CSE100: Design and Analysis of Algorithms
Lecture 08 – Selection and Median (wrap up), Maximum Subarray & Matrix Multiplication

Feb 10th 2022

More Recursion, Beyond the Master Theorem, More divide and conquer, Strassen’s algorithm

The Plan

1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
   a) The outline of the algorithm.
   b) How to pick the pivot.
4. Return of the Substitution Method.

A good enough pivot (review)

- We split the input not quite in half:
  - 3n/10 < len(L) < 7n/10
  - 3n/10 < len(R) < 7n/10
- If we could do that (let's say, in time O(n)), the **Master Theorem** would say:
  - $T(n) \leq T\left(\frac{7n}{10}\right) + O(n)$
  - So a = 1, b = 10/7, d = 1
  - $T(n) \leq O\left(n^d\right) = O(n)$

STILL GOOD!

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

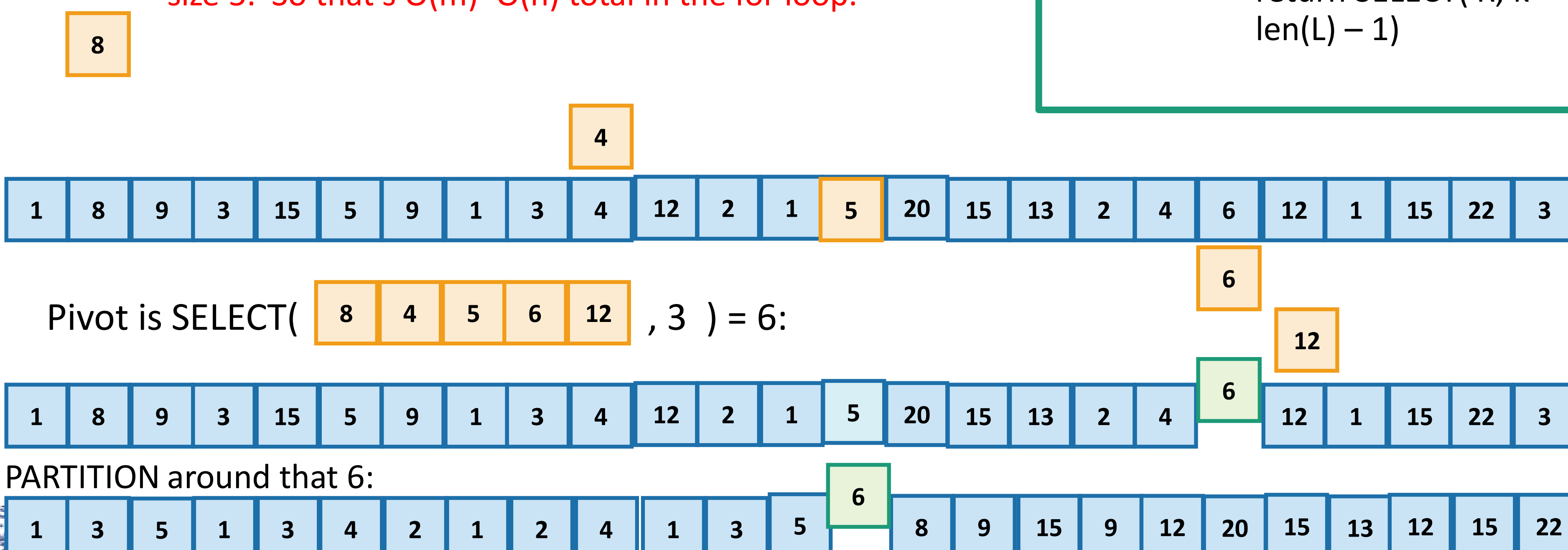
$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

How to pick the pivot (review)

- CHOOSEPIVOT(A):
  - Split A into m = $\left\lceil \frac{n}{5} \right\rceil$ groups, of size <=5 each.
  - **For** i=1, .., m:
    - Find the median within the i^{th} group, call it p_i
  - p = SELECT([$p_1, p_2, p_3, \ldots, p_m$] , m/2)
  - **return** p

This takes time O(1), for each group, since each group has size 5. So that's O(m)=O(n) total in the for loop.

