Proof: $\underline{\text{let's}}$ say, 1

j is at index K

and $\hat{A}[i] + \hat{A}[j'] = T$ where $\underline{i < j' < K}$

as. $\hat{A}[i] + \hat{A}[K] \geq \hat{A}[i] + \hat{A}[j']$

$\geq T$

the alogorithm will decrement j by 1.

and this process will repeat, until $j = j'$ OR

$\hat{A}[j] = \hat{A}[j']$

---

from this subclaim, we can prove the claim (1)

So, we have reached i. implies, for all $i = 0 \to i-1$

the answer doesn't exist.

So, $\boxed{\text{decreasing } j \text{ is the only option}}$

Case (ii): If $T' < T$, $[T' = \hat{A}[i] + \hat{A}[j]]$
where $i \leq j$

by the similar argument, we can say that.

$T'' = \hat{A}[i+1] + \hat{A}[j] \geq \hat{A}[i] + \hat{A}[j]$

$\geq T'$

So, T'' is a good candidate for T.

5/10

Rough work

The only subarrays we haven't considered ~~&~~ are subarrays that includes the mid element. or A[mid]

To consider this, we wrote a function ~~that~~ maxOver (A, mid)

we are keeping 4 counters

Initialization

maxsum1 = 0	keep track of maximum sum possible starting at mid+1 → n-1	in right side
current sum1 = 0	keeps track of current sum starting at mid+1 → n-1	
max sum 2 = 0	same but for left side.	
current sum 2 = 0	starting from mid-1 to 0	

You have to compare these sums with A[mid].

in the first loop, if we are at index K.

You have merely described what your algorithm is doing, and not explained why it is correct.

then $current sum1_K = A[mid+1] + \dots A[K]$

$maxSum1_K = \max_{i=mid+1 \to K} (currentSum1_i)$

Similarly in the 2nd loop,

$K \leq mid-1$

$current sum2_K = A[mid-1] \dots A[K]$

$max sum 2_K = \max_{i=mid+1 \to K} (currentSum2_i)$

Why? Just saying it won't make it so!

So, A[mid] + maxSum2_0 + maxsum 2_{n-1} will give us the maxsubarray for subarrays that span across (mid).

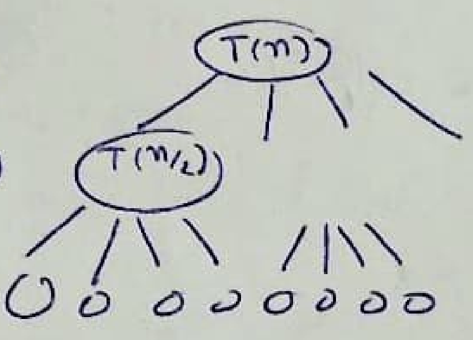
Hence our algo correctly returns the max sub...

Part B

For questions in part (B), you have to write your answer with a short explanation in the space provided below. For numerical answers, the following forms are acceptable: fractions, decimals, symbolic e.g.: $\binom{n}{r}$, ^nP_r, $n!$ etc.

(1) (a) $T(n) = 4T(n/2) + 10$

All the $(\log n)$ are equivalent to $\log_2 n$ unless specified



$4T(\frac{n}{2})$ — 10

$16T(\frac{n}{4})$ — 40.

$\log_2 n$ steps / levels

ith level $4^i T(\frac{n}{2^i}) + 10 + 40 \dots \quad 10 \times 4^{(i-1)}$

(2) So.

$$T(n) = 4T(\tfrac{n}{2}) + 10$$
$$= 16\,T(\tfrac{n}{4}) + 10 + 40$$
$$= 4^i T(\tfrac{n}{2^i}) + 10 + 40 + \quad 10 \times 4^{(i-1)}$$

Putting $i = \log_2 n$ or $2^i = n$

$$= 4^{\log_2 n} T(1) + 10(1 + 4^1 + 4^2 + \cdots 4^{\log_2 n - 1})$$

$$= n^{\log_2 4}\, 1 + 10 \cdot \frac{4^{\log_2 n} - 1}{4-1}$$ (assuming $T(1) = 1$)

$$= n^2 + \frac{10}{3}\,\frac{n^2 - 1}{1}$$

$$= \frac{3n^2 + 10n^2 - 10}{3}$$

$\therefore T(n) = \Theta(n^2)$ is a reasonable estimate

Rough work

cont

(b) We will prove the correctness of MaxSubArray (A) using ~~induction on length of the Array A~~ induction on length of the Array A

n = length (A)

Base case: If ($n <= 0$) the max value of vacuous subarray is 0.

if ($n == 1$) the only element will be a good candidate

So, algorithm handles them correctly

Termination:

before every recursive call

we are extracting the mid element and then passing the two halves to the function.

