Kyle Nealy

10/21/15

C343

HW 4

CH12 Exercises

--------------------------------------------------------------------------------------------------------

R-12.7 Suppose we are given two n-element sorted sequences A and B each with distinct elements, but potentially some elements that are in both sequences. Describe an O(n)-time method for computing a sequence representing the union A ∪ B (with no duplicates) as a sorted sequence.

Algorithm for representing the union of two sorted sequences should do the following:

1.) Create a list (array) and counter (integer) for iterating over the list.

2.) Create index variables to iterate over the given sequences.

3.) Iterate over each sequence until we reach the end of both.

Here is a simple python implementation:

def seq\_union(A,B):

r = []

cnt = -1

a=0

b=0

while a < len(A) and b < len(B):

if r:

if a < len(A) and r[-1] == A[a]:

a+=1

continue

if b < len(B) and r[-1] == B[b]:

b+=1

continue

# when a runs out of items, just insert B:

if a >= len(A):

r.append(B[b])

elif b >= len(B):

r.append(A[a])

elif A[a] < B[b]:

r.append(A[a])

else:

r.append(B[b])

return r

R-12.13 If the conditional at line 14 of our in place quick sort implementation of Code Fragment 12.6 were changed to use condition left < right (rather than left <= right), there would be a flaw. Explain the flaw and give a specific input sequence on which such an implementation fails.

If the conditional at line 14 were changed to use the condition left<right, the algorithm will ignore the conditional for elements to the left of the pivot that are equal to the value of the right marker, and won't swap the markers when they are equal to one another. As a result, the algorithm won't be able to reach the conditions necessary to break the while loop, and continues to run infinitely on sequences containing duplicates. For instance, if the array passed to the function is something like [1,3,4,5,6,5,7,8,2,3,5], the function would never return a value and just keep running until a Python Runtime Error occurs.

R-12.17 Is the bucket-sort algorithm in-place? Why or why not?

No, the bucket-sort algorithm is not an in-place algorithm. In a bucket-sorting algorithm, extra storage space, or memory, is allotted to create smaller arrays (buckets) in which the elements are sorted before being combined into another array of size n that replaces the original sequence.

R-12.19 Suppose S is a sequence of n values, each equal to 0 or 1. How long will it take to sort S with the merge-sort algorithm? What about quick-sort?

It will take O(n log n) time with the merge-sort algorithm, because the algorithm doesn't take into consideration the number of possible values within the sequence. Alternatively, the quick-sort algorithm, which basically splices the array into three partitions, would only take O(n) time, since there are only two possible values and three partitions.

C-12.39 Given an array A of n integers in the range [0, n^2 − 1], describe a simple method for sorting A in O(n) time.

The given array can be sorted in O(n) time using radix sort, which takes O(v\*(n+b)) time to complete.

Let v be number of digits in the input values, and b be the base for representing integers.

