Last Name: *Chen* **First Name:** *Robert* **Email:** *rchen87@gatech.edu*

Problem 1: Dynamic Programming, Max Sum Contiguous Subsequence (10 points)
A contiguous subsequence of a list S is a subsequence made up of consecutive elements of S. For instance, if S is $(5, 15, -30, 10, -5, 40, 10)$ then $(15, -30, 10)$ is a contiguous subsequence, but $(5, 15, 40)$ is not. Give a linear time algorithm for the following task:
Input: A list of numbers, $(a_1, a_2, \ldots, a_n)$.
Output: The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero).
For the preceding example, the answer would be $(10, -5, 40, 10)$, with a sum of 55.
Hint: For each $j \in \{1, 2, \ldots, n\}$ consider a contiguous subsequences ending exactly at position j.

Answer:
We are going to solve this with dynamic programming.
Let table T store the max sums. Therefore value $T(i)$ will be the max sum of contiguous subsequence ENDING AT s_i, THE $i-th$ number in the sequence.

Base case is

$$T(1) = s_1$$

because max possible subequence if only one item is just that item. Similarly for an array with 2 items we do:

$$T(2) = max(s_1 + s_2, s_2) = max(T(1) + s_2, s_2)$$

To generalize we do teh following:

$$T(j) = max(T(j-1) + s_j, s_j)$$

Meaning that the max sum subsequence is either the max sum from the item in the array before this one added to this one, or just this one. Basically if $T(j-1) + s_j$ is negative and s_j greater , then s_j will be bigger. Algorithm shown below:

```
def max_contig_sum_subsequence(S):
    # input: array S
    # output: max sum of subseq in S
    n = len(S) # n= length of S
    T = []
    T[0] = 0
    max_sum = 0 # update max sum
    l_best_subseq = [S[0] ] #update max subseq; initialize at first element of S
    for j in range(1, n):
        T[j] = max( [ T[j-1] + S[j], S[j] ] )
    for j in range(0, n):
        if T[j] > max_sum:
            max_sum = T[j] # remember best sum
            l_best_subseq.append( S[j] )# remember best subseq
        else:
            l_best_subseq = [ S[j] ] # best subseq would be subseq starting with this s_j
    return max_sum, l_best_subseq
```

This is O(2n) = O(n). Linear time.

Problem 2, Dynamic Progreamming Application (10 points)
You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \ldots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance a_n), which is your destination. You would ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the penalty for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penaltythat is, the sum, over all travel days, of the daily penalties. Give and analyze a polynomial time algorithm that determines the optimal sequence of hotels at which to stop.

Answer:
We are going to solve this with dynamic programming.
Subproblems:
Let $T(i)$ be the minimum total penalty to get to hotel i.

Recursion:
The value of $T(i)$ will be basically be, consider all possible hotels j we stay at the night before reaching hotel i. For each of these, the minimum penalty to reach i is the sum of the following:

1. $T(j)$ = min penalty to reach j
2. cost $200 - (a_j - a_i))^2$ = cost of a trip from j to i taking one day

Therefore, min penalty to reach i is the min of these values over all possible j:

$$T(i) = min_{0 \leq < i}\{T(j) + (200 - (a_j - a_i))^2\}$$

Note that $T(0) = 0$ (base case).

```
def hotel_trip(a):
    # input: a, a list of hotel mile posts
    # output: the best optimal cost, and the best sequence for that

    T = [] # initialize table

    T[0] = 0 # base case

    best_cost_path = []

    for i in range(1, n):
        best_cost = infinity
        best_j_for_this_i = NIL

        for j in range(0, i-1):
            cost_this_i_to_this_j = T[j] + (200-(a[j]- a[i]))^2
            if cost_this_i_to_this_j < best_cost:
                best_cost = cost_this_i_to_this_j
                best_j_for_this_i = j

        T[i] = best_cost #record best cost to point i
        best_cost_path.append(best_j_for_this_i) #append to optimal seq up to point i


    return T[n], best_cost_path
```

Complexity: This is $O(n^2)$ because you have to loop thru all hotels i (hotel you want to end at on current day) and then inside that loop you have to loop thru all hotels j (hotels you could have ended at on previous day).