So, at every call the arraysize is atleast decreasing by 1, as mid is excluded, (Precisely, it will be called with two arrays with of size $\left(\frac{n-1}{2}\right)$)

Induction hypothesis: if A = B ++ A[mid] ++ C (++ means array concatenation.)

MaxSubarray (B) and MaxSubArray(C) returns correctly.

Induction step: So, we know the maximum subarray in B and C.

Again RTP, $T(n) = \Omega(n^2)$

or $\exists c_0', n''$ $T(n) \geq c_0' n^2 \quad \forall n > n_0''$

Base case $T(n'') \geq c''$
given

also, $T(n'') \geq c_0 n''^2$

$c_0 n''^2 \leq c''$

$c_0 \leq \frac{c''}{n''^2}$

---

(5) Induction hypothesis: $T(\frac{n}{2}) \geq c_0 \frac{n^2}{4}$

Now, $T(n) = 4T(\frac{n}{2}) + 10$

$\geq 4c_0 \frac{n^2}{4} + 10$

$\geq c_0 n^2 + 10$

$\geq c_0 n^2$

Hence,

$\therefore T(n) = \Omega(n^2)$ ✓

$\therefore T(n) = \Theta(n^2)$

10/15

work

4(c) So. $T(n) = O(n\log n) + n + 7t + 3$

t denotes the number of times the while loop runs over the array.

at each iteration we are either decreasing j by 1 or increasing i by 1 or returning the answer.

So, $(j-i)$ is decreasing by atleast 1

and at start, $j-i = (n-1)$

and at the termination $j-i = -1$

So, $\boxed{t \leq n+1}$

$\exists c, n_0$

$\therefore T(n) \leq cn\log n + n + 7(n+1) + 3 \quad \forall n \geq n_0$

$\therefore T(n) \leq (c+1)n\log n - (n\log n - n - 7n - 10)$

$= (c+1)n\log n - (n\log n - (8n+10))$

$\leq (c+1)n\log n \quad \forall n > 2^{100}$ [as $2^{100}.100 - 8 2^{100} - 10 > 0$]

So, $T(n) \leq c' n\log n \quad \forall n > \max(n_0, 2^{100})$

$c' = (c+1)$

So, $T(n) = O(n\log n)$

10/10

13)

Again RTP: $T(n) = \Omega(n \log n)$

$\exists c_1, n_1$

or $T(n) \geq c_1 n \log n \quad \forall n \geq n_1$

~~Now T(n)~~

Base case: $T(n_1) > c_1'$ given

~~also T(n)~~

So, $\boxed{c_1 < \frac{c'}{n_1 \log n_1}}$

---

(14)

Induction hypothesis: $T(\frac{n}{2}) \geq c_1 \frac{n}{2} \log \frac{n}{2}$

15/15

Induction step:

$T(n) = 2T(\frac{n}{2}) + 6n$

$\geq 2c_1 \frac{n}{2} \log \frac{n}{2} + 6n$

$\geq c_1 n(\log n - 1) + 6n$

$\geq c_1 n \log n + 6n - c_1 n$

$\geq c_1 n \log n + n(6 - c_1)$

If we take c_1 as 1

$T(n) \geq 1 \cdot n \log n + 5n$

$\geq 1 \cdot n \log n \quad \forall n \geq 1 = n_1$

$\therefore$ Proved $T(n) = \Omega(n \log n)$

9

(3) Now,

RTP, $T(n) = O(n^2)$

$\exists c, n_0', d$

or $T(n) \leq cn^2 - d$ $\forall n \geq n_0'$

Base case $n = n_0'$

$\therefore T(n_0') \leq c'$. also, $T(n_0') \leq c(n_0')^2 - d$

So, $c \geq \frac{c'+d}{(n_0')^2}$.

