分之治之

Lecture 05 — Divide and Conquer (Part 1)

COMPSCI 308 — Design and Analysis of Algorithms

Xing Shi Cai https://newptcai.gitlab.io

Duke Kunshan University, Kunshan, China

Jan-Mar 2026

Summary

ITA 2.3 Merge Sort

ITA 4.1 The Maximum-Subarray Problem

ITA 4.2 Strassen's Algorithm for Matrix Multiplication

Assignments[1]

Practice makes perfect!

📖 Required Readings:

- Section 2.3
- Section 4.1
- Section 4.2

✏️ Required Exercises:

- Exercises 2.3: 1–3.
- Exercises 4.1: 1–4
- Exercises 4.2: 1–5.

[1] 💣 Assignments will not be collected; however, quiz problems will be selected from them. (This includes both Readings and Exercises.)

ITA 2.3 Merge Sort

Divide and Conquer

Divide and Conquer is a common technique in designing algorithms:

- 🔪 Divide: Break the problem into smaller sub-problems.
- ⚔️ Conquer: Solve the smaller sub-problems.
- 🥗 Combine: Merge the solutions to create a solution for the original problem.

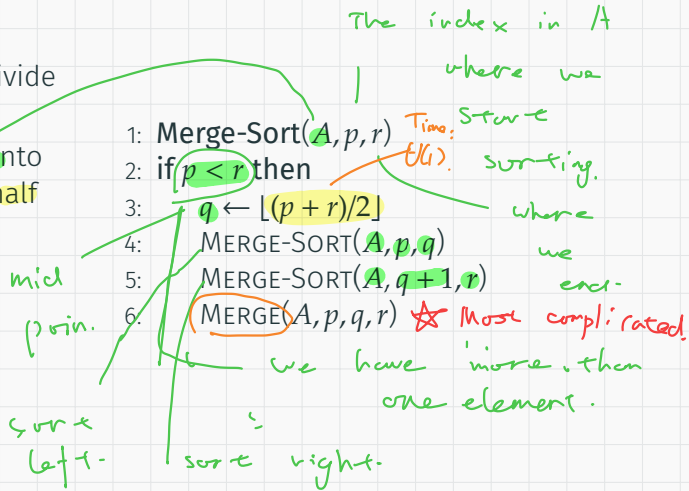
Figure 1: How to divide and conquer a big 🌱 pizza

Merge Sort

Merge Sort is a classic example of a divide and conquer algorithm.

- 🔪 Divide: Split the unsorted array into two subarrays, each containing half of the elements.
- ⚔️ Conquer: Sort each subarray.
- 🥗 Combine: Merge the two sorted subarrays.

```
1: Merge-Sort(A, p, r)
2: if p < r then
3:     q ← ⌊(p + r)/2⌋
4:     MERGE-SORT(A, p, q)
5:     MERGE-SORT(A, q + 1, r)
6:     MERGE(A, p, q, r)
```



