Instructions. You will form groups to work on these problems in discussion section. Please turn in your own sheet in at the end of class.

1. **Maximum Subsequence Sum.** Find the MSS of -1, 7, -8, 7, -3, -3, 1, 6. Remember the MSS divide and conquer algorithm from class:

```plaintext
if length(Arr) = 1 then
  max(A[0], 0)
end if
mid = length(Arr)/2
L = MSS(Arr[0:mid])
R = MSS(Arr[mid:length(Arr)])
Set sum = 0, L' = 0
for i = mid-1 down to 0 do
  sum += Arr[i], L' = max(L', sum)
end for
Set sum = 0, R' = 0
for i = mid-1 up to length(Arr)-1 do
  sum += Arr[i], R' = max(L', sum)
end for
return max(L, R, L' + R')
```
2. General 1D Closest Pair

For this exercise, we will consider two algorithms, Monotonic-Number-Line-Closest-Distance and General-Number-Line-Closest-Distance.

The \texttt{median}(a) algorithm returns the median. For an odd-sized \(a\), median will return the value that would be in the middle of the array if \(a\) was sorted. For an even-sized \(a\), median will return the average of the two elements that would be in the middle if \(a\) was sorted. Note that if all elements in \(a\) were unique, it is ensured that the number of elements that are greater than the median will be equal to the number of elements that are less than the median. For this exercise, \texttt{median}(a) takes \(\Theta(n)\) time, where \(n\) is the length of \(a\).

The \texttt{range}(a, s, e) algorithm returns an array of all elements in array \(a\) that are between \(s\) and \(e\). \texttt{range} is inclusive of the lower bound \(s\), but not inclusive of the upper bound \(e\). For this exercise, \texttt{range}(a, s, e) takes \(\Theta(n)\) time, where \(n\) is the size of the output.

\textbf{Algorithm 1] Monotonic-Number-Line-Closest-Distance} (\(a, \text{length}\))

\begin{verbatim}
result = \infty
for i from 2 to length do
    dif = a[i] - a[i - 1]
    result = min(dif, result)
end for
return result
\end{verbatim}

\textbf{Algorithm 2] General-Number-Line-Closest-Distance} (\(a\))

\begin{verbatim}
mid = \text{median}(a)
low = \text{range}(a, -\infty, mid)
high = \text{range}(a, mid, \infty)
low-val = \text{General-Number-Line-Closest-Distance}(low)
high-val = \text{General-Number-Line-Closest-Distance}(high)
val = min(low-val, high-val)
result = val
low-boundary = \text{range}(low, median - val, mid). \{Begin Refactoring on This Line\}
for i \in low-boundary do
    possible-closest = \text{range}(high, mid, median + val).
    for j \in possible-closest do
        result = min(result, |i - j|)
    end for
end for \{Stop Refactoring on This Line\}
return result.
\end{verbatim}

Now you are given an array that is not sorted. \texttt{General-Number-Line-Closest-Distance} is closely based on the closest-distance algorithm shown in lecture. Assume that the input array is a power of 2 in length.

(a) How many elements can be in possible-closest?

(b) Refactor the code marked in \texttt{General-Number-Line-Closest-Distance}. No for loops should be needed in the refactored region.
(c) What is the complexity of General-Number-Line-Closest-Distance? Did it change during refactoring? If we sorted \( a \) and ran Monotonic-Number-Line-Closest-Distance, would we do better or worse than running General-Number-Line-Closest-Distance?

(d) Design as efficient an algorithm as possible to find the greatest distance between any two elements in the input array. What is the \( \Theta \) complexity of this? How does it compare with sorting and running your previous algorithm for finding the greatest distance?