**Week 3 Studio**

**Assessed Preparation**

Problem 1:

def countSort(num\_list, exp, b):

    number\_of\_items = len(num\_list)

    output = [0] \* number\_of\_items

    item\_c = [0] \* (b)

    for items in range(0, number\_of\_items

        index = (num\_list[items] / exp)

        item\_c[int(index % b)] += 1

    for digit in range(1, b):

        item\_c[digit] += item\_c[digit - 1]

    i = number\_of\_items - 1

    while i >= 0:

        index = (num\_list[i] / exp)

        output[item\_c[int(index % b)] - 1] = num\_list[i]

        item\_c[int(index % b)] -= 1

        i -= 1

    i = 0

    for i in range(0, len(num\_list)):

        num\_list[i] = output[i]

    return num\_list

Problem 2:

Initial list Sorting on rightmost digit moving right to left

4329 4321 4321 2099 2099

5169 2121 2121 2121 2121

4321 -> 4329 -> 4329 -> 5169 -> 3369

3369 5169 5169 4321 4321

2121 3369 3369 4329 4329

2099 2099 2099 3369 5169

^ ^ ^ ^

1st digit 2nd digit 3rd digit 4th digit

d

Initial Array:

[baaa, aaab, aaaa, aaba, abab, abaa]

-> [baaa, aaaa, aaba, abaa, aaab, abab]

-> [baaa, aaaa, abaa, aaab, abab, aaba]

-> [baaa, aaaa, aaab, aaba, abaa, abab]

Text, letter

Description automatically generated

Invariant: At the start of the ith iteration, min will be the minimum element in subarray A[1...i-1]

Initialisation:

Upon initialisation, min is set to be A[1]. This satisfies the loop invariant as min is representing the minimum element in subarray A[1], which is the only element thus far.

Maintenance:

On the kth iteration of the loop, lets assume that min represents the minimum element in the subarray A[1...k-1], where k is some value 1 <= k. Upon the next iteration of the loop, this algorithm checks if the element A[k] < min, and updates the minimum to be A[k] if this is true. Otherwise, minimum is still the minimum element in the array A[1..k] and there is no need to update it. Thus, by induction, the loop invariant will be satisfied for the k+1th iteration and so forth.

Termination:

At the termination of this algorithm, i = n the size of the array. Yada yada

At initialisation, the array contains the minimum value in the array, i.e. A[1]

Invariant: On the ith iteration of the algorithm the min will be updated to the minimum item in the sub array A[1] to A[i]

Invariant: min is always the minimum element at the start of the ith iteration in the sub array A[1..i-1].

Initialisation:

Min is set to A[1] which satisfies the loop invariant as the current min is the minimum element in the sub array A[1…(2-1)] which is A[1]

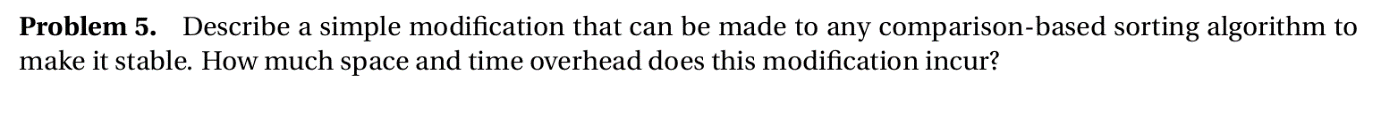
Base case:

* If A has one element that element is the minimum element in the array as there is no other element to compare to
  + Therefore basecase holds true

Inductive step:

* Assume that the algorithm holds true for the sub array of A[1...n] up to the kth element (MINIMUM\_ELEMENT[1...k] holds true)
* On the k+1t iteration the algorithm will either update min to the k+1th element if the element is less than min or not update min if it is not.
  + Hence min will always be the minimum item of the subarray A[1...k+1]

-> Hence this algorithm is correct



Having an additional array to keep track of the positioning of the elements in the original array can help ensure stability as the elements are compared in the order that they appear in the original array. This requires O(n)

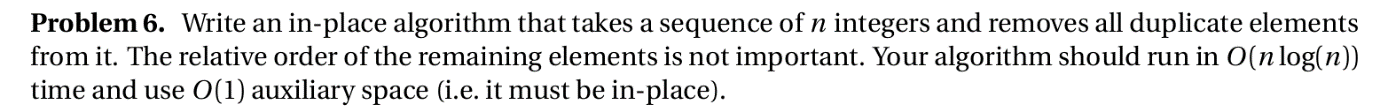
[(1,0),(5,1),(3,3),(4,2),(4,4)]

X = (1,0) y = (3,3)

x = (4,2) y = (4,4)

Expression:

X[0] < y[0] or X[0] ==y[0] and x[1] < y[1]



HeapSort(arr)

j = 1

for i = 2 to n

If A[i] == A[i-1]

    i ++

Else

    A[j+1] = A[i]

    i ++

    j++

for k = n to j+1

    pop(A)

return arr

Sort with unique elements at the start of the list. Pop a duplicate element subarray?

A[unique] .. [duplicate]

J = 1

I = 2

While i <= n:

    If A[i] != A[i-1]:

        A[i-1] = A[i]

        J++

        i++

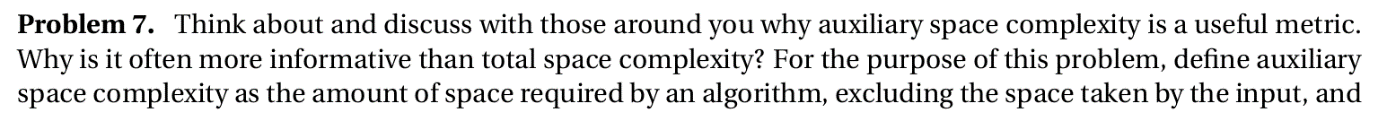
    if A[i] == A[i-1]:

        i++

For i = 1:n-j:

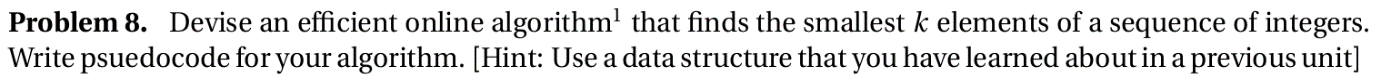
    pop(array)

Return array





Auxiliary space complexity is a useful measure as it effectively measures how much extra space a program takes up aside from the input. In reality, programs are called with the arguments already allocated in memory, so that space is a given in most cases and the important part is how much space the program uses after being called.



heap?

k\_find(array, k):

heapify(lst)  # min heap where root is smallest

removemin() k times to get the smallest k elements

Text

Description automatically generated

def bin\_search(A,key):

    count = A.length

    lo = A[0]

    hi = A[count - 1]

    while lo < (hi - 1):

        mid = hi+lo/2

        if key > A[mid]:

            lo = mid

        else:

            hi = mid

        if A[hi] == key:

            return hi

Text

Description automatically generated

a)

# my brain is fried +1

merge(~~a1,a2..,an~~A)

K = len(A)

curr1 = a1[1] # might need something to keep track of other lists head(curent) as well

curr2 = a2[1]

Current = [0]\*k

Etc

Output = []

For i in range(k\*n):

Min = min(curr1, curr2, …, curr\_k)

output.insert(min)

If curr\_i is the min, increment curr\_i = ai[++]

b)

Merge them all at once? heap?

c)

No, I think O(nlog(k)) is the fastest