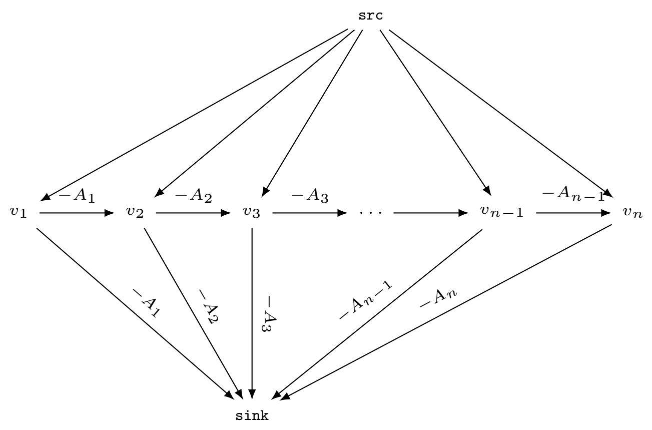
Maximum Subarray Sum

Pratik Deoghare

June 12, 2025



Given an array A of length n construct a directed graph as follows.
Add vertices

- Add special vertices src, $sink$.
- Add one vertex v_i for each A_i.

$$V := \{src, sink\} \cup \{v_i \text{ for } 1 \le i \le n\}$$

Add edges

- Add edge of weight 0 from src to each v_i.
- Add edge of weight $-A[i]$ from v_i to $sink$.
- Add edge of weight $-A[i]$ from v_i to v_{i+1}.

$$E := \{(src, v_i, 0)\} \cup \{(v_i, sink, -A[i]\} \cup \{(v_i, v_{i+1}, -A[i])\}$$

We find shortest path from src to $sink$. -1 times cost of that path is our answer.
Now, you might be thinking Kadane's algorithm is just a few lines and this one will need a lot of code. Let me show you how little code is needed. This

is a DAG so we can find the shortest path by relaxing edges going out of vertices in topological order. We don't need to implement topological sort because we already know the order i.e. $sink, v_n, v_{n-1}, \ldots, v_0, src$.

```
MAX-SUBARRAY-SUM($A$)
1  d[v] = 0 for $v \in V$
2  d[$v_n$] = $-A[n]$
3  for $v_i = v_{n-1}, \ldots, v_1, src$
4      for $v \in Adj[v_i]$
5          RELAX($v_i$, v, weight($v_i$, v))
6  return -min{d[v] for $v \in V$ }
```

We know there are only two outgoing edges from the vertices being relaxed. We replace RELAX by its implementation for each of those edges to get the code below.
Substitute adjacent vertices.

```
MAX-SUBARRAY-SUM($A$)
1  d[v] = 0 for $v \in V$
2  d[$v_n$] = $-A[n]$
3  for $v_i = v_{n-1}, \ldots, v_1, src$
4      RELAX($v_i$, $v_{i+1}$, weight($v_i$, $v_{i+1}$))
5      RELAX($v_i$, sink, weight($v_i$, sink))
6  return -min{d[v] for $v \in V$ }
```

Substitute weights.

```
MAX-SUBARRAY-SUM($A$)
1  d[v] = 0 for $v \in V$
2  d[$v_n$] = $-A[n]$
3  for $v_i = v_{n-1}, \ldots, v_1, src$
4      RELAX($v_i$, $v_{i+1}$, $-A[i]$)
5      RELAX($v_i$, sink, $-A[i]$)
6  return -min{d[v] for $v \in V$ }
```

Substitute implementation of $RELAX$.

```
MAX-SUBARRAY-SUM($A$)
1  d[v] = 0 for $v \in V$
2  d[$v_n$] = $-A[n]$
3  for $v_i = v_{n-1}, \ldots, v_1, src$
4      d[$v_i$] = min(d[$v_i$], d[$v_{i+1}$] $-A[i]$)
5      d[$v_i$] = min(d[$v_i$], d[sink] $-A[i]$)
6  return -min{d[v] for $v \in V$ }
```

This in turn is equivalent to the following final code since $d[sink] = 0$.

```
MAX-SUBARRAY-SUM($A$)
1  d[v] = 0 for $v \in V$
2  d[$v_n$] $= -A[n]$
3  for $v_i = v_{n-1}, \ldots, v_1, src$
4        d[$v_i$] = min($-A[i]$, d[$v_{i+1}$] $-A[i]$)
5  return -min{d[v] for $v \in V$ }
```

Here is golang implementation

```
func maxSubarraySum(A []int) int {
    n := len(A)
    d := make([]int, n)
    d[n-1] = -A[n-1]
    for i := n-2; i >= 0; i-- {
        d[i] = min(-A[i], -A[i] + d[i+1])
    }
    return -slices.Min(d)
}
```

We could maintain the global min while looping instead of computing it at the end using `slices.Min`.

```
func maxSubarraySum(A []int) int {
    n := len(A)
    d := make([]int, n)

    d[n-1] = -A[n-1]
    minn := d[n-1]
    for i := n-2; i >= 0; i-- {
        d[i] = min(-A[i], -A[i] + d[i+1])
        minn = min(minn, d[i])
    }
    return -minn
}
```

Looks exactly like Kadane's algorithm, doesn't it?