- SELECT(A, p=k):
  - **If** len(A) <= 50:
    - A = MergeSort(A)
    - Return A[k]
  - p = CHOOSEPIVOT(A)
  - L, A[p], R = PARTITION(A,p)
  - **If** len(L) = k - 1:
    - **Return** A[p]
  - **Else If** len(L) < k - 1:
    - **Return** SELECT(L, k)
  - **Else if** len(L) > k - 1:
    - return SELECT(R, k – len(L) – 1)



This part is L

This part is R: it's almost the same size as L.

The whole algorithm (review)

- SELECT(A, p=k):
  - **If** len(A) <= 50:
    - A = MergeSort(A)
    - Return A[k]
  - p = CHOOSEPIVOT(A)
  - L, A[p], R = PARTITION(A,p)
  - **If** len(L) = k - 1:
    - **Return** A[p]
  - **Else If** len(L) > k - 1:
    - **Return** SELECT(L, k)
  - **Else if** len(L) < k - 1:
    - return SELECT(R, k – len(L) – 1)

Note: We use recursion in two ways! Both in SELECT itself, and in CHOOSEPIVOT.

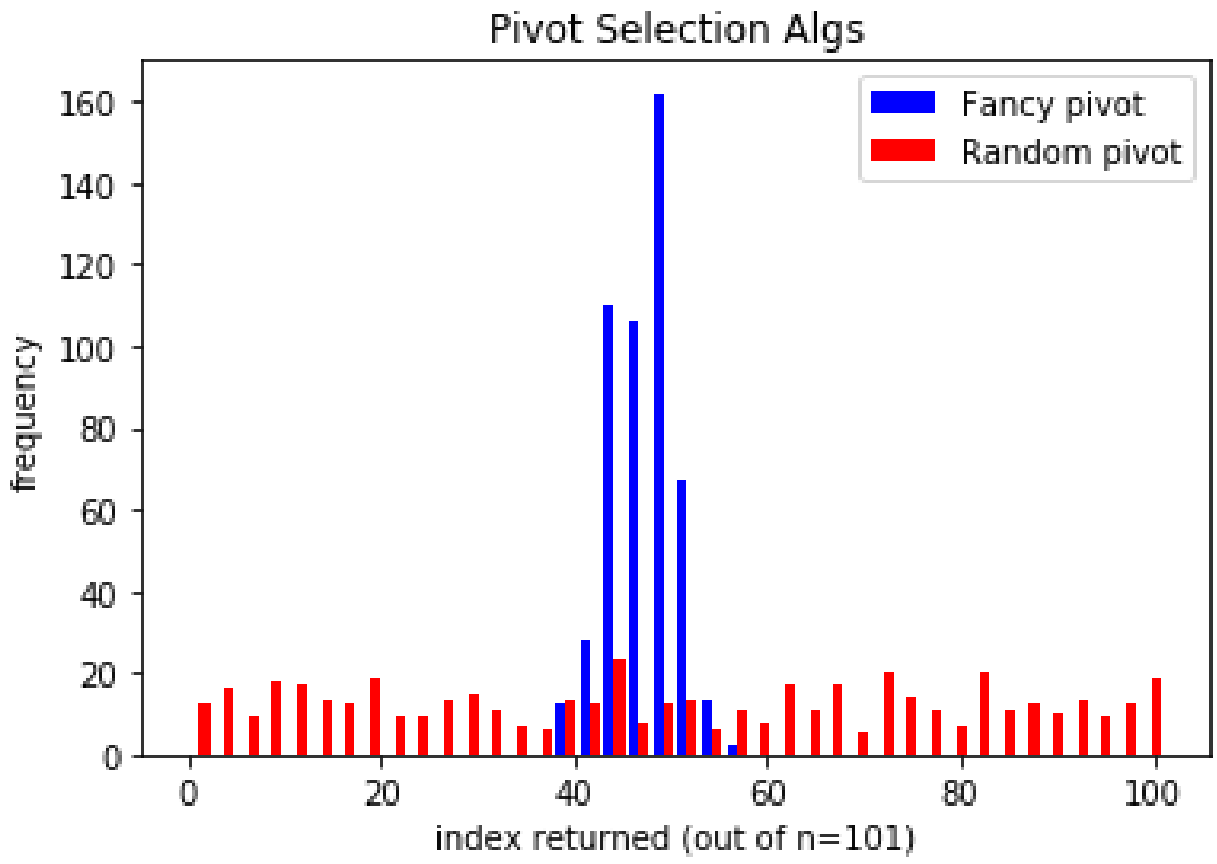
- PARTITION(A, p):
  - L = new array
  - R = new array
  - **For** i=1,…,n:
    - **If** i==p, continue
    - **Else If** A[i] <= A[p]:
      - L.append(A[i])
    - **Else if** A[i] > A[p]:
      - R.append(A[i])
  - **Return** L, A[p], R

