MAXIMUM-SUBARRAY SUM

Divide-and-conquer (recursive approach)

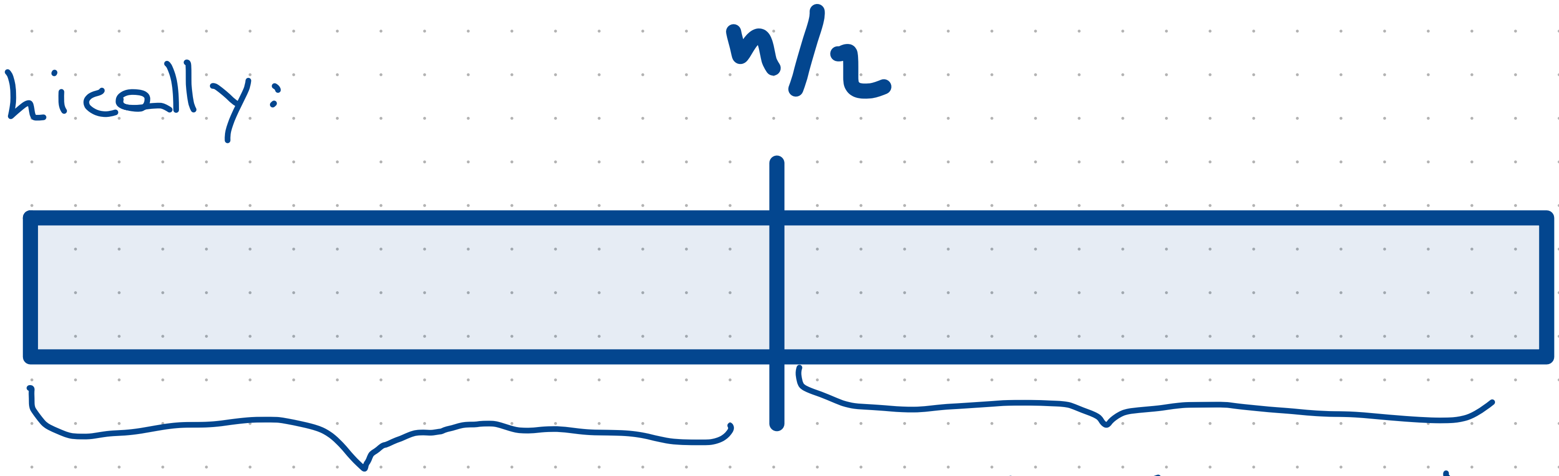
Given array a: $a_1, a_2 \dots a_n$ ($n = 2^k$ for large k)

We do the following:

1. We compute the maximum-subarray sum in $a_1 \dots a_{\frac{n}{2}}$. Let L_1 be the result.
2. We compute the maximum-subarray sum in $a_{\frac{n}{2}+1} \dots a_n$. Let L_2 be the result.
3. We compute the maximum subarray sum of $a_i \dots a_j$ such that $i \le \frac{n}{2}$, $j > \frac{n}{2}$. Let M be the result.

FINAL ANSWER: $\max(L_1, L_2, M)$

Graphically:



$C_1 :=$ Maximum subarray sum on "the left"

$C_2 :=$ Maximum subarray sum on "the right"

$M :=$ Maximum subarray sum starting on "the left" and ending on "the right".

<u>Questions:</u> how to compute C_1, C_2 and M?

Let's start with M.

Example $\quad -10 \quad 2 \quad 5 \;|\; 3 \quad 1 \quad -8$

M is maximum subarray sum starting on "the left" and ending on "the right". How do we compute it?

Let's compute the prefix-sum array on "the right"

$$3 \quad 4 \quad -4$$

And the suffix-sum array on "the left"

$$5 \quad 7 \quad -3$$

M = maximum element in the prefix-sum array
+ " " " suffix-sum array

In this case $4 + 7 = 11$

How do we compute prefix/suffix sum array?

We illustrate the algorithm to compute the prefix/suffix sum array on $b_0 \dots b_n$. For our problem we compute the prefix sum on $a_{n/2+1} \dots a_n$ and the suffix sum on $a_1 \dots a_{n/2}$.

prefix[0] $\leftarrow b_0$	suffix[0] $\leftarrow b_n$
for $i = 1 \dots n$	for $i = 1 \dots n$
prefix[i] $\leftarrow$ prefix[i-1] $+ b_i$	suffix[i] $\leftarrow$ suffix[i-1] $+ b_{n-i}$

It can be computed in:

$O(n)$ compute "prefix"

$O(n)$ compute "suffix"

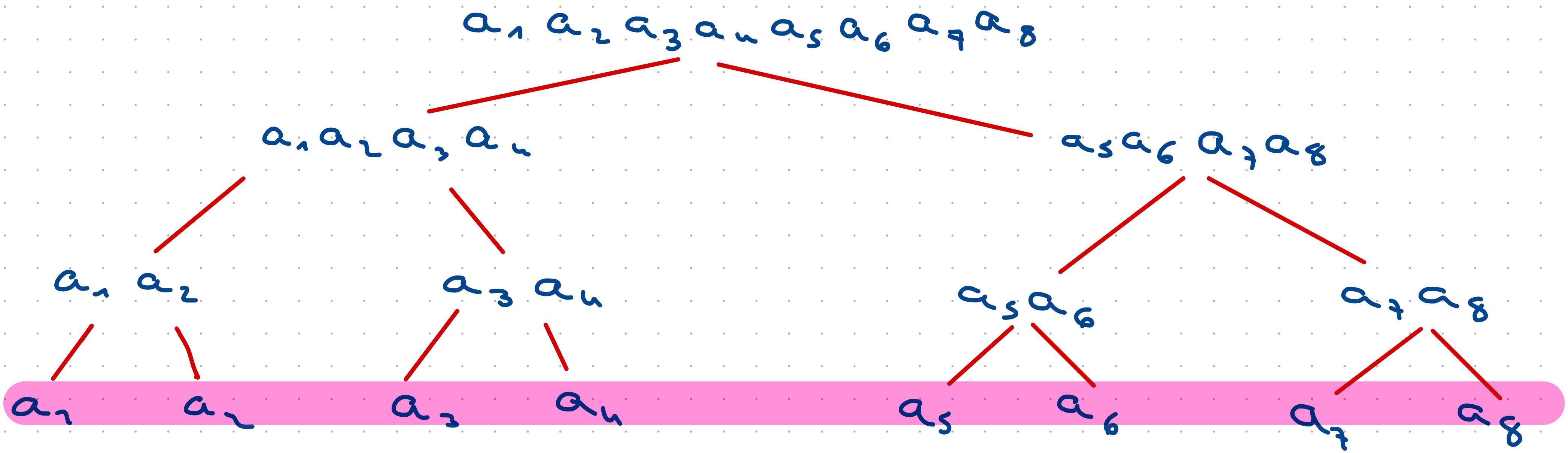
$O(n)$ find max in prefix/suffix

$O(1)$ sum of maximum suffix and maximum prefix

$O(n)$ Total

How do we compute C_1 and C_2? RECURSIVELY

The idea of recursion is that we "trust" our algorithm: it will call itself multiple times until it hits a base case. Let's look at it graphically (in red the recursive calls)



In pink the base-cases

In pseudo-code our algorithm is the following:

ALGO(a)

if a has length 1: return $\boxed{\max(0, a_1)}$ → Why?

else

$L_1 \leftarrow$ ALGO($a_1 \dots a_{n/2}$)

$L_2 \leftarrow$ ALGO($a_{n/2+1} \dots a_n$)

$M \leftarrow \dots$ → we compute it with prefix/suffix sum arrays as explained above

return $\max(L_1, L_2, M)$

RECURSION: we "trust" that ALGO will eventually hit the base case and return the correct solution

Total runtime $T(n)$? Here we prove only for $n = 2^k, k \geq 0$
The generalization is technical (but similar in spirit).

$$T(n) = \underbrace{2T\left(\frac{n}{2}\right)}_{\text{recursive calls}} + \underbrace{c \cdot n}_{\text{computation of } m}$$

$$= 2\left(2T\left(\frac{n}{4}\right) + c\,\frac{n}{2}\right) + cn = 4T\left(\frac{n}{4}\right) + 2cn$$

$$= 4\left(2T\left(\frac{n}{8}\right) + c\,\frac{n}{4}\right) + 2cn = 8T\left(\frac{n}{8}\right) + 3cn$$

$$= \ldots$$

$$= 2^k\, T\left(\frac{n}{2^k}\right) + k\,cn$$

That's the informal idea, now we are going to prove it (via induction)

$$\boxed{\begin{array}{ll} T(1) = \tilde{c} \quad (1) & \underline{\text{Claim}} \quad T(2^k) = 2^k\, T(1) + k c 2^k \quad (3) \\ T(n) = 2\,T\left(\frac{n}{2}\right) + cn \quad (2) & \end{array}}$$

Induction over k.

$\boxed{\text{BC}, k=0}$ $\quad T(2^0) = T(1) \overset{(1)}{=} \tilde{c} \qquad T(2^0) \overset{(3)}{=} 2^0\, T(1) + 0 \cdot c \cdot 2^0 = T(1) = \tilde{c}$

$\boxed{\text{IH}}$ Claim holds for a k

$\boxed{\text{IS}}$
$$\begin{aligned} T(2^{k+1}) &\overset{(2)}{=} 2\,T\left(\frac{2^{k+1}}{2}\right) + c \cdot 2^{k+1} \\ &= 2\,T(2^k) + c \cdot 2^{k+1} \\ &\overset{\text{IH}}{=} 2\left[2^k\, T(1) + k c 2^k\right] + c \cdot 2^{k+1} \\ &= 2^{k+1}\, T(1) + k c 2^{k+1} + c \cdot 2^{k+1} \\ &= 2^{k+1}\, T(1) + c \cdot 2^{k+1} (k+1) \end{aligned}$$

$\blacksquare$

<u>My suggestion:</u> Try to implement it yourself and then check the solution on the GitHub of the exercise session!