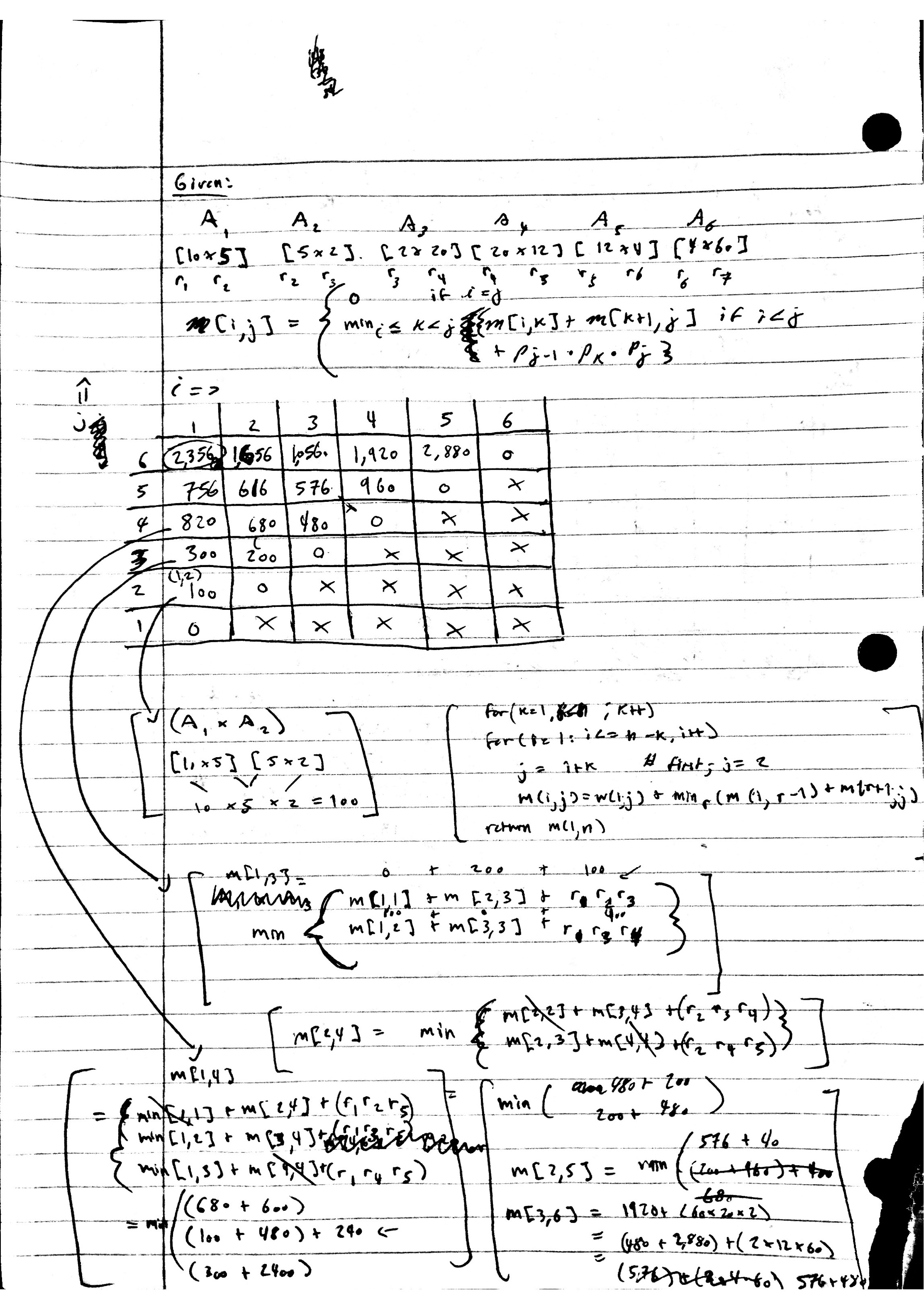
n^2-1 is the maximum value of the array, so the value of v is O(log(base b)(n))

