beginning of the sorted part of the list. This will cause a maximum number of calls to the **erase** and **insert** functions.

worst case analysis:

erase function will run for **n-i-1** times

> as it needs to go through the for loop to remove the element, and every element goes through 1 memory write operation. So, total memory write operations:
>
> n-i-1

insert function will run $n - 1 - j$ memory operations.

- each element is in the wrong position and needs to be inserted to the start of the list on every iteration, so *j = 0*:

  n-1

- 1 memory write operation where we need to insert element **t** to the list: a[j] ← t

  1
- n-1+1 = n memory writes for insert function worst case

This means we will have **erase + insert** memory write operations for every item in the list since every element in the list will call erase and insert functions when the list is reversed.

we can represent the sum of operations as:

$$T(n) = \sum_{i=1}^{n-1} [(n - i - 1) + (n - 1)]$$

$$T(n) = (2n - 2 - i)$$

$$T(n) = (2n - 2)(n - 1) - \frac{n(n-1)}{2}$$

$$T(n) = \frac{2(2n-2)(n-1)}{2} - \frac{n(n-1)}{2}$$

$$T(n) = (3n^2 - 7n + 4)/ 2$$

therefore:

$$T(n) = O(n^2)$$

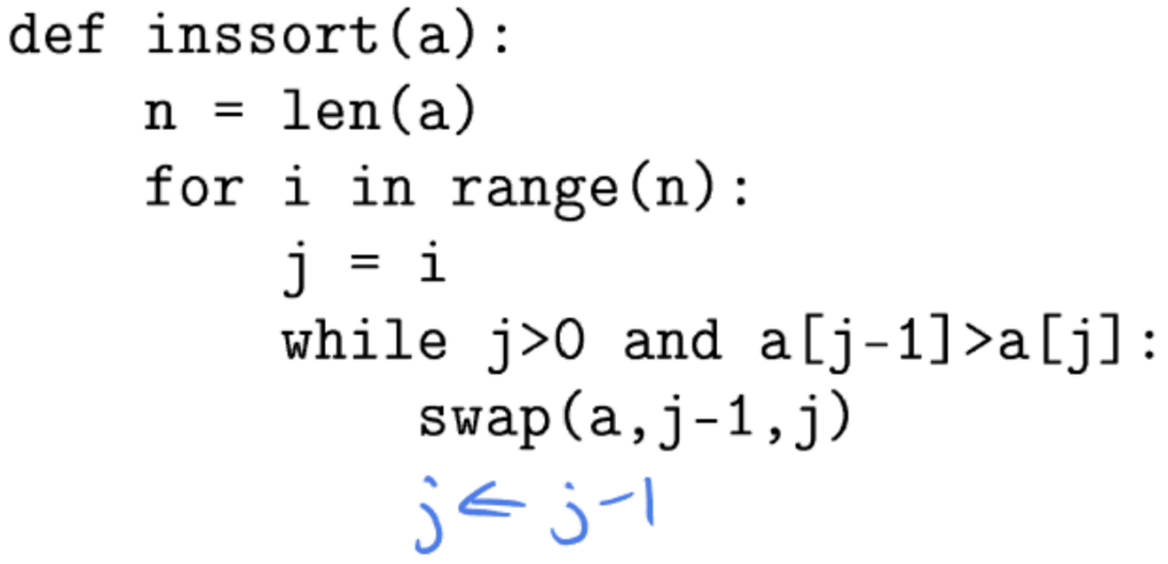
example where this is attained:

a = [7, 6, 5, 4, 3, 2, 1]

(d)

comparison of memory write operations with the standard insertion sort:

standard insertion sort:



```
def inssort(a):
    n = len(a)
    for i in range(n):
        j = i
        while j>0 and a[j-1]>a[j]:
            swap(a,j-1,j)
            j ⇐ j-1
```

best case:

0 memory write operations as there is no swapping happening.

worst case:

- 2 memory write operations for every swap action:

  swap(a, j-1, j)

- every element has a maximum number of swaps of **i-1**, where i is the index of the current element.

  so, we can represent the total number of memory write operations as:

  $$2 * \sum_{i=2}^{n} (i - 1)$$

  $$= 2 * \frac{(n-1) * n}{2}$$

  $$= n(n - 1)$$

  $$= n^2 - n$$

  So, the standard insertion sort will have

  $O(n^2)$

  for memory write operations.

our implementation of insertion sort:

- makes 0 on best case
- makes $O(n^2)$ on worst case (as discussed in b)

The best cases are the same (both 0) for the 2 insertion sort implementations.