Problem 3, Dynamic Programming, Making Change (10 points)
Given an unlimited supply of coins of denominations $x_1 < x_2 < \ldots < x_n$, we wish to make change for a given value v; that is, we want to find a set of coins whose total value is v. This might not be possible: for example, if the denominations are 5 and 10 then we can make change for 15 but not for 12. Give an $O(nv)$ dynamic programming algorithm for the following problem:
Input: $x_1, x_2, \ldots, x_n$
Output: Yes or No, is it possible to make change for v using denominations $x_1, x_2, \ldots, x_n$?

Answer:
We solve this with dynamic programming.

First, define the lookup table T as follows:
Table size of $v+1$. Table entry $T(i)$ is an indicator (true vs false) indicating whether or not the value i can be made from the coins. There is an entry in T for all values from 0 to v.
The recursive definition of $T(i)$ is:

$$T(v) = \vee_{1 \le i \le n} x = \begin{cases} T(v - x_i), & \text{if } x_i \le v, \\ FALSE, & \text{otherwise.} \end{cases}$$

```
def coin_change(x , v):
    # inputs:
    #   x: array of coin denominations
    #   v: value you want to make
    # output:
    #   True if possible to make change for v with items in x
    #   False if not possible

    ## declare lookup table T of size v+1, init to all Falses
    T[0] = True
    for i in range(1,v):
        T[i] = False
    for v_lookup in range(1, v):
        for j in range(1, n):
            if x[j] <= v:
                T[v_lookup] = T[v_lookup] and T[v - v_lookup] ## set to True if
                                                             ## T[v_lookup] == True
                                                             ## and T[v - v_lookup] == True
                                                             ## else set to False

            else:
                T[v_lookup] = False
    if T[v] == True:
        return "YES"
    else:
        return "NO"
```

Problem 4, Dynamic Programming, Making Change Optimally (10 points)
Here is yet another variation on the change-making problem. Given an unlimited supply of coins of denominations $1 = x_1 < x_2 < \ldots < x_n$, we wish to make change for a value v using as few coins as possible (note that, since $x_1 = 1$, any value v is realizable.)
(a) Give an $O(nv)$ algorithm that finds the minimum number of coins, say $k(v)$, whose total value is v.
(b) Give an $O(nv)$ algorithm that finds a solution $s_1, \ldots, s_n$, where s_i denotes the number of coins of denomination x_i, $1 \leq i \leq n$, such that (a)$\sum_{i=1}^{n} s_i x_i = v$ and (b)$\sum_{i=1}^{n} s_i = k(v)$.

Answer:

We are going to solve this with dynamic programming.

PART a
Table definition: Defining the lookup table as T, where the table entry $T(i, j)$ is the minimum number of coins, from the coins with denominations $x_1, ..., x_i$ and combined value j.

Recurrence: The recurrence will be as follows:
When considering the $i-th$ coin, we set the table entry to be the **minimum** of the following two choices: 1. coin i is used , 2. coin i is NOT used. There are two cases based on whether or not the value of $i-th$ coin is greater than j or not:
Recurrence relation:
if $x_i \leq j$:
$T(i, j) = min\{T(i-1, j-x_i)+1, T(i-1, j)\}$
else:
$T(i, j) = T(i-1, j)$

```
def coin_change_a(x , v):
    # inputs:
    #   x: array of coin denominations
    #   v: value you want to make
    # output:
    #   minimum number of coins whose total value is v

    ## declare lookup table T
    n = len(x) # n= number of different coins

    T[0, 0] = 0
    for j in range(1,v):
        T[0,j] = infinity
    for i in range(1, n)
        for j in range(1, v):
            if x[i] <= j:
                T(i,j) = min( [ T(i-1, j-m*x[i])+m, ... T(i-1, j) ]) , such that j = m*x[i]
                                                                    # the best way to make value j is
                                                                    # either simply add current coin i,
                                                                    # i.e., T(i-1, j-x[i])+ 1
                                                                    # OR, use whatever best combo of
                                                                    # coins up to last coin i-1 that
                                                                    # still yielded value j
                                                                    # i.e., T(i-1, j)
            else:
                T(i,j) = T(i-1, j) # if current coin bigger than
                                   # target value j, then you can only use
                                   # whatever best combo up to x_{i-1} is
    return T(n,v) # return the min number of coins
                  # from x1...xn that yield value v
```

