Last Name: .... **First Name:** .... **Email:** ....

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:
Assume that the list contains at least one non-negative integer, otherwise, if all elements are negative, the contiguous subsequence of maximum sum must have length zero, since any non-zero length contiguous subsequence will have negative sum.

Let s_k be sum of the maximum sum contiguous subsequence ending at k (and including a_k), $1 \leq k \leq n$.
We may now express the RECURSIVE FORM of the problem is $s_k := \max\{s_{k-1} + a_k, a_k\}$, for $1 \leq k \leq n-1$, with INITIALIZATION $s_0 := 0$. Using the above recurrence relation, we can compute the sum of the maximum sum contiguous subsequence, which would be the maximum over s_k. This would also indicate the ending point of the subsequence.

The problem also asks for the starting point. It is easy to see that this will be just after the last position at which the sum went negative, since a higher sum can always be found by dropping any negative-sum prefix. The above leads us to the following MEMOIZATION in the implementation:

```
$s_0 := 0$
for $k := 1$ to $n$
        if $(s_{k-1} + a_k \geq s_{k-1})$ then $s_k := s_{k-1} + a_k$
                                          else $s_k := a_k$
max := 0
endmax:= 0
for $i := 1$ to $n$ if $s_k \geq$ max then { max := $s_k$
                                           { endmax := $k$
beginmax:=endmax
while $s_{\text{beginmax}-1} \geq 0$
         beginmax:= beginmax − 1
RETURN (max, beginmax, endmax)
```

Complexity Analysis: This is a linear algorithm. It consists of three independent iteration, with each iteration having at most n steps.

Problem 2, Dynamic Programming 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 penalty, that 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:
Let C_i be the minimum cost if you were to start at mile 0 and complete your trip at hotel i. Let us first give a recursive rule for computing C_i, $1 \leq i \leq n$, where, obviously, $C_1 = 0$, and we eventually want to know C_n. In general, to compute C_i, we consider all places k, $1 \leq k < i$, that we might have stopped the night before. The cost of having k as the previous stop is the minimum cost of getting to hotel k, which is C_k and has been already computed, followed by the cost of travelling in one day from k to i, a distance of $(a_i - a_k)$. Minimizing over all k gives the following RECURSIVE FORM OF THE SOLUTION:

$$C_i = \min_{1 \leq k < i} \{C_k + (200 - (a_i - a_k))^2\}$$

Complexity Analysis: This can be implemented with dynamic programming in $O(n^2)$, going from $i = 2$ to n, and checking i different possibilities in step i. Finally, while computing C_i, we may also remember which k minimized C_i in a new array P_i, with $P_1 = \text{Nil}$.

```
Initialization $C_1 := 0$ and $P_1 = \text{Nil}$
               for $i := 2$ to $n$ $\left\{ \begin{array}{l} C_i := (200 - a_i)^2 \\ P_i := 1 \end{array} \right.$
Memoization for $i := 3$ to $n$
                for $k := 2$ to $i - 1$
                    if $C_i > C_k + (200 - (a_i - a_k))^2$ then $\left\{ \begin{array}{l} C_i := C_k + (200 - (a_i - a_k))^2 \\ P_i := k \end{array} \right.$
Output RETURN($C_n$)
       last := $n$
       while (last $\neq$ Nil)
           RETURN(last)
           last := $P_{\text{last}}$
```

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 introduce an additional coin $x_0 = 0$, for convenience, and initialize index x_0 to Yes.
We may assume $v \geq x_n$ and initialize indices x_i, $1 \leq i \leq n$, to No.
The RECURSIVE FORM OF THE SOLUTION then is

$$\text{Possible}(k) := \text{Yes}, \text{ if and only if, for some } i \text{ with } x_i \leq k, \text{ Possible}(k - x_i) = \text{Yes}$$

<u>Complexity Analysis</u> We will evaluate Possible(k) from 0 to v, as usual with dynamic programming. In each step we have to check at most n $x_i's$, for total running time $O(vn)$.

```
Initialization for $k := 1$ to $v$ set Possible($k$) := No
               Possible(0) := Yes
Memoization for $k := 1$ to $v$
                  for $i := 0$ to $n$
                     if (($k - x_i \geq 0$) AND (Possible($k - x_i$) = Yes)) then Possible($k$) := Yes
```

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:
(a) We introduce an additional coin $x_0 = 0$, for convenience, with $k(0) = 0$.
We may assume $v \geq x_n$ and initialize indices $k(u) = u$, $1 \leq u \leq v$.
The RECURSIVE FORM OF THE SOLUTION then is

$$k(u) := \min_{x_i \leq u} \{k(u), 1 + k(u - x_i)\}$$

Complexity Analysis We will evaluate $k(u)$ from 0 to v, as usual with dynamic programming.
In each step we have to check at most n $x_i's$, for total running time $O(vn)$.

```
Initialization for $u := 1$ to $v$ set $k(u) := u$
               $k(0) := 0$
Memoization for $u := 1$ to $v$
                for $i := 0$ to $n$
                    if (($u - x_i \geq 0$) AND ($k(u) > 1 + k(u - x_i)$)) then $k(u) := 1 + k(u - x_i)$
```