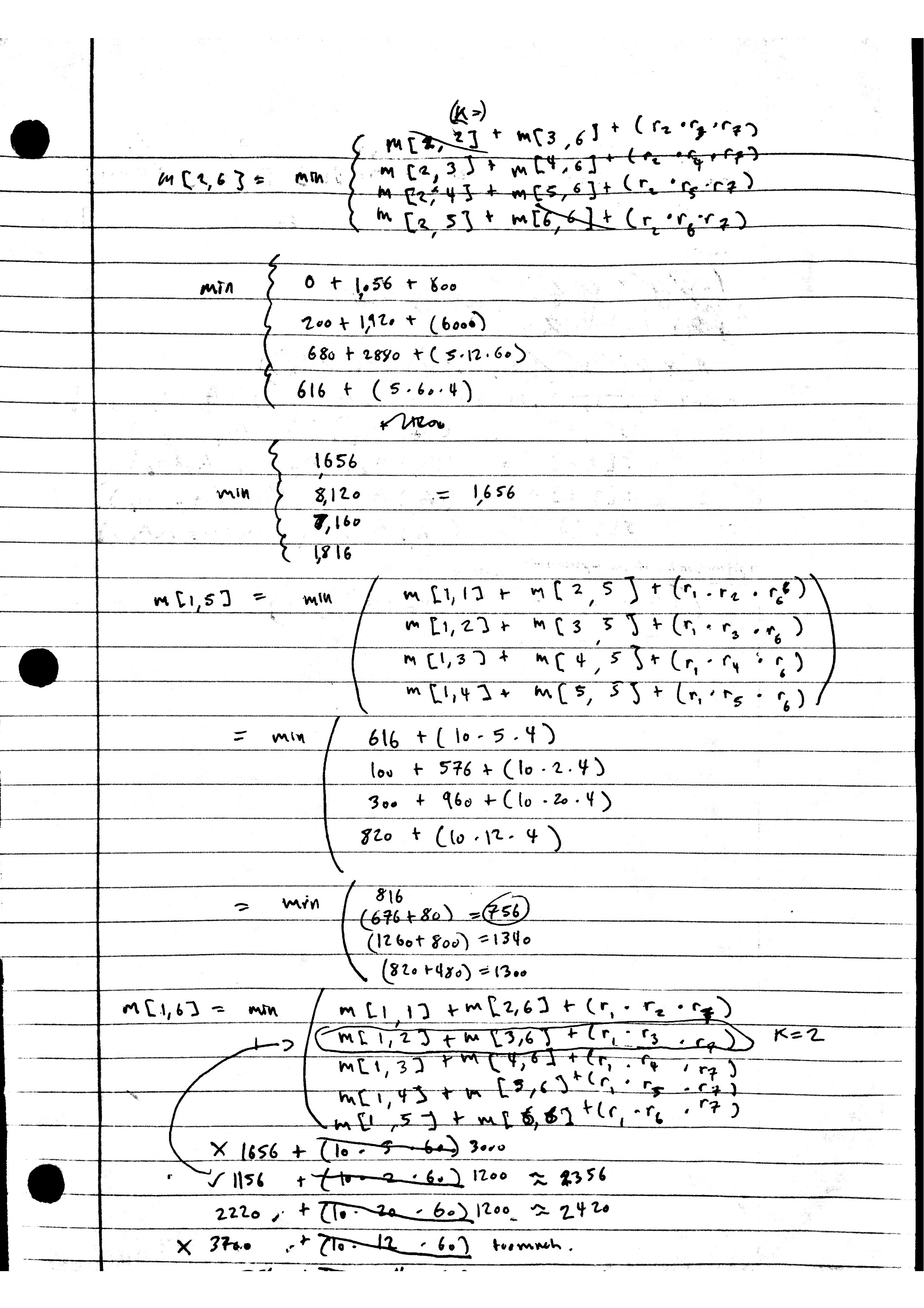
Thus, the overall time complexity is O((n+b)\*O(log(base b)(n))).