Example of Merge Sort

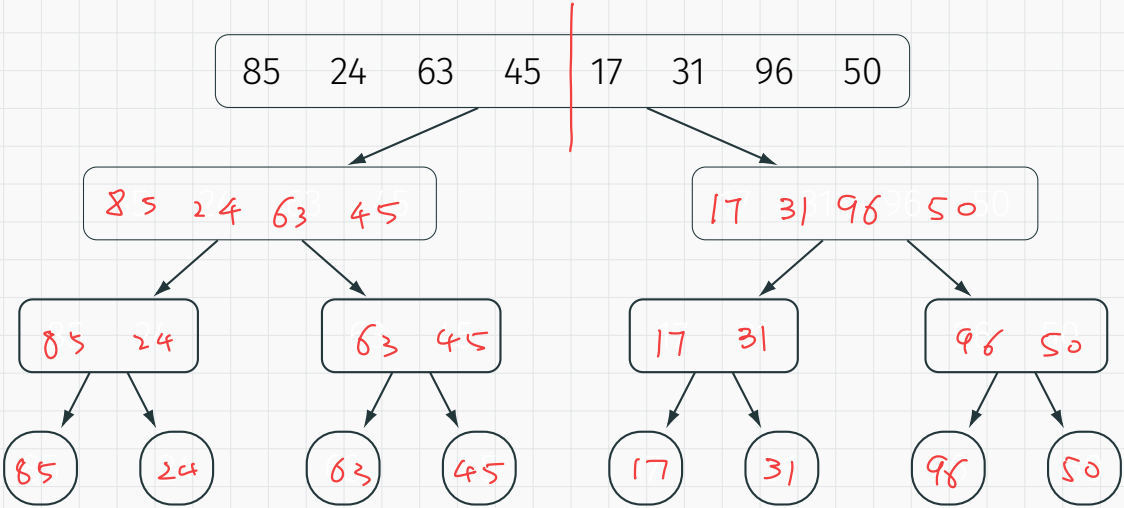


Figure 2: Divide

Example of Merge Sort

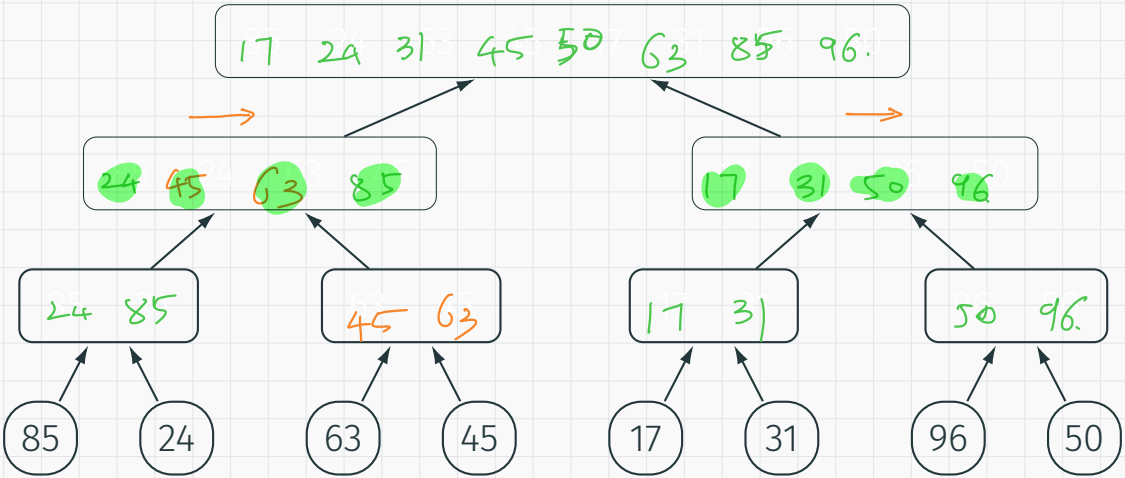


Figure 2: Combine

哨兵

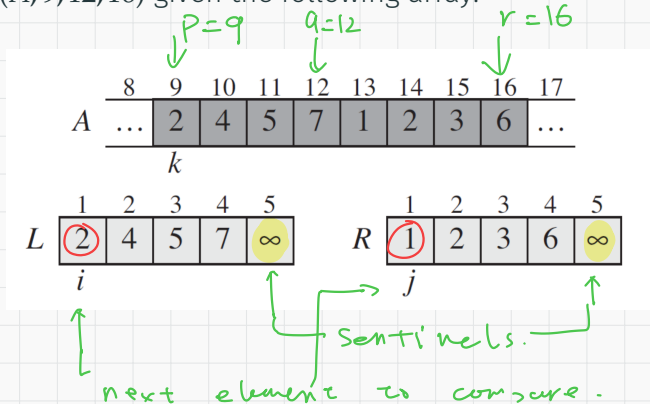
A sentinel is a soldier whose job is to guard something.