(b) To actually find a solution that achieves $k(v)$, we only need to remember which coin x_i minimized $k(u) := \min_{x_i \leq u}\{k(u), 1 + k(u - x_i)\}$ (similar to Problem 2.)

```
Initialization for $u := 1$ to $v$ set $\begin{cases} k(u) := u \\ P(u) := 1 \end{cases}$
               $k(0) := 0$ and $P(0) :=$ Nil
Memoization for $u := 1$ to $v$
                for $i := 0$ to $n$
                    if (($u - x_i \geq 0$) AND ($k(u) > 1 + k(u - x_i)$)) then $\begin{cases} k(u) := 1 + k(u - x_i) \\ P(u) := i \end{cases}$
Output RETURN($k(v)$)
    for $i := 1$ to $n$ set $s_i := 0$
        last := $v$
    while (last $\neq$ Nil)
        $I := P$(last)
        $s_I := s_I + 1$
        last := last $- x_I$
    for $i := 1$ to $n$ RETURN($s_i$)
```

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:
Let $L(i)$ be the length of the longest path that starts in vertex v_i, $1 \leq i \leq n$.
We know that $L(n) := 0$ and we want $L(1)$.
The RECURSIVE FORM OF THE SOLUTION is $L(i) := \max_{j: v_i \to v_j \in E}\{1 + L(j)\}$.
Using dynamic programming, we can implement this computing $L(i)$ starting from n and going down to 1.

```
Initialization for $i := 1$ to $n$ set $L(i) := 0$
Memoization for $i := (n-1)$ to 1
                for all $j$ such that $v_i \to v_j \in E$
                        $L(i) := \max\{L(i), 1 + L(j)\}$
Output RETURN($L(1)$)
```

Complexity Analysis:
In the main loop, each vertex and each edge are examined once, for total complexity $O(|V| + |E|)$.

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:
We use the linear time KSelect algorithm for $k = (n+1)/2$ and find the median m of $a_1, \ldots, a_n$.
We scan the array once and find $(n+1)/2$ elements smaller than or equal to m and $(n-1)/2$ elements strictly larger than m.
We place the first set of elements in the odd position of the original array, and the second set of elements in the even positions of the original array.
This results in a wiggly arrangement, and the entire algorithm completes in linear time.

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 will argue that using one comparison we reduce the problem to one of half the size.
Case 1: n is odd. Compare $X(\frac{n+1}{2})$ with $Y(\frac{n+1}{2})$. Assume, without loss of generality, that $X(\frac{n+1}{2}) < Y(\frac{n+1}{2})$. Then, we know that elements $X(1)\dots X(\frac{n-1}{2})$ are smaller than elements $X(\frac{n+1}{2})\dots X(n)$ as well as elements $Y(\frac{n+1}{2})\dots Y(n)$, so they are smaller than a total of $(n+1)$ elements of $X \cup Y$. Thus elements $X(1)\dots X(\frac{n-1}{2})$ are smaller than the n-th element of $X \cup Y$.
Similarly, we know that elements $Y(\frac{n+3}{2})\dots Y(n)$ are larger than elements $Y(1)\dots Y(\frac{n+1}{2})$ as well as elements $X(1)\dots X(\frac{n+1}{2})$, so they are larger than a total of $(n+1)$ elements of $X \cup Y$. Thus elements $Y(\frac{n+3}{2})\dots Y(n)$ are larger than the n-th element of $X \cup Y$.
The problem now reduces to sorted arrays $X' = X(\frac{n+1}{2})\dots X(n)$ and $Y' = Y(1)\dots Y(\frac{n+1}{2})$, each having $\frac{n+1}{2}$ elements, and we want the $\frac{n+1}{2}$-th element of $X' \cup Y'$.
Case 2: n is even. Compare $X(\frac{n}{2})$ with $Y(\frac{n}{2})$. Assume, without loss of generality, that $X(\frac{n}{2}) < Y(\frac{n}{2})$. Then, we know that elements $X(1)\dots X(\frac{n}{2}-1)$ are smaller than elements $X(\frac{n}{2})\dots X(n)$ as well as elements $Y(\frac{n}{2})\dots Y(n)$, so they are smaller than a total of $(n+2)$ elements of $X \cup Y$. Thus elements $X(1)\dots X(\frac{n}{2}-1)$ are smaller than the n-th element of $X \cup Y$.
Similarly, we know that elements $Y(\frac{n}{2}+2)\dots Y(n)$ are larger than elements $Y(1)\dots Y(\frac{n}{2}+1)$ as well as elements $X(1)\dots X(\frac{n}{2}+1)$, so they are larger than a total of $(n+2)$ elements of $X \cup Y$. Thus elements $Y(\frac{n}{2}+1)\dots Y(n)$ are larger than the n-th element of $X \cup Y$.
The problem now reduces to sorted arrays $X' = X(\frac{n}{2})\dots X(n)$ and $Y' = Y(1)\dots Y(\frac{n}{2}+1)$, each having $\frac{n}{2}+1$ elements, and we want the $\frac{n+1}{2}$-th element of $X' \cup Y'$.