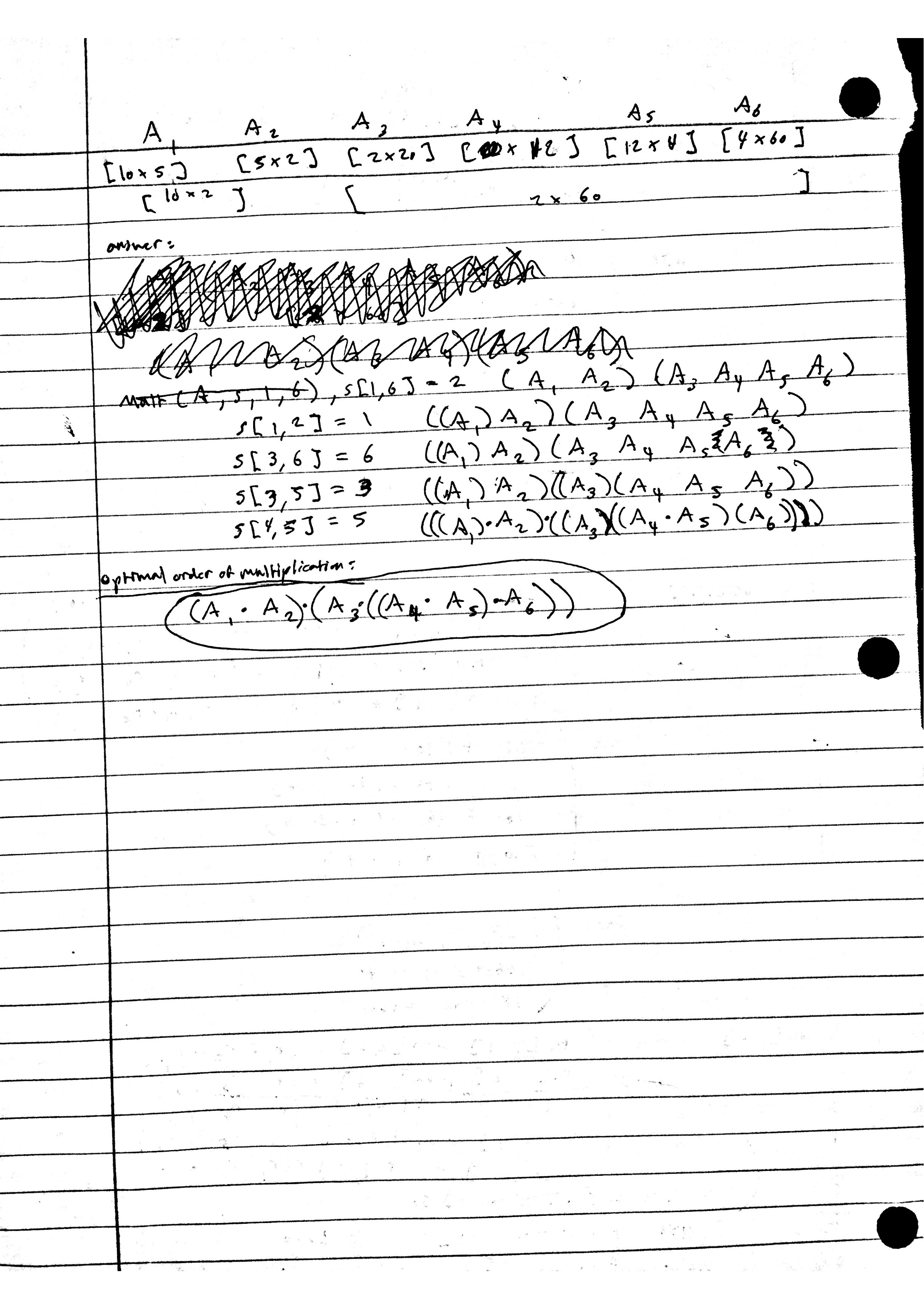
If we set the value of b to n, the value of O(log(base b)(n)) becomes O(1), and the overall time is within O((n+n)\*O(1)), which can be simplified to O(2n\*O(1)).

The smaller order of big-Oh gets and the constants get ignored, leaving the overall time complexity at O(n).

R-13.8 What is the best way to multiply a chain of matrices with dimensions that are 10×5, 5×2, 2×20, 20×12, 12×4, and 4×60? Show your work:







R-13.9 In Figure 13.8, we illustrate that GTTTAA is a longest common sub-sequence for the given strings X and Y. However, that answer is not unique. Give another common subsequence of X and Y having length six.

CTAATA

R-13.10 Show the longest common subsequence array L for the two strings:

X = "skullandbones"

Y = "lullabybabies"

What is a longest common subsequence between these strings?

array\_L = ["u","l","l","a","b","e","s"]

LCS = "ullabes"

C-13.29 Describe an efficient greedy algorithm for making change for a specified value using a minimum number of coins, assuming there are four denominations of coins (called quarters, dimes, nickels, and pennies), with values 25, 10, 5, and 1, respectively. Argue why your algorithm is correct.

Given some n.