Induction hypothesis $T\left(\frac{n}{2}\right) \leq c\frac{n^2}{4} - d$

Induction step: $T(n) = 4T\left(\frac{n}{2}\right) + 10$.

$\therefore$ $T(n) \leq 4 \cdot c\frac{n^2}{4} + 10 - 4d$

$= cn^2 + 10 - 4d$

---

(4)

$= cn^2 ~~-(d-10)~~ - (4d - 10)$

$\therefore$ $\leq cn^2$ (if $d > \frac{10}{4}$)

But this is not what you set out to prove.

Hence $T(n) = O(n^2)$

Now, the recurrence will be changed to [Not rough]

$$T(n, n) = 2n+2 + \max\left(T(n-1, n-1), \{T(n, n-1) + T(n-1, n)\}\right)$$

as both of the variable are of form n.

$$\therefore T(2n) = 2n + 2 + \max\left(T(2n-2), T(2n-1) + T(2n-1)\right)$$

~~as $T(2n-1) > T(2n-2)$~~

~~at worst case,~~ ~~[evident from Line ①]~~

~~$T(2n) = 2n+2+2T(2n-1)$~~

~~So, $T(2n) = 2T(2n-1) + 2n + 2$~~

~~$2T(2n-1) = 2^2T(2n-2) + 2(2n-1+2)$ .. ①~~

~~$= 2^2T(2n-2) + 2^2(2n-1$~~

~~2n step~~ Not required to solve

Explanation:

if we pass the array A, B. (length (A) = a, length (B) = b)

if last element of they are equal. it will call the function with A and B but both of them have last element removed. $\{n = a, m = b\}$ so, in that case, $T(n, m) = cn + d + T(n-1, m-1)$.

[This algorithm returns the maximum number when everything in it is negative]

returns
[-1, -2, -3 -4] → -1

5. (a) Max Sub Array (A) : T(n) n = length (A)

n = length (A) // n

If (AllNegative (A)) return maxVal (A) // $T'(n) + T''(n)$

If (n == 1) return A[0] // 1

If (n <= 0) return 0 // 1

mid = $\frac{n}{2}$ // 1

25/35

~~30/30~~

B = [A[0], A[1] . . A[mid-1]] // $\frac{n}{2}$

C = [A[mid], A[mid+1], , A[n-1]] // $\frac{n}{2}$

maxB = MaxSubArray (B) // $T(\frac{n}{2})$

(18)

maxC = MaxSubArray (C) // $T(\frac{n}{2})$

maxOver = CalcMax (mid, A) $T'''(n)$

return max ({maxB, maxC, maxOver}) 1 #*

//(reference for part c)

AllNegative (A) : $T'(n)$

n = length (A) // n

for i = 0 → n-1 : // n

- if A[i] >= 0 : // n

- - return false // 1

return true. // 1

maxVal (A) $T''(n)$

n = length (A) // n

if (n = 0) return 0. // 1

curMax = A[0] // 1

for i = 1 → n-1 { // n

curMax = max (A[i], curMax) // n

}

return curMax // 1

11

#* (max) of 3 elements returns pairwise max.

Rough work

So, our only option will be decreasing ~~(n-1)~~ by 1 → $j-1$

~~or Now~~ $\cancel{T'' = A[0] + A[n-2]}$

$\cancel{\leq A[0] + A[n-1]}$

$\cancel{\leq T'}$

~~So, T'' might be a good candidate~~ for T.

Now. $T'' = A[i] + A[j-1] \leq A[i] + A[j]$

$\leq T'$

So T'' might be a good candidate for T

Observation : $\boxed{\text{Decreasing } i \text{ by 1 was also a good candidate}}$

But that's not an option

claim ① if we have reached some index K after increasing i
i.e. $i = K$ (now)
then for $i = 0, K-1$, the answer doesn't exist or
(any index from $0, K-1$ can't be the answer)

Subclaim ⓐ if i is one of the index s/.
Algorithm if $A[i] + A[j] = T$ $[i \leq j]$
then, we will find the answer without increasing i
~~answer~~ i.

Rough work

then they will still be present in $\hat{A}$ (maybe in different index)

So. Again $\hat{A}_{f(i)}$, $\hat{A}_{f(j)}$ will sum exactly upto T.