- Does it work?
- Yes, our proof before worked for any pivoting strategy.

- CHOOSEPIVOT(A):
  - Split A into m = $\left\lceil \frac{n}{5} \right\rceil$ groups, of size <=5 each.
  - **For** i=1, .., m:
    - Find the median within the i^{th} group, call it p_i
  - p = SELECT([$p_1, p_2, p_3, \ldots, p_m$] , m/2)
  - **return** p

CLAIM: this works
divides the array *approximately* in half

- Empirically:



CLAIM: this works
divides the array *approximately* in half

- Formally, we will prove (later):

Lemma: If we choose the pivots like this, then

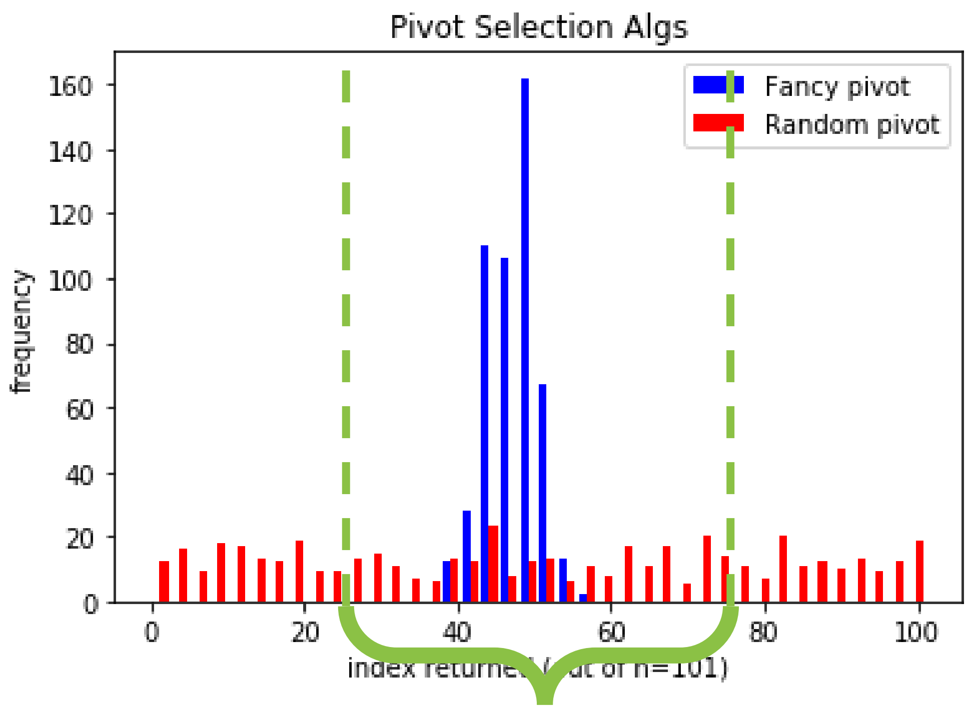
$$|L| \leq \frac{7n}{10} + 5$$

and

$$|R| \leq \frac{7n}{10} + 5$$

Sanity Check

$|L| \leq \frac{7n}{10} + 5$ and $|R| \leq \frac{7n}{10} + 5$



That's this window

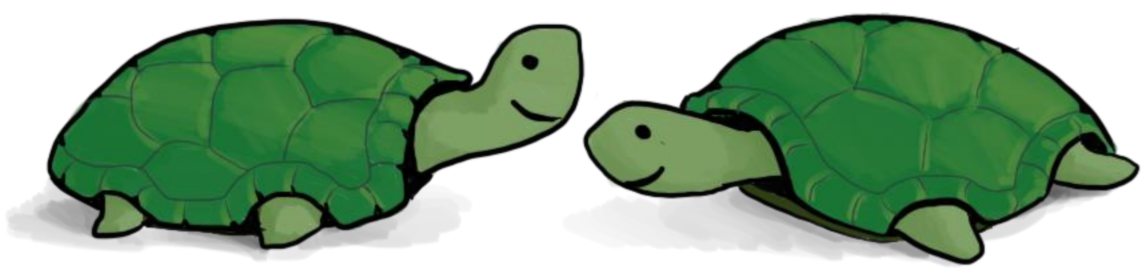
Actually in practice (on randomly chosen arrays) it looks **even better**!

But this is a worst-case bound.

How about the running time?

- Suppose the Lemma is true. (It is).
  - $|L| \leq \frac{7n}{10} + 5$ and $|R| \leq \frac{7n}{10} + 5$
- Recurrence relation:

$$T(n) \leq ?$$



Pseudocode

- CHOOSEPIVOT(A) returns some pivot for us.
  - How?? We'll see later…
- PARTITION(A, p) splits up A into L, A[p], R.

- SELECT(A,k):
  - **If** len(A) <= 50:
    - A = MergeSort(A)
    - **Return** A[k-1]
  - p = CHOOSEPIVOT(A)
  - L, pivotVal, R = PARTITION(A,p)
  - **if** len(L) == k-1:
    - return pivotVal