1. If n is a decimal value, remove the decimal. (1.50) = 150
2. Create variables representing the number of each coin denomination that can fit into the value given, and find out the value of n after subtracting each number of coins from the quarters down to nickels.

quarters = n/25

n = n-quarters \* 25

dimes = n/10

n = n-dimes \* 10

nickels = n/5

1. Finally, find the number of pennies left after subtracting all the higher denomination coins.

pennies = n - nickels \* 5

1. Answer: Smallest # of coins = (quarters + dimes + nickels + pennies)

Since the coins with the highest denomination give us the most value for the least cost (highest $ for least coins), we should begin by seeing how many coins of each denomination can fit into the given value consecutively. First, divide the value by the dollar amount for quarters and subtract it from the value, then do the same for dimes with the new value, and nickels. Finally, the value that is left over can be accounted for with pennies. The answer is the number of each denomination of coin.

Here is a simple python implementation:

def greedy\_change(val):

v = int(val\*100)

end = dict()

total = 0

q = v/25

v = v-q\*25

end['quarters']=q

d = v/10

v = v-d\*10

end['dimes']=d

n = v/5

end['nickels']=n

p = v-n\*5

end['pennies']=p

for e in end.values():

total += e

return repr(end)+'\ntotal coins: '+str(total)

CH5 Exercises:

-------------------------------------------------------------------------------------------------------

R-5.5 Redo the justification of Proposition 5.1 assuming that the cost of growing the array from size k to size 2k is 3k cyber-dollars. How much should each append operation be charged to make the amortization work?

Each append operation should be charged 6k cyber-dollars to make the amortization work.

R-5.6 Our implementation of insert for the DynamicArray` class, as given in Code Fragment 5.5, has the following inefficiency. In the case when a re- size occurs, the resize operation takes time to copy all the elements from an old array to a new array, and then the subsequent loop in the body of insert shifts many of those elements. Give an improved implementation of the insert method, so that, in the case of a resize, the elements are shifted into their final position during that operation, thereby avoiding the subsequent shifting.

def insert(self, k, value):

”””Insert value at index k, shifting subsequent values rightward.”””

# (for simplicity, we assume 0 <= k <= n in this verion)

if self.n == self.capacity:

self.A[n+1:] +=[0]\*self.capacity

self.A[k:]+=[value]+self.A[k+1:]

self.n += 1

R-5.8 Experimentally evaluate the efficiency of the pop method of Python’s list class when using varying indices as a parameter, as we did for insert on page 205. Report your results akin to Table 5.5.

CH9 Exercises:

---------------------------------------------------------------------------------------------------------

R-9.3 What does each remove min call return within the following sequence of priority queue ADT methods: add(5,A), add(4,B), add(7,F), add(1,D), remove min(), add(3,J), add(6,L), remove min(), remove min(), add(8,G), remove min(), add(2,H), remove min(), remove min()?

R-9.13 Illustrate the execution of the in-place heap-sort algorithm on the follow- ing input sequence: (2,5,16,4,10,23,39,18,26,15).

R-9.15 Explain why the description of down-heap bubbling does not consider the case in which position p has a right child but not a left child.

R-9.18 Show that the sum,

n

∑ log i

i=1

Which appears in the analysis of heap-sort, is Ω(n log n).

Ω(f(n)) means:

There exists some constant C and integer k such that:

For all n>k, f(n) >= C\*g(n).

Choose k=0, C=1

Need to show:

For all n>0; (log i + log i+1 + ... + log n) >= (n log n).

Plug in:

(log i + log i+1 + ... + log n) >= C \* (n log n)

n =1, C=1

log (1) >= (1) log (1)

0 >= 0

Thus, the summation above is Ω(n log n).

R-9.22 Show all the steps of the algorithm for replacing key of entry (5,A) with 18 in the heap of Figure 9.1, assuming the entry had been identified with a locator.

C-9.35 Given a heap T and a key k, give an algorithm to compute all the entries in T having a key less than or equal to k. For example, given the heap of Figure 9.12a and query k = 7, the algorithm should report the entries with keys 2, 4, 5, 6, and 7 (but not necessarily in this order). Your algorithm should run in time proportional to the number of entries returned, and should not modify the heap.