Problem 1: Finding consecutive elements in an array that maximize sum.

Input: An array $A[1, \ldots, n]$ where each $A[i] \in \mathbb{Z}$.

Output: $i, j$ s.t. $1 \leq i \leq j \leq n$, and

$$\sum_{k=i}^{j} A[k]$$

is maximum over all $i, j$.

Note: If $\forall k, A[k] \geq 0$, then output $i=1, j=n$.

Note: $A = [−3, 7, 2, −8, 0, −3, 3, 15, 4, −7]$.

Naive: Try all $1 \leq i \leq j \leq n$ $O(n^3)$

Brute Force

Try all $i=1, j=1, 2, 3, \ldots, n$

Try all $i=2, j=2, 3, 4, \ldots, n$

Try all $i=n, j=n$

$O(n^2)$

Divide & Conquer

$\sqrt{\frac{n}{2}}$
Case 1: \( i, j \leq \frac{n}{2} \) 

Case 2: \( i, j > \frac{n}{2} \) 

Case 3: \( i \leq \frac{n}{2}, j > \frac{n}{2} \) Solution is of Special Type

\[
\begin{bmatrix}
-3 & 7 & 2 & -8 & 9 \\
-1 & 3 & -15 & 4 & 7 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
10 & 3 & -16 \\
2 & -13 & -9 & 1 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
-7 & 2 & -89 & -1 & 3 \\
\end{bmatrix}
\]

Running Time \( T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n) \).

KADANL'S ALGORITHM

Last One: \( \forall k, 1 \leq k \leq n \), find the best solution ending at \( k \) given the best solution ending at \( k-1 \).

Let \( v_k = \text{value of best solution ending at } k \).

Let \( v_k^* = \text{value of best solution seen so far} \).

Given \( v_{k-1}, v_{k-1}^* \), how to find \( v_k, v_k^* \).
KADANE'S ALGORITHM

INPUT: \( A[1..n] \) of elements in \( \mathbb{Z} \).

OUTPUT: \((i, j)\) such that
\[
\sum_{k=i}^{j} A[k] \text{ is maximized for all } 1 \leq i \leq j \leq n
\]

ALGORITHM:

Definition: For \( k := 0 \) to \( n \) define

\[ a) \quad v_k = \text{value of best solution ending at } k \]

\[ b) \quad v_k^* = \text{value of best solution for subarray } A[1..k] \]

Initialize \( v_0 := -\infty; \quad v_0^* := -\infty \)

Increment Step

\[
\begin{aligned}
\text{For } i := 1 \text{ to } n \text{ do} \\
&v_i := \max \left[ A[i], v_{i-1} + A[i] \right] ; \\
&v_i^* := \max \left[ v_{i-1}^*, v_i \right] \\
\end{aligned}
\]

RETURN Indices corresponding to \( v_n^* \).
\[
\begin{align*}
V_0^* &= -\infty \\
V_0^* &= -\infty
\end{align*}
\]

Assume there is an element in \( A \) that is \( \geq 0 \).

\[
\text{for } i = 1 \text{ to } n \text{ do}
\]

\[
\begin{align*}
V_i &= \max\left[A[i], A[i] + V_{i-1}\right] \\
V_i^* &= \max\left[V_{i-1}^*, V_i\right]
\end{align*}
\]

Example:

\[
\begin{array}{cccccccccccc}
-3 & 7 & 2 & -8 & +9 & -1 & 3 & -15 & 4 & -7 \\
\hline
V_i & 0 & -3 & 7 & 9 & 1 & 10 & 9 & 12 & -3 & 4 & -3 \\
V_i^* & 0 & 0 & 7 & 9 & 9 & 10 & 10 & 12 & 12 & 12 & 12 \\
\end{array}
\]

\[
\uparrow \quad \uparrow
\]

Running Time: \( O(n) \).

Why is it correct?
More details on Divide-and-Conquer Algorithm
Partition $A$ in two equal halves. Assume $n = 2^k$, $k > 0$.

$$A = \begin{array}{c}
1 & \frac{n}{2} & \frac{n}{2}+1 & n \\
\end{array}$$

Note: Our objective is to find $i \leq j \leq n$

$s.t. \quad \sum_{k=i}^{j} A[i]$ is maximized.

$i \neq j$ must satisfy exactly one of the following

Case 1. $1 \leq i \leq j \leq \frac{n}{2}$ [Both $i \neq j$ in left half]

Case 2. $\frac{n}{2}+1 \leq i \leq j \leq n$ [Both $i \neq j$ in right half]

Case 3. $1 \leq i \leq \frac{n}{2}$ and $\frac{n}{2}+1 \leq j \leq n$

[$i$ in left half, $j$ in right half]

We do not know where $i \neq j$ are, so we try all three cases.

For Case 1 & 2, we appeal to recursion.

Case 3 has a special structure and explained on the next page.
Case 3:

In Case 3, we know $1 \leq i \leq \frac{n}{2}$ and $\frac{n}{2} + 1 \leq j \leq n$.

We are interested to find what values of $i + j$ will maximize the sum

$$\sum_{k=i}^{j} A[k].$$

Note:

$$\sum_{k=i, \frac{n}{2} \leq k \leq j} A[k] = \sum_{k=i}^{\frac{n}{2}} A[k] + \sum_{k=\frac{n}{2}+1}^{j} A[k].$$

Thus, we want to find index $i$ such that $A[\frac{n}{2}] + \cdots + A[j]$ is maximum.

Both of these can be found by performing a simple scan in their respective subarrays.
Analysis of Divide & Conquer algorithm.

Our solution consists of the best solution among the best solution obtained from Cases 1, 2, 3.

If \( T(n) \) represents the time to solve the problem on an array of size \( n \), then the analysis of the above algorithm can be done as follows.

\[
T(1) = 1.
\]

\[
T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n)
\]

\[
\uparrow \quad \uparrow \quad \uparrow
\]

Time for Case 1 Time for Case 2 Case 3

\[
= 2 \cdot T\left(\frac{n}{2}\right) + O(n)
\]

\[
= 2^2 \cdot T\left(\frac{n}{2^2}\right) + 2O\left(\frac{n}{2}\right) + O(n)
\]

\[
= 2^3 \cdot T\left(\frac{n}{2^3}\right) + 2^2 O\left(\frac{n}{2^2}\right) + 2O\left(\frac{n}{2}\right) + O(n)
\]

\[
= 2^l \cdot T\left(\frac{n}{2^l}\right) + 2^{l-1} O\left(\frac{n}{2^{l-1}}\right) + 2^{l-2} O\left(\frac{n}{2^{l-2}}\right) + \ldots + O(n)
\]

\[\text{--- (I)}\]
Since $n = 2^l$ (by assumption),

$\quad \Leftrightarrow \quad l = \frac{\log n}{\log 2}$.

Thus Equation I can be expressed as

$$T(n) = 2^l T(1) + O(n) + O(n) + \ldots + O(n)$$

$\quad \underbrace{\log n \text{ times}}_{\log 2}$

Since $T(1) = 1$, and $n = 2^l$, we obtain

$$T(n) = O(n \log n).$$