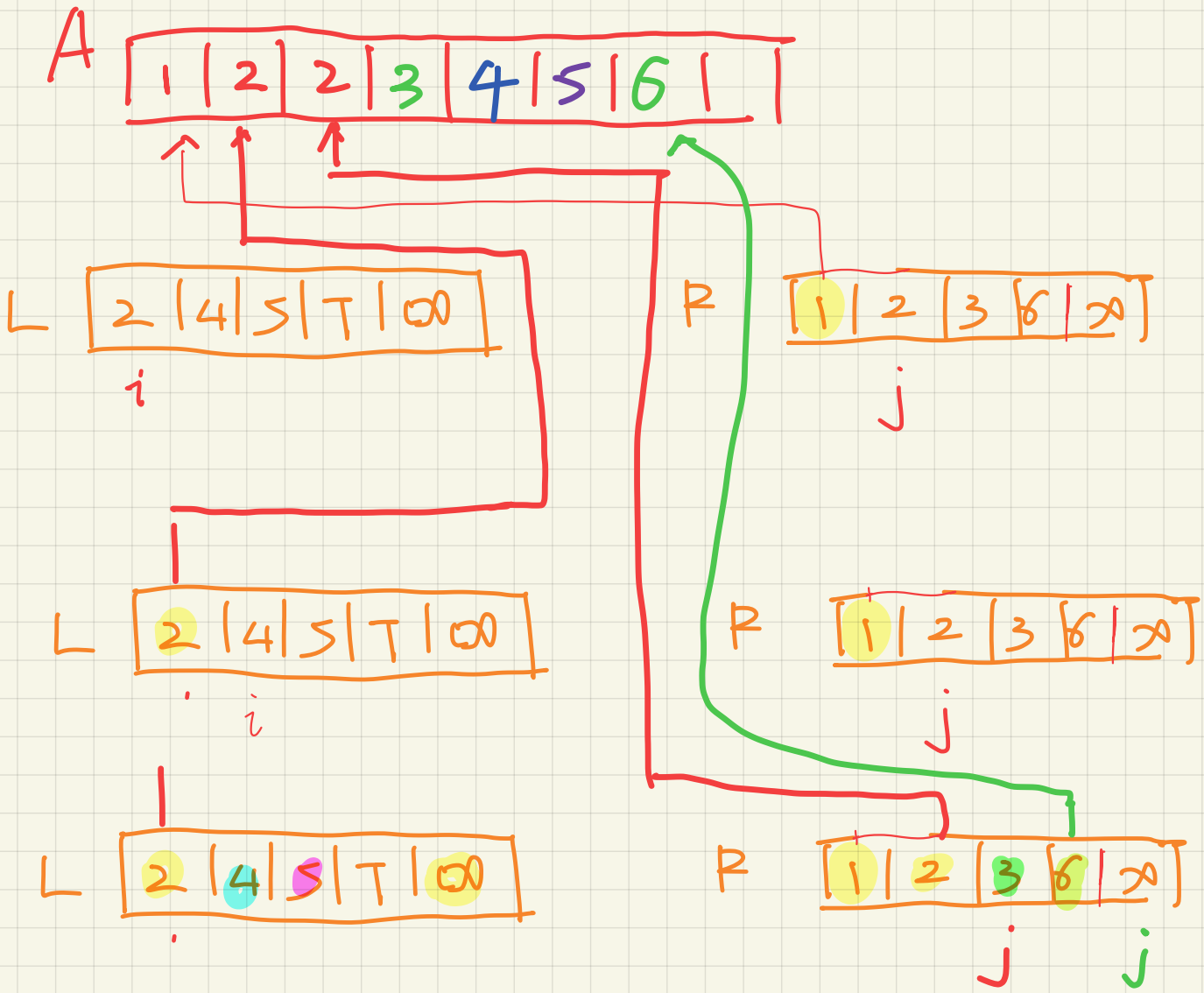
— Oxford Learner's Dictionary

Figure 3: The Carrot Sentinel!

Example of Merge

Let's run `Merge`$(A, 9, 12, 16)$ given the following array:




A
1 2 2 3 4 5 6
L
2 4 5 T ∞
i
R
1 2 3 6 ∞
j
L
2 4 5 T ∞
i
R
1 2 3 6 ∞
j
L
2 4 5 T ∞
i
R
1 2 3 6 ∞
j
j

Pseudocode of Merge Function

1: **Merge**(A, p, q, r)
2: $n_1 \leftarrow q - p + 1$
3: $n_2 \leftarrow r - q$
4: Let $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$ be new arrays
5: **for** $i \leftarrow 1$ to n_1 **do**
6: $\quad L[i] \leftarrow A[p + i - 1]$
7: **for** $j \leftarrow 1$ to n_2 **do**
8: $\quad R[j] \leftarrow A[q + j]$
9: $L[n_1 + 1] \leftarrow \infty$ ▷ A sentinel
10: $R[n_2 + 1] \leftarrow \infty$ ▷ Another sentinel

begin of left array
end of left
end of right
len of left
len of right.
length n_1+1
len of n_2+1
for the ∞ (sentiel).
copy left and right in L and R.

Pseudocode of Merge Function

```
1:  Merge(A, p, q, r)
2:  n_1 ← q − p + 1
3:  n_2 ← r − q
4:  Let L[1 ... n_1 + 1] and R[1 ... n_2 + 1] be
    new arrays
5:  for i ← 1 to n_1 do
6:      L[i] ← A[p + i − 1]
7:  for j ← 1 to n_2 do
8:      R[j] ← A[q + j]
9:  L[n_1 + 1] ← ∞                ▷ A sentinel
10: R[n_2 + 1] ← ∞                ▷ Another sentinel
11: i ← 1
12: j ← 1
13: for k ← p to r do
14:     if L[i] ≤ R[j] then
15:         A[k] ← L[i]
16:         i ← i + 1
17:     else
18:         A[k] ← R[j]
19:         j ← j + 1
```

$n_1 + 1 = q - p + 2$.

$n_2 + 1 = r - q + 1$

← which two elements to compare.