PART b
To do Part b, we are essentially trying to find the number of coins of each denomination needed in order to

make a total value of v. The answer is pretty much same as the one for part a, with an additional step to simply count the number of new coin i to add to your collection.

```
def coin_change_b( x , v ):
    # inputs:
    #   x: array of coin denominations
    #   v: value you want to make
    # output:
    #   min_coins = minimum number of coins whose total value is v
    #   l_num_each_coin = array ( n elts) of number of coins from x_1 ... x_n
    #

    ## declare lookup table T
    n = len(x) # n= number of different coins
    l_num_each_coin = []       # initialize list of coin counts
    for i in range(1, n):
        l_num_each_coin[i] = 0 #initialize coin counts to 0

    T[0, 0] = 0
    for j in range(1,v):
        T[0,j] = infinity
    for i in range(1, n)
        for j in range(1, v):
            if x[i] <= j:
                T(i,j) = min( [ T(i-1, j-m*x[i])+m, ... T(i-1, j) ]) , such that j = m*x[i]
                                                                # the best way
                                                                # to make value j is
                                                                # either simply add any number m of current coin i,
                                                                # i.e., T(i-1, j-x[i])+ 1
                                                                # OR, use whatever best combo of
                                                                # coins up to last coin i-1 that
                                                                # still yielded value j
                                                                # i.e., T(i-1, j)
                if T(i-1, j- m * x[i])+ m <  T(i-1, j): #
                    print "add m of coin i"
                    l_num_each_coin[i] += 1 # add one of coin i
                else: #
                    print "do nothing" # don't add any new coins because there's already a
                                       # more optimal solution involving coins x1 to x_{i-1}
            else:
                T(i,j) = T(i-1, j) # if current coin bigger than
                                   # target value j, then you can only use
                                   # whatever best combo up to x_{i-1} is
    min_coins = T(n,v)


    return min_coins, l_num_each_coin # return the min number of coins
                 # from x1...xn that yield value v
```

Problem 5, Dynamic Programming, Longest Path out of Specific Vertex (10 points)
Let $G(V, E)$ be a directed acyclic graph presented in topologically sorted oredr, that is, if $V = \{v_1, \ldots, v_n\}$, then $(v_i \to v_j \in E) \Rightarrow (i < j)$ - ie, all edges are directed from lower order vertices to higher order vertices. Give an $O(|V| + |E|)$ complexity alogithm that finds the length of the longest path that starts in v_1.

Answer:
We have to keep track of longest path from each vertex in the graph. Work backwards (in reverse topological order) from the end and continually find longest path from that vertex. The running time is $O(|V| + |E|)$ because you loop through all the vertices and you assess all edges connected to each vertex. Theres only E vertices you assess because it is a DAG.

```
def longest_path_from_sorted_DAG(G):
    # input: G=(V,E)
    # output: LENGTH of the longest path in G
    let n = |V|  #let n = number of vertices

    # define array length_longest_path_from_this_v
    #   -- this array stores the length of longest path starting from vertex v
    length_longest_path_from_this_v = []
    for v_i in V:
        length_longest_path_from_this_v[v_i] = 0

    # loop thru all vertices in reverse topological order
    #   -- check
    for v_i in V (reverse order):
        if v_i has no outgoing edges:
            length_longest_path_from_this_v[v_i] = 0
        else:
            for all u in V such that (v,u) in E:
                if length_longest_path_from_this_v[v_i] < length_longest_path_from_this_v[u] + 1:
                    length_longest_path_from_this_v[v_i] = length_longest_path_from_this_v[u] + 1

    return length_longest_path_from_this_v[1]
```

Problem 6, Wiggly Sorting, KSelect Application (10 points)
Let $a_1, \ldots, a_n$ be an unsorted array of n distinct integers, where n is odd. We say that the array is *wiggly-sorted* if and only if $a_1 < a_2 > a_3 < a_4 > a_5 < \ldots > a_{n-2} < a_{n-1} > a_n$.
Give a linear time algorithm that wiggly-sorts the initial array $a_1, \ldots, a_n$.