By comparing the worst cases:

n^2 *is* $\Theta(n^2)$

Therefore the standard insertion sort is *asymptotically equal* to the insertion sort algorithm given in this question.

Problem 2

Assumption:

I have assumed all the inputs - n are powers of 2 implied by the question. T(1)=1 is a different case that does not follow this recurrence.

The solution to this recurrence relation is:

T(1) = 1

$T(n) = \frac{n}{2}$ for n >= 2

I have guessed the solution by using the bottom up method to list out some inputs of n that are powers of 2.

Bottom up method:

T(1) = 1

T(2) = T(2/2) = 1

T(4) = T(4/4) + T(4/2) = 2

T(8) = T(8/2) + T(8/4) + T(8/8) = 4

T(16) = T(16/2) + T(16/4) + T(16/8) + T(16/16) = 8

T(32) = T(32/2) + T(32/4) + T(32/8) + T(32/16) + T(32/32) = 16

...

From the results, it is easy to see that T(n) is always equal to half of **n**.

Therefore, I think the solution is T(n) = n/2

Proof by induction

induction hypothesis:

$T(1) \ = \ 1$

$T(n) \ = \ \frac{n}{2}$ for n >= 2

is the solution to this recurrence.

base case:

n = 2.

By following the recurrence pattern, we can find that

T(2) = T(2/2)

= T(1)

= 1

Therefore, T(2) = 1.

$1 = 2/2$

Our base case holds true for T(n) = n / 2.

inductive case:

Assuming our solution is correct for predicting **n**, we want to prove that the next case after n; 2n is also true (because every input - n is a power of 2 implied by the question).

T(n) can be written as:

$$T(n) = T(n/2) + T(n/4) + T(n/8) + ... + T(1)$$

T(2n) can be written as:

$$T(2n) = T(2n/2) + T(2n/4) + T(2n/8) + ... + T(1)$$

$$= T(n) + T(n/2) + T(n/4) + T(n/8) ... + T(1)$$

we can see that T(2n) contains all the recursive components T(n) has and the only difference is that it has **1 extra T(n)** recursive call.

This means that:

$$T(2n) = 2T(n)$$

since we assumed T(n) = n/2,

$T(2n) \;=\; 2(n/2) \;=\; n$

$n \;=\; \frac{2n}{2}$

Since we assumed T(n) to be correct, we will get: T(2n) = n, which corresponds to our hypothesis T(n) = n/2, and has proven our induction hypothesis to be correct.

Problem 3

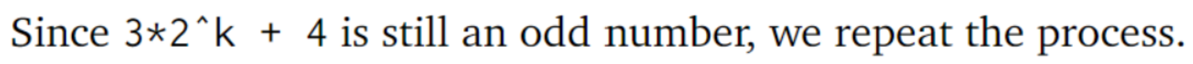
correct

- the LLM has correctly provided the base case:
  when n=1, T(1) = 0
  This is correct because the algorithm does return 0 when input n is 1 without any further recursive calls

- The LLM has also correctly suggested the recursive cases for the recurrence relation as it stated that the function will return T(n//2) when n is even,
  T(3n+1) when n is odd which are both correct.

errors

- the LLM incorrectly assumed that $n = 2^k$ is the **worst** case. Because when n is even and a power of 2, it simply cannot ever invoke the recursive call → T(3n + 1) because n will never become odd and always stay even

- the LLM has incorrectly given the solution for the recurrence for when n is odd:
  it claimed that
  
  Since 3*2^k + 4 is still an odd number, we repeat the process.
  But $3 * 2^k + 4$ is even. Because $3 * 2^k$ is even and 4 is also even. The sum is even. Therefore, we would not repeat the process contradicting what the LLM has suggested.

ways to fix errors

This problem is an unsolved mathematical problem known as the Collatz conjecture. So, there is no solution we can implement for this problem.