So, Sorting A to get $\hat{A}$ and then trying to find the answer won't change the answer to this question

Now. $\hat{A}[0] \leq \hat{A}[1] \leq \hat{A}[2] \ldots \leq \hat{A}[n]$ ... (1)

So, Now. our pointers i, j are set to 0 and $n-1$ respectively. where $(i \leq j)$ ~~as (0, n-1)~~

Now, at ~~the first~~ some iteration we are calculating

Case 1, $T' = (\hat{A}[i] + \hat{A}[j])$. and comparing it with T.

$\hookrightarrow (T' = (\hat{A}[i] + \hat{A}[j]))$

So, if $(T' = T)$ we are done as $\hat{A}[i] + \hat{A}[j] = T$

So, we return yes

Case 2 $\boxed{T' > T}$

Now. we know ~~A[0] + A[n-1] is not the answer~~

$T' = \hat{A}[i] + \hat{A}[j] > T$

but we also know

for any $k > i$, also,

$\hat{A}[k] + \hat{A}[j]$ ~~is not the answer~~ $> T$

as. $\hat{A}[k] + \hat{A}[j] \geq \hat{A}[i] + \hat{A}[j] = \hat{A}[i] + \hat{A}[j]$
$\geq T'$
$> T$

as $\hat{A}[k] \geq \hat{A}[i]$ when $k > i$

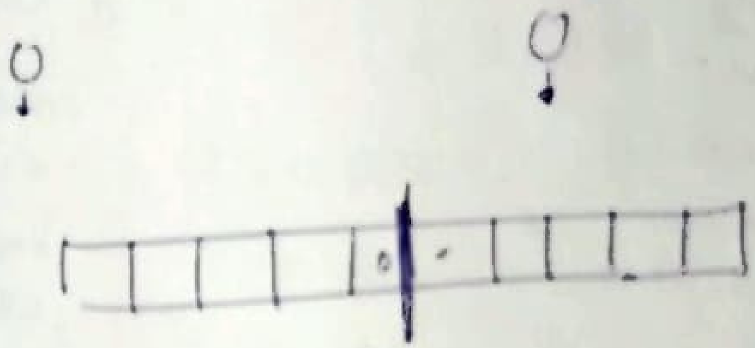
Rough page

Please don't evaluate

Rough work

$T(n) = 4T(n/2) + 10$

$T(n) = 2T(\frac{n}{2}) + 6n$



$T(n,m) = \max(T(n-1, m-1) , T(n, m-1) + T(n-1, m)$

$T(n) = 2T(2n-1)$

$4T($

Rough work

So. increasing i will give us a better chance of finding T.

Now, termination :

we run the loop until $i \leq j$.

$(\leq)$ matters as j might be equal to i.

and $\hat{A}[i] + \hat{A}[i] = 2\hat{A}[i]$
$= T$

might be possible.

if $i > j$, we will be going over similar pairs of i, j

But by claim ① we know the answer can't exist

So, we return "NO"

10/10

Rough work

4(b) Proof of correctness :-

checkSum (A,T) returns yes if two array elements (possibly same) sum upto exactly T, otherwise "No"

→ To solve it in specified time bound, we are first sorting the Array (A) in $O(n \log n)$ time using given MergeSort function.

So, we will specify the sorted A as $\hat{A}$ (i.e. $\hat{A}$ = MergeSort(A

Observation : If checkSum (A,T) returns yes. checkSum ($\hat{A}$, T) returns also.

Proof. checkSum (A,T) = Yes mean two elements in A (possibly same) sum up exactly T say A_i, A_j

when they are not equal,

~~T(n, m) will~~

the function will call $LCS(A[0:a-2], B)$

and $LCS(A, B[0:b-2])$

and return the max.