  - **Else if** len(L) > k-1:
    - return SELECT(L, k)
  - **Else if** len(L) < k-1:
    - return SELECT(R, k – len(L) – 1)

Base Case: If the len(A) = O(1), then any sorting algorithm runs in time O(1).

Case 1: We got lucky and found exactly the k'th smallest value!

Case 2: The k'th smallest value is in the first part of the list

Case 3: The k'th smallest value is in the second part of the list

How about the running time?

- Suppose the Lemma is true. (It is).
  - $|L| \leq \frac{7n}{10} + 5$ and $|R| \leq \frac{7n}{10} + 5$

- Recurrence relation:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

The call to CHOOSEPIVOT makes one further recursive call to SELECT on an array of size n/5.

Outside of CHOOSEPIVOT, there's at most one recursive call to SELECT on array of size 7n/10 + 5. We're going to drop the "+5" for convenience, but see CLRS for a more careful treatment which includes it.

The Plan

1. The Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
   a) The outline of the algorithm.
   b) How to pick the pivot.
4. Return of the Substitution Method.

This sounds like a job for…

The Substitution Method!

Step 1: generate a guess
Step 2: try to prove that your guess is correct
Step 3: profit

$$T(n) \le T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

Base case: $T(n) = 1$ when $1 \le n \le 50$

That's convenient! We did similar examples at the beginning of lecture!