Answer:

Let $A = [a_1, ..., a_n]$ be the array.

Let B be the wiggle-sorted array.

1. Find the median k of the array A. The median can be found with a median finding algorithm which is $O(n)$.

2. Place all items less than the median in an array A_{lowers}. Place all items greater than the median in an array $A_{highers}$.

3. Set $B[1]$ to be the first element of the wiggle sorted array.

4. For $i = 2, ...n$:
if i **is even**, then set $B[i]$ to be number from from $A_{highers}$
if i **is odd**, then set $B[i]$ to be number from from A_{lowers}

Since everything from $A_{highers}$ is going to be greater than median then therefore everythign from $A_{highers}$ is going to be greater than everything from A_{lowers}.

The above will be $O(n)$. It takes $O(n)$ to find median of A and it requires n steps to fill array A_{lowers} and $A_{highers}$. It takes n steps to fill array B. Linear time. Yes.

Problem 7, Finding Median under Partial Sorting (10 points)
Let X and Y be sorted arrays of length n. Suppose also that the elements of $X \cup Y$ are all distinct. Give an $O(\log n)$ comparison algorithm that finds the n-th element of $X \cup Y$.

Answer: We are going to find the median recursively. To find the median of an array that is already sorted is O(1) – you simply take the $n/2 - th$ element.

Let $x*$ be the median of array X and let $y*$ be the median of array Y.

Recursively do the following:

1. find the median of X, $x*$, and find the median of Y, which is $y*$.
2. there are two cases:

case 1 if $x* < y*$, then all elements in $Y[\frac{n}{2}...n]$ are greater than $x*$, so the median cannot be in $X[1...\frac{n}{2}]$ or $Y[\frac{n}{2}...n]$. therefore you throw those two halves away and recurively solve subproblem with $X[\frac{n}{2}...n]$ and $Y[1...\frac{n}{2}]$.

case 2 if $y* < x*$, then all elements in $X[\frac{n}{2}...n]$ are greater than $y*$, so the median cannot be in $Y[1...\frac{n}{2}]$ or $X[\frac{n}{2}...n]$. therefore you throw those two halves away and recurively solve subproblem with $Y[\frac{n}{2}...n]$ and $X[1...\frac{n}{2}]$.

In either case: each iteration results in half the size subproblem so therefore you are doing $O(lg\ n)$ comparisons.

The recurrence for this is the following:

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

which means that

$$T(n) = O(lg\ n)$$

The running time is $O(lg\ n)$ which you can verify with the master theorem.

Problem 8, Strong Majority, KSelect Application (10 points)
Let $a = a_1 \dots a_n$ be an input array of n unsorted and not necessarily distinct numbers. We say that a value m is a *strong majority* of a if and only if the value of at least $\left(\lfloor \frac{n}{2} \rfloor + 1\right)$ elements of a is equal to m:

$$|\{i \,:\, a_i = m, 1 \le i \le m\}| \ge \left(\lfloor \frac{n}{2} \rfloor + 1\right) \quad .$$

Give a linear time algorithm that determines if the input array a has a strong majority.
If the answer is YES, then give also the value m of the strong majority.