Complexity Analysis: $T(n) = T\left(\frac{n}{2}\right) + 1$, with $T(1) = 1$, which solves to $O(\log n)$.

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

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

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

Answer:
Main Fact: If an element m is a strong majority then m is necessarily the median of $\overline{a}$ (if n is even then m is both the lower and upper median.)

The algorithm finds the median m of $\overline{a}$ in linear time, and scans the entire array once more to count how many elements are equal to m. If at least $\left(\lfloor \frac{n}{2} \rfloor + 1\right)$ elements are equal to m, then m is a strong majority of $\overline{a}$, otherwise $\overline{a}$ does not habe a strong majority element.

Complexity Analysis: Clearly linear time to find the median, and linear time to count the number of elements that are equal to the median.

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 $O(\log n)$ 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:
STEP 1: Find the length of the array, say, n.
(a) For this, check positions that are powers of 2, in increasing order $a(1), a(2), a(2^2), a(2^3), \ldots$, until we find the first position that does not contain an element of the array $a(\cdot)$. If, for some k, $a(2^k)$ contains an element of the array, but $a(2^{k+1})$ does not contain an element of the array, then we know that the array has length n: $2^k \leq n < 2^{k+1}$. Clearly, STEP 1(a), takes $O(\log_2 n)$ steps.
(b) Do binary search in the interval $a(2^k) toa(2^{k+1})$ to determine n. Clearly, STEP 1(b) also takes $O(\log_2 n)$ steps.
STEP 2: Perform the usual binary search on $a(1), a(2), a(3), \ldots, a(n)$ to determine if the array contains x. Of course, STEP 2 also takes $O(\log_2 n)$ steps.

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: Assume, without loss of generality, that n is a power of 2.
We may partition the significant inversions of $\overline{a} = (a_1, \ldots, a_n)$ to three sets:
The <u>first</u> set consists of pairs $1 \leq i < j \leq \frac{n}{2}$, with $a_i > 2a_j$.
The <u>second</u> set consists of pairs $\frac{n}{2} + 1 \leq i < j \leq n$, with $a_i > 2a_j$.
The <u>third</u> set consists of pairs $1 \leq i \leq \frac{n}{2} < \frac{n}{2} + 1 \leq j \leq n$, with $a_i > 2a_j$.
<u>Main Fact:</u> For the third set of inversions, it is important to observe that their number remains the same if we permute arbitrarily the first part $(a_1, \ldots, a_{\frac{n}{2}})$ and the second part $(a_{\frac{n}{2}+1}, \ldots, a_n)$. This is because, if there were elements $x > 2y$, with the original position of x in the first half of the array and the original position of y in the second half of the array, then these elements will give rise to a significant inversion no matter what position x is permuted to within the first half of the array, and no matter what position y is permuted to within the second half of the array.

We may now use mergesort$(a_1, \ldots, a_n)$:
(a) Where each recursive call mergesort$(a_1, \ldots, a_{\frac{n}{2}})$ and mergesort$(a_{\frac{n}{2}+1}, \ldots, a_n)$, in addition to sorting, each compute the number of the <u>first</u> and <u>second</u> sets of significant inversions respectively.
(b) The <u>third</u> set of inversions is computed by a modification of procedure merge as follows:
STEP 1: (a) Copy $(a_1, \ldots, a_{\frac{n}{2}})$ to new array $(b_1, \ldots, b_{\frac{n}{2}})$.
Populate new array $(b'_1, \ldots, b'_{\frac{n}{2}})$ with the values $(2a_{\frac{n}{2}+1}, \ldots, 2a_n)$.
Merge arrays b and b' in to new array $c(1) \ldots c(n)$ using two pointers as usual.
Whenever we take an element from $(b'_1, \ldots, b'_{\frac{n}{2}})$, we increment a global counter of significant inversions by the number of remaining elements of $(b_1, \ldots, b_{\frac{n}{2}})$.
Clearly STEP 1(a) takes time $O(n)$.
STEP 1: (b) Proceed to merge $(a_1, \ldots, a_{\frac{n}{2}})$ and $(a_{\frac{n}{2}+1}, \ldots, a_n)$ as usual.
Clearly STEP 1(b) also takes time $O(n)$, as we know from the analysis of merge in mergesort.

The modification of merge keeps the running time of merge at $O(n)$, thus the total running time is $O(n \log n)$, just as the unmodified mergesort.