~~if A~~ as A has a elements

$A[0:a-2]$ will have $a-1$ "

Similarly $B[0:b-2]$ will have $b-1$ elements

So, $T(n,m) = cn+d + T(n-1,m) + T(n,m-1)$

So, $$T(n,m) = cn+d + \max[T(n-1,m-1), T(n-1,m)+T(n,m-1)]$$

10/10

Rough work

(2) (a) $LCS(A,B) : T(\overset{a}{\overset{\uparrow}{n}}, \overset{b}{\overset{\uparrow}{m}})$

A [i : j] denotes [A[i], ... , A[j]]

A [0: 2]
[A[0], A[1], A[2]]

The question specifically stated that the algorithm should return a sub-sequence!

a = Length (A) // n

b = length (B) // m

if (a == 0) return 0 // 1

if (b == 0) return 0. // 1

Notice, A[0: -1] is valid array of length 0. (or an empty array)

if A [a-1] == B [b-1]

return 1+ LCS(A[0: a-2], B[0: b-2]) //
T(n-1, m-1)

5/20

else.

return max (LCS (A [0: a-2], B), LCS(A, B[0:b-2])

// T(n, m-1) + T(n-1, m)

Consider n = a, m = b

(c) $T(n,m) = \cancel{\quad} \; n+m+2 + \max\{(T(n-1, m-1)), (T(n, m-1) + T(n-1, m))\}$

~~[illegible]~~ T(0) = 1.

without loss of generality

now. ~~[illegible]~~ WLOG, $n \geq m$.

So, ~~[illegible]~~

$T(m,m) > \boxed{T(n,m)} \geq T(n,n)$

PTO

Rough work

$T'''(n) = 2n + 3\frac{n}{2} + 3\frac{n}{2} + 4 = 5n+4 = O(n)$

7/10

$T''(n) = 4n+2 = O(n)$

$T'(n) = 3n+2 = O(n)$

So, $T(n) = n + T''(n) + T'(n) + 3 + \frac{n}{2} + \frac{n}{2} + 2T(\frac{n}{2}) + T'''(n) + 1$

$= 2T(\frac{n}{2}) + \underbrace{cn + d + 3[O(n)]}_{T''''(n)}$ $\quad \because T''''(n) = 3[O(n)] + cn + d = O(n)$

$= 2T(\frac{n}{2}) + O(n^1) = aT(\frac{n}{b}) + O(n^d)$

where $a=2, b=2, d=1$

from master theorem,

$\boxed{T(n) = O(n \log n)}$ [as $\log_b a = d = 1$]

Missing: Statement of the theorem.

---

4. (a)

checkSum (A, T) $\quad \underline{T(n)}$ Time taken for reference to (4C)

$j-- \equiv j = j-1$	
$i++ \equiv i = i+1$	

$\hat{A}$ = MergeSort (A) // $O(n \log n)$

n = length ($\hat{A}$) // n

i = 0, // 1

j = n-1, // 1

while (i ≤ j): // t

20/20

if ($\hat{A}$[i] + $\hat{A}$[j]) == T // t

return "Yes" // t // Programme terminates here

else if $\hat{A}$[i] + $\hat{A}$[j] > T // t

j--; // t

else if $\hat{A}$[i] + $\hat{A}$[j] < T // t

i++; // t

// 1

"N..."

(11)

Now, RTP, $T(n) = O(n\log n)$

$\exists c_0, n_0$

or $T(n) \leq c_0 n\log n \quad \forall n \geq n_0$

Base case $T(n_0) = c'$ (given)

and also, $T(n_0) \leq c_0 n_0 \log n_0$

$\therefore c_0 n_0 \log n_0 \geq c'$

$$\boxed{c_0 > c'/n_0\log n_0}$$

(12) hypothesis

Induction: $T\left(\frac{n}{2}\right) \leq c_0 \frac{n}{2} \log \frac{n}{2}$.