Answer:

```
def strong_majority_finder(a = [1,2,3,3,3,3]):
    """
        Inputs: a is array
        In our example default for a, the value 3 would be the strong majority
        because (floor(n/2) +1)  =  4 and the number 3 happens 4 times.

        Running time: This is O(n). If there is to be a strong majority
        then it should appear consecutively more often than any other
        number. It has to, in order for at least half the values
        to be equal to that.
    """
    n = len(a) # n is the length of a
    ## if theres 2 items in array and they're the same then return "YES"
    ## and value of the strong majority
    if n ==0:
        return "NO" # empty array --> no majority
    if n ==1:
        return "YES" # if only 1 item in array then its the majority
    if n ==2:
        if a[0] = a[1]:
            return ("YES", a[1])
        else:
            return "NO"
    ## if theres more than 2 items in array then create temporary array
    ## if any consecutive 2 numbers are the same then put that into the temp array
    temp = []
    for i in range(n-1): #compare every adjacent set of 2
        if a[i] = a[i+1]:
            temp.append(a[i]) # insert a[i] into temp array temp
        i = i + 1
    return strong_majority_finder(temp)
```

Problem 9, Unknown Length Binary Search (10 points)
You are given an infinite array $a(\cdot)$ in which the first n cells contain positive integers in sorted order and the rest of the cells are not defined. By not defined, we mean that if an algorithm probes a position $i > n$ then the program returns an error message. You are not given the value of n. Describe an $O(\log n)$ comparison algorithm that takes a positive integer x as input and finds a position in the array containing x, if such a position exists, otherwise the algorithm returns NO.

Answer: We are going to do a one-sided binary search. This is O(n) and it searches starting from the left side of the array.

```
def binary_search_unknown_length(A = [1,2,3,4, nan, nan, nan , nan]):
    # goal is to find index i such that x<A[i]
    # start at i = 1
    # if x > A[i] then double i
    # repeat until x <= A[i];
    # return x if x == A[i], else return "NO"
    i = 1
    if x == A[i]:
        return x
    while (x  > A[i] and A[i] != np.nan): # check that x > A[i] and A[i] not undefined
        if x > A[i]:
            i = i+2  #keep doing this until we find upper bound for i
                    # this i will be increased at most lg(n) times
        else:
            return binary_search_unknown_length(A[0:i]) #binary search on everything up to this i
```

Problem 10, Counting Significant Inversions (10 points)
Let $\overline{a} = (a_1, \ldots, a_n)$ be an array of n distinct unsorted numbers. Say that a pair $1 \leq i < j \leq n$ is a *significant inversion* on $\overline{a}$ if and only if $a_i > 2a_j$. Give an $O(n \log n)$ comparison algorithms that determines the number of inversions of an arbitrary array of n distinct unsorted integers.
Answer:

this is essentially a modified merge sort algo where you have an additional step to check if there is inversion whenever comparing an element from the left sub-array with an element from the right sub-array and if the element from the left sub-array is bigger than 2x the element from the right sub-array then you increment your count by 1.

```
cnt_subinversions = 0

def count_significant_inversions(A = [1,2,3,6,11]):
    # goal is to COUNT the number of inversions;
    # in the example input array the inversions
    #   are (2,3) and (6,11) => num inversions is 2
    #
    # this is essentially a modified merge sort algo
    # running time: O(n lg n)
    #

    n = len(A)
    # first, recursively partition the sequence of n elements into
    # two subsequences with the same size until only one
    # element is in each sub sequence

    # merging subarrays
    if len(A) > 1:
        midpoint = len(A)//2
        array_left = A[ 0 : midpoint]
        array_right = A[ midpoint : n]
        count_significant_inversions(array_left)
        count_significant_inversions(array_right)

        i, j, k = 0
        while i < len(array_left) and j < len(array_right):
            if array_left[i] < array_right[j]:
                A[k] = array_left[j]
                i = i+1
            else: #elt from left side is bigger than elt from right side
                if array_left[i] > 2 * array_right[j]:#check for significant inversion
                    cnt_subinversions += 1 #add to count of inversions
                A[k] = array_right[j]
                j = j+1
            k = k+ 1

        while i < len(array_left):
            A[k] = array_left[i]
            i= i + 1
            k = k+ 1

        while j < len(array_right):
            A[k] = array_right[j]
            j = j+1
            k = k+1
```