Our goal: $T(n) = O(n)$

Technically we will prove for
$T(n) \le T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n,$
not when the last term has a big-Oh…

Plucky the Pedantic Penguin

The Substitution Method

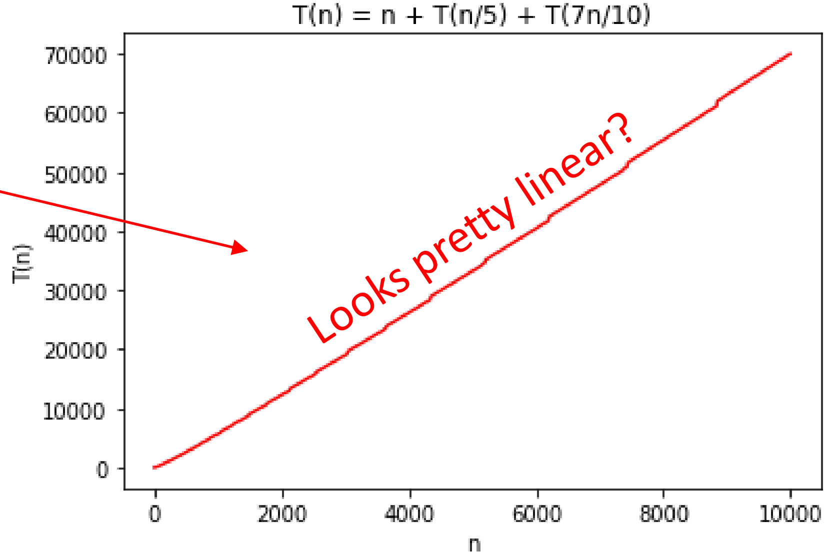
- Step 1: Guess what the answer is.
- Step 2: Prove by induction that your guess is correct.
- Step 3: Profit.

Step 1: guess the answer

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n \text{ for } n > 50.$$

Base case: $T(n) = 1$ when $1 \leq n \leq 50$

- Trying to work backwards gets gross fast…
- We can also just try it out.
- Let's guess O(n) and try to prove it.



Step 2: prove our guess is right

$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n$ for $n > 50$.

Base case: $T(n) = 1$ when $1 \leq n \leq 50$

C is some constant we'll have to fill in later!

- Inductive Hypothesis: $T(j) \leq \boldsymbol{C}j$ for all $1 \leq j \leq n$.
- Base case: $1 = T(j) \leq \boldsymbol{C}j$ for all $1 \leq$ j ≤ 50
- Inductive step:
  - Assume that the IH holds for $n = k - 1$.
  - $T(k) \leq k + T\left(\frac{k}{5}\right) + T\left(\frac{7k}{10}\right)$

$$\leq k + \boldsymbol{C} \cdot \left(\frac{k}{5}\right) + \boldsymbol{C} \cdot \left(\frac{7k}{10}\right)$$

$$= k + \frac{\boldsymbol{C}}{5}k + \frac{7\boldsymbol{C}}{10}k$$

$$\leq \boldsymbol{C}k \;??$$

  - (aka, want to show that IH holds for $k = n$).

Whatever we choose C to be, it should have C≥1

Let's solve for C and make this true! C = 10 works. (whiteboard)

- Conclusion:
  - There is some $\boldsymbol{C}$ so that for all $n \geq 1$, $T(n) \leq \boldsymbol{C}n$
  - Aka, $\mathrm{T}(n) = O(n)$. (Technically we also need $0 \leq T(n)$ here…)

Step 3: Profit

(Aka, pretend we knew this all along).

$$T(n) \leq n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) \text{ for } n > 50.$$

Base case: $T(n) = 1$ when $1 \leq n \leq 50$

(Assume that $T(n) \geq 0$ for all n. Then,)

Theorem: $T(n) = O(n)$

Proof:

- Inductive Hypothesis: $T(j) \leq \mathbf{10}j$ for all $1 \leq j \leq n$.
- Base case: $1 = T(j) \leq \mathbf{10}j$ for all $1 \leq$ j ≤ 50
- Inductive step:
  - Assume the IH holds for $n = k - 1$.
  - $T(k) \leq k + T\left(\frac{k}{5}\right) + T\left(\frac{7k}{10}\right)$
    $\leq k + \mathbf{10} \cdot \left(\frac{k}{5}\right) + \mathbf{10} \cdot \left(\frac{7k}{10}\right)$
    $= k + 2k + 7k = \mathbf{10}k$
  - Thus, IH holds for $n = k$.
- Conclusion:
  - For all $n \geq 1, T(n) \leq \mathbf{10}n$
  - (Also $0 \leq T(n)$ for all $n \geq 1$ since we assumed so.)
  - Aka, $T(n) = O(n)$, using the definition with $n_0 = 1, c = 10$.