$r - p + 2$.

← Put L[i] in A

← Put R[j] in A.

What is the running time of the above algorithm?

Upper bound:

$$c_1(n_1+1) + c_2(n_2+1) + c_3 \cdot (r-p+2)$$

$$= c_1(q-p+2) + c_2(r-q+1) + c_3(r-p+2)$$

$$\leq c_4 \cdot [q-p+2 + r-q+1 + r-p+2] \qquad (c_4 = \max(c_1, c_2, c_3))$$

$$= c_4[2 \cdot r - 2p + 5].$$

$$\leq O(r-p)$$

Lower bound

$$\Omega(r-p).$$

The run time $\Theta(r-p)$.

Time Complexity of Merge Sort

Let $T(n)$ denote the running time of the merge sort algorithm. Then $T(n)$ satisfies

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ D(n) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n) & \text{otherwise.} \end{cases}$$

divide: $\Theta(1)$. conquer: combine (Merge)

Time Complexity of Merge Sort

Let $T(n)$ denote the running time of the merge sort algorithm. Then $T(n)$ satisfies

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ D(n) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n) & \text{otherwise.} \end{cases}$$

Usually the order of growth of such an algorithm does not depend on

- the floor and ceiling functions,
- and the boundary conditions.

This leads to the simplified equation

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

Later we will see

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

Solve an Exact Recursive Equation

Suppose instead of asymptotic notations in the recursion, we have

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T\left(\frac{n}{2}\right) + cn & \text{otherwise.} \end{cases}$$

Moreover, assume that $n = 2^k$ for some integer $k \geq 0$.

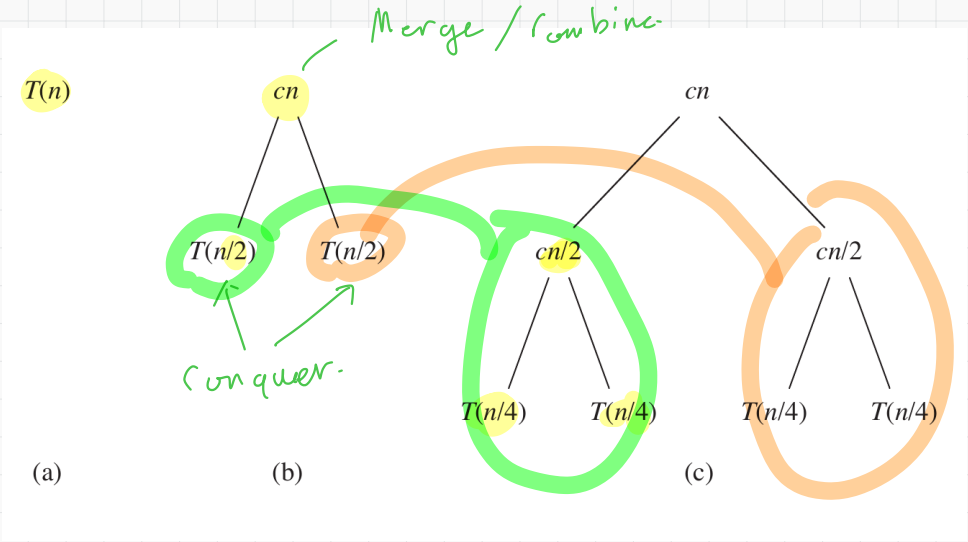
Solving this recursive equation, we get

$$T(n) = cn \log_2 n + cn = cn \lg n + cn.$$

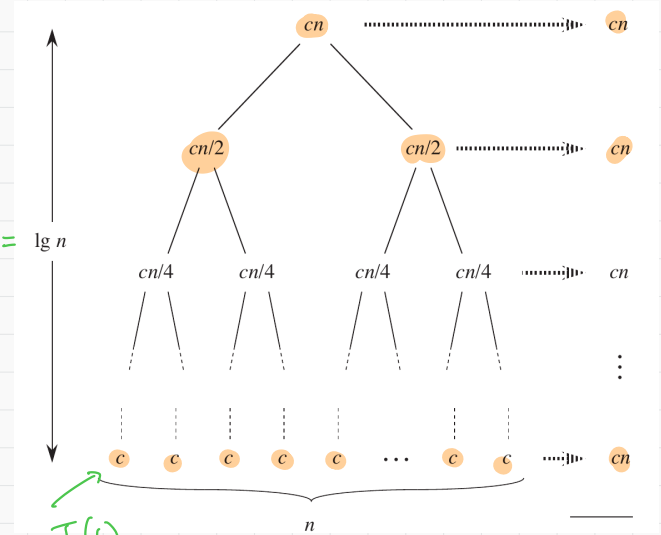
The argument uses a 🎄.

💣 This is not a real proof since the recursion is not exact.

Solve an Exact Recursive Equation by Tree



Solve an Exact Recursive Equation by Tree



$\log_2 n =$

$T(1)$

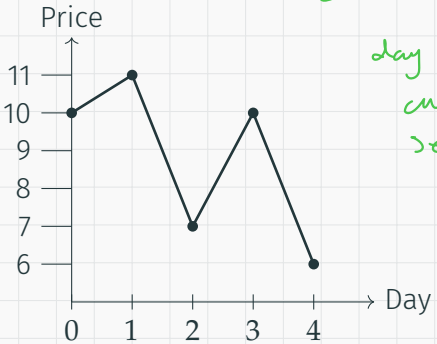
$T(n) = cn \cdot \log_2 n.$

ITA 4.1 The Maximum-Subarray Problem

📌 Example — Stock Price

The prices of a stock are given in the following plot.

🍰 If you buy once and sell once, when should you do so?



Common buy on day 4 and sell on day 1.

$S = \langle 10, 11, 7, 10, 6 \rangle$

Figure 4: What is the algorithm to maximize 💰!

$A = \langle 1, -4, 3, -4 \rangle$

The Maximum Subarray Problem

Let $S[0, \ldots, n]$ be the stock prices from day 0 to day n. Let

$$A[i] = S[i] - S[i-1], \text{ for } 1 \le i \le n.$$

Then if you buy on day $p-1$ and sell on day q, the profit is

$$S[q] - S[p-1] = \underline{S[q] - S[q-1] + S[q-1]} - S[q-2] + S[q-2] \cdots + S[p] - S[p-1].$$

$$= \underbrace{A[q] + A[q-1] \cdots + A[p]}$$

The sum of a subarray of A.

The Maximum Subarray Problem

Let $S[0, \ldots, n]$ be the stock prices from day 0 to day n. Let

$$A[i] = S[i] - S[i-1], \text{ for } 1 \leq i \leq n.$$

Then if you buy on day $p-1$ and sell on day q, the profit is

$$S[q] - S[p-1] = \sum_{i=p}^{q} A[i]$$

Given an array $A[1, \ldots, n]$, the maximum subarray problem aims to find indices p and q which maximizes

$$\sum_{i=p}^{q} A[i]$$

Divide of Conquer of Max-Subarray

L ← Left subarray of A

R ← Right subarray of A.

ml = the max subarray of L

mr = the max subarray of R.

m = max (ml, mr)

or

m = ml + mr

} Wrong

The Maximum Subarray Problem

Given an array $A[1, \ldots, n]$, the maximum subarray problem aims to find indices p and q which maximizes

$$\sum_{i=p}^{q} A[i]$$

```
1:  Max-Subarray-Brute-Force(A, n)
2:  max_profit ← −∞
3:  for p ← 1 to n do
4:      sum ← 0
5:      for q ← p to n do
6:          sum ← sum + A[q]
7:          if sum > max_profit then
8:              max_profit ← sum
9:              (buy, sell) ← (p − 1, q)
10: return max_profit
```

This runs in $O($_____$)$ time.

The Divide and Conquer Algorithm

🔪 **Divide**: Split the array into two halves.

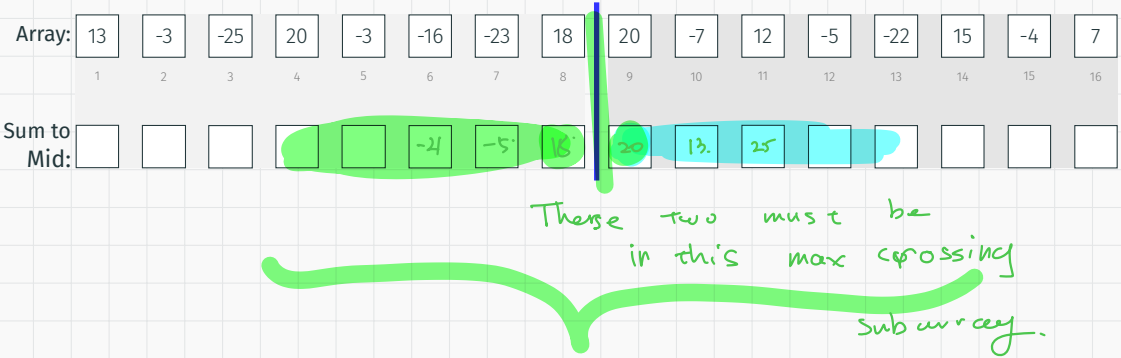
⚔️ **Conquer**: Recursively find the maximum subarray in each case:

- ⬅️ Maximum subarray entirely in the left half.
- ➡️ Maximum subarray entirely in the right half.
- ↔️ Maximum subarray crossing the midpoint.

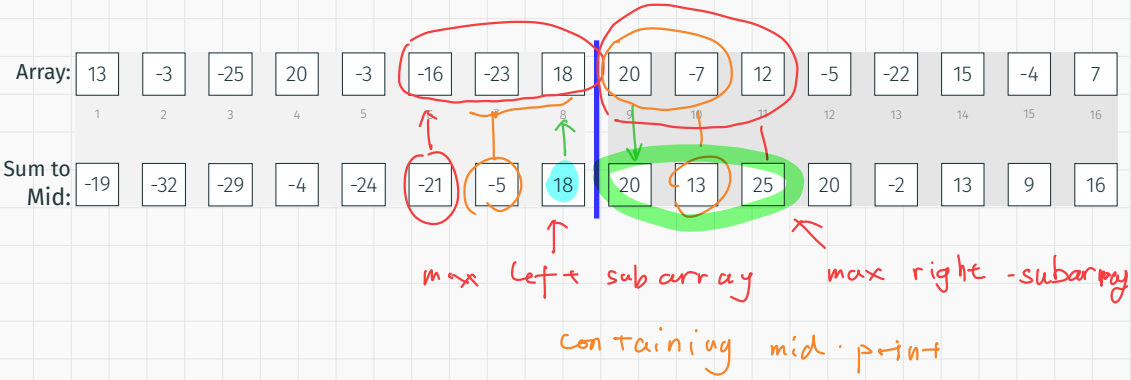
🥗 **Combine**: Pick the maximum subarray from each of the three possibilities.

🍰 What is the maximum crossing subarray in the following array?

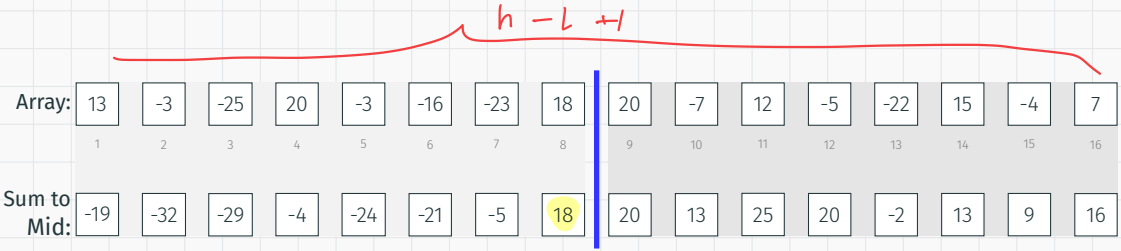
What is the maximum crossing subarray in the following array?



What is the maximum crossing subarray in the following array?



What is the maximum crossing subarray in the following array?



The maximum crossing subarray contains two parts:

- The maximum subarray entirely in the left half including the midpoint.
- The maximum subarray entirely in the right half including the midpoint.

Find Maximum Crossing Midpoint Subarray

```
1: Max-Crossing-Subarr(A, l, m, h)
2: left-sum ← −∞
3: sum ← 0
4: for i ← m downto l do
5:     sum ← sum + A[i]
6:     if sum > left-sum then
7:         left-sum ← sum
8:         max-left ← i
```

The array

low mid high

Find the max subarray containing mid point.

Find Maximum Crossing Midpoint Subarray

```
1:  Max-Crossing-Subarr(A, l, m, h)
2:  left-sum ← −∞
3:  sum ← 0
4:  for i ← m downto l do
5:      sum ← sum + A[i]
6:      if sum > left-sum then
7:          left-sum ← sum
8:          max-left ← i
9:  right-sum ← −∞
10: sum ← 0
11: for j ← m + 1 to h do
12:     sum ← sum + A[j]
13:     if sum > right-sum then
14:         right-sum ← sum
15:         max-right ← j
16: return
    (max-left, max-right, left-sum +
    right-sum)
```

Find max right sub array containing mid point.

What is the time complexity of this procedure? $\Theta(h - l)$

Find Maximum Subarray Pseudocode

```
1:  Max-Subarr(A, l, h)
2:  if h == l then
3:      return (l, h, A[l])
4:  m ← ⌊(l+h)/2⌋
5:  (ll, lh, ls) ← Max-Subarr(A, l, m)
6:  (rl, rh, rs) ← Max-Subarr(A, m + 1, h)
7:  (cl, ch, cs) ← Max-Crossing-Subarr(A, l, m, h)
8:  if ls ≥ rs and ls ≥ cs then
9:      return (ll, lh, ls)
10: else if rs ≥ ls and rs ≥ cs then
11:     return (rl, rh, rs)
12: else
13:     return (cl, ch, cs)
```

divide: $O(1)$

conquer:

$\rightarrow \Theta(h - l) = \Theta(n)$ if A has length n.

Combine: $O(1)$.

Time Complexity of Maximum Subarray

Let $T(n)$ denote the running time of the previous algorithm. Then $T(n)$ satisfies

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ D(n) + 2T\left(\frac{n}{2}\right) + \Theta(n) + C(n) & \text{otherwise.} \end{cases}$$

divide: $O(1)$ conquer: combine: $O(1)$

Time Complexity of Maximum Subarray

Let $T(n)$ denote the running time of the previous algorithm. Then $T(n)$ satisfies

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ D(n) + 2T\left(\frac{n}{2}\right) + \Theta(n) + C(n) & \text{otherwise.} \end{cases}$$

This simplifies to

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

As the recursion is the same as merge sort, the solution is also

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

What is the *sum* of the maximum subarray that crosses the midpoint in the following array?

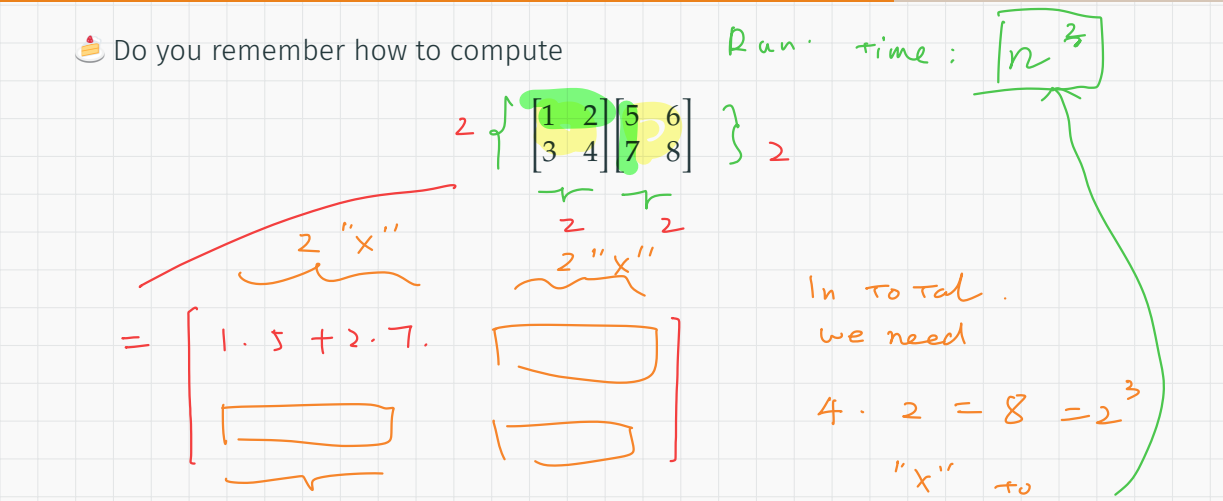
$$A = \langle 0, 3, 2, 2, -5, 1, -2, 2, 3, -1, -5, -2, 9, -4, -2, -2 \rangle.$$

Answer: 9.

ITA 4.2 STRASSEN'S ALGORITHM FOR MATRIX MULTIPLICATION

Review: Matrix Multiplication

Do you remember how to compute



Square Matrix Multiplication by Brute Force

Let A and B be two $n \times n$ matrices.

Let a_{ij} and b_{ij} be the entries of A and B respectively.

Let $C = A \cdot B$ and c_{ij} be the entries of C.

Then

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

This is called the row-column rule.

Square Matrix Multiplication by Brute Force

Let A and B be two $n \times n$ matrices.

Let a_{ij} and b_{ij} be the entries of A and B respectively.

Let $C = A \cdot B$ and c_{ij} be the entries of C.

Then

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

This is called the row-column rule.

1: **Square-Matrix-Multiply**(A, B)
2: $n \leftarrow A$.rows
3: let C be a new $n \times n$ matrix
4: **for** $i = 1$ to n **do**
5: **for** $j = 1$ to n **do**
6: $c_{ij} \leftarrow 0$
7: **for** $k = 1$ to n **do**
8: $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$
9: **return** C

🍰 What is the time complexity of this algorithm?

Matrix Partitioning

Given $n \times n$ matrices A, B, and $C = AB$, we can partition them

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

n/2 × n/2.

where the sub-matrices are of size $n/2 \times n/2$.

Matrix Partitioning

Given $n \times n$ matrices A, B, and $C = AB$, we can partition them

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

where the sub-matrices are of size $n/2 \times n/2$.

For instance, consider

$$A = \begin{pmatrix} 2 & 3 & 0 & 2 \\ 2 & 3 & 0 & 0 \\ 2 & 1 & 2 & 2 \\ 2 & 2 & 3 & 0 \end{pmatrix}, B = \begin{pmatrix} 3 & 3 & 3 & 2 \\ 1 & 0 & 1 & 3 \\ 3 & 1 & 1 & 1 \\ 3 & 3 & 0 & 0 \end{pmatrix}, \quad C = \begin{pmatrix} 15 & 12 & 9 & 13 \\ 9 & 6 & 9 & 13 \\ 19 & 14 & 9 & 9 \\ 17 & 9 & 11 & 13 \end{pmatrix}$$

Can you identify what A_{11}, ..., C_{22} are?

Then, $C = A \cdot B$ can be written as:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

This leads to four corresponding equations.

$$\begin{cases} C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \\ C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \\ C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \\ C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \end{cases}$$

we still need $\frac{n^3}{4} \times 4 = n^3$ multiplication

$\left(\frac{n}{2}\right)^3$ "x" $\left(\frac{n}{2}\right)^3$ "x" $\frac{n^3}{4}$ "x"

Recursive Square Matrix Multiplication Algorithm

Let us abbreviate Square-Matrix-Multiply-Recursive as SMMR.

SMMR(A, B)
$n \leftarrow A$.rows
$C \leftarrow$ new $n \times n$ matrix
if $n = 1$ then
    $C_{11} \leftarrow a_{11} \cdot b_{11}$
else
    Partition A, B, and C as in equations (4.9)
    $C_{11} \leftarrow \text{SMMR}(A_{11}, B_{11}) + \text{SMMR}(A_{12}, B_{21})$
    $C_{12} \leftarrow \text{SMMR}(A_{11}, B_{12}) + \text{SMMR}(A_{12}, B_{22})$
    $C_{21} \leftarrow \text{SMMR}(A_{21}, B_{11}) + \text{SMMR}(A_{22}, B_{21})$
    $C_{22} \leftarrow \text{SMMR}(A_{21}, B_{12}) + \text{SMMR}(A_{22}, B_{22})$
return C

Time Complexity of Recursive Square Matrix Multiplication

Let $T(n)$ denote the running time of SMMR. Then $T(n)$ satisfies

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8T\left(\frac{n}{2}\right) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

Thus,

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2).$$

In the next lecture, we'll see that this implies $T(n) = \Theta(n^{\lg 8}) = \Theta(n^3)$.

😩 Nothing has changed!

Strassen's Algorithm

Strassen's method improves SMMR by computing

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

which reduces the number of matrix-multiplications to 7.

The resulting product matrix $C = A \cdot B$ is then computed as: You can merge the given expressions into a single equation as follows:

$$C = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}.$$

Thus, the running time satisfies:

Num of "x" needed.

Combine.

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2).$$

Conquer

Num of "x" needed for $\left(\frac{n}{2} \times \frac{n}{2}\right)$ times $\left(\frac{n}{2} \times \frac{n}{2}\right)$ matrices

In the next lecture, we'll see that this implies

$$T(n) = \Theta(n^{\lg 7}) \approx \Theta(n^{2.81}) = o(n^3)$$

$\log_2 7$

which is much better than $\Theta(n^3)$!

AlphaGo

- Treated the complex game of Go as a **search problem**.
- Beat the best human players of Go in 2017.

Figure 5: Demis Hassabis (1976-), co-founder of Google DeepMind.

🤪 One Logic, Two Breakthroughs

🏆 AlphaGo

- Treated the complex game of Go as a **search problem**.
- Beat the best human players of Go in 2017.

🔢 AlphaTensor

- Treats **matrix multiplication** as a game.
- Found algorithms faster than human limits to speed up all AI.

Figure 5: Demis Hassabis (1976-), co-founder of Google DeepMind.

🤪 One Logic, Two Breakthroughs

🏆 AlphaGo

- Treated the complex game of Go as a **search problem**.
- Beat the best human players of Go in 2017.

🔢 AlphaTensor

- Treats **matrix multiplication** as a game.
- Found algorithms faster than human limits to speed up all AI.

🧬 AlphaFold

- Treats **protein folding** as a search for the best shape.
- **Result:** 2024 Nobel Prize in Chemistry.

Figure 5: Demis Hassabis (1976-), co-founder of Google DeepMind.

What are your main takeaways today? Any questions?