$\therefore T(n) = 2T\left(\frac{n}{2}\right) + 6n$

$\Rightarrow T(n) \leq 2 . c_0 \frac{n}{2} \lg \frac{n}{2} + 6n$

$= c_0 n \log n/2 + 6n$

$= c_0 n(\log n - 1) + 6n$

$= c_0 n \log n + (6 - c_0) n$

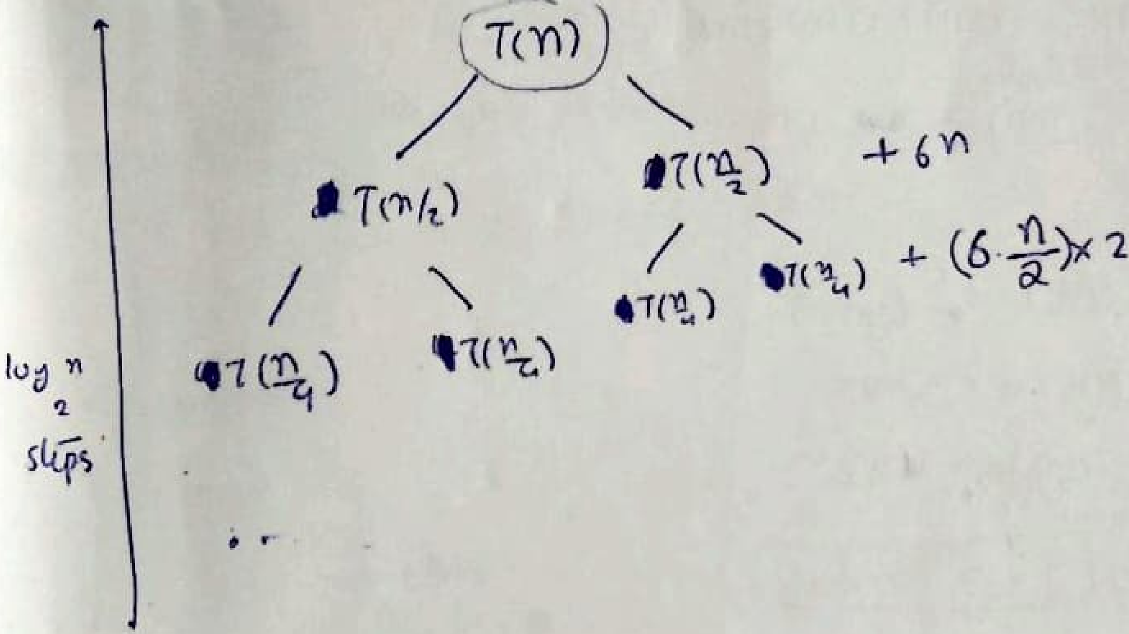
$= c_0 n \log n - (c_0 - 6)n$

$\geq c_0 n \log n$ [if $c_0 > 6$]

What is it that you want to prove?

(b)

(9) $T(n) = 2T(n/2) + 6n$.



(10) Now, $T(n) = 2T(\frac{n}{2}) + 6n$

$= 4T(\frac{n}{4}) + \underbrace{6n + 2\cdot 6\frac{n}{2}}_{2\ \text{terms}}$

$= 8T(\frac{n}{8}) + \underbrace{6n + 2\cdot 6\frac{n}{2} + 4\cdot 6\cdot\frac{n}{4}}_{3\ \text{terms}}$

$= 2^i T\frac{n}{2^i} + \underbrace{6n + 6n + \ldots}_{i\ \text{terms}}$

now putting $i = \log_2 n$ or $2^i = n$

$= 2^{\log_2 n} T(\frac{n}{2^{\log_2 n}}) + 6n(\log_2 n)$

$= n\,T(1) + 6n\log n$

$= n + 6n\log n$ assuming $(T(1) = 1)$

$\therefore T(n) = \Theta(n\log n)$ is a reasonable estimate

(19)

CalcMax (mid, A) $T'''(n)$

n = length(A)   // n

~~[B[ ] ... empty array of size (n)]~~

~~for i (mid+1)~~   // 1

currentSum1 = 0.

maxSum1 = 0.   // 1

for i = (mid+1) → n-1   // n/2

currentSum1 += A[i]   // $\frac{n}{2}$

maxSum1 = max(maxSum1, currentSum1)   // $\frac{n}{2}$

currentSum2 = 0   1

maxSum2 = 0   1

(20) for i (mid-1) → 0.   // n/2

currentSum2 += A[i]   // $\frac{n}{2}$

maxSum2 = max(maxSum2, currentSum2)   // $\frac{n}{2}$

~~return A[mid] + currentSum1 + currentSum2~~

return A[mid] + maxSum1 + maxSum2   1

This will return the wrong answer if the only subarray with the maximum sum that contains A[mid], is [A[mid]].

(b) if all elements are negative, it first checks so, and returns the maximum of that array in O(n) time.

If not, it proceeds as usual, and takes max of null arrays as 0, as our answer will always be ≥ 0. If not all negative