Plucky added the stuff about $T(n) \geq 0$ because this is part of the definition of O() and we were ignoring it…

Step 3: Profit

(Aka, pretend we knew this all along).

$T(n) \leq n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)$ for $n > 50$.

Base case: $T(n) = 1$ when $1 \leq n \leq 50$

(Assume that $T(n) \geq 0$ for all n. Then,)

Theorem: $T(n) = O(n)$

Proof:

- Inductive Hypothesis: $T(n) \leq \mathbf{10}n$.
- Base case: $1 = T(n) \leq \mathbf{10}n$ for all $1 \leq \mathrm{n} \leq 50$
- Inductive step:
  - Assume the IH holds for all $1 \leq n \leq k - 1$.
  - $T(k) \leq k + T\left(\frac{k}{5}\right) + T\left(\frac{7k}{10}\right)$
    $\leq k + \mathbf{10} \cdot \left(\frac{k}{5}\right) + \mathbf{10} \cdot \left(\frac{7k}{10}\right)$
    $= k + 2k + 7k = \mathbf{10}k$
  - Thus, IH holds for $n = k$ too.
- Conclusion:
  - For all $n \geq 1, T(n) \leq \mathbf{10}n$
  - (Also $0 \leq T(n)$ for all $n \geq 1$ since we assumed so.)
  - Aka, $T(n) = O(n)$, using the definition with $n_0 = 1, c = 10$.

Plucky added the stuff about $T(n) \geq 0$ because this is part of the definition of O()…

That was pretty pedantic

- Can't we just use the nice O() notation?
- $T(n) \leq c \cdot n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10} + 5\right)$
- Inductive hypothesis: $T(n) = O(n)$.

vs.

$$T(n) \leq \begin{cases} d \cdot 50 & if \;\; n \leq 50 \\ d \cdot n & if \;\; n > 50 \end{cases}$$

- Then the inductive step is just

$$\begin{aligned} T(n) &\leq c \cdot n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10} + 5\right) \\ &\leq c \cdot n + O\left(\frac{n}{5}\right) + O\left(\frac{7n}{10} + 5\right) = O(n) \end{aligned}$$

Actually, that doesn't work

- Consider this "proof" that MERGESORT runs in time O(n). (It doesn't).
- $T(n) \leq 2\,T\left(\frac{n}{2}\right) + c \cdot n$
- Inductive hypothesis: $T(n) = O(n)$.

- Then the inductive step is just

$$T(n) \leq 2\,T\left(\frac{n}{2}\right) + c \cdot n \leq 2 \cdot O\left(\frac{n}{2}\right) + c \cdot n = O(n).$$

- What's wrong???
  - (It turns out the base case is fine).

The problem is being sloppy with O()

- When we use O() in the inductive hypothesis, it might have a different constant "c" in the definition of O() each time the hypothesis is called. As a result, this "constant" might depend on n, which is not allowed in the definition of O().
- Try rigorously:
  - Suppose that $T(n) \leq 2\,T\left(\frac{n}{2}\right) + c \cdot n$
  - Let's guess the solution is $T(n) \leq \begin{cases} d \cdot n_0 & if \quad n \leq n_0 \\ d \cdot n & if \quad n > n_0 \end{cases}$
  - Then the inductive argument would go....
  - $T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n \leq 2 \cdot \frac{dn}{2} + c \cdot n = (d + c)n$
  - We need that to be smaller than dn for the induction to work.
  - No way that's going to happen, since $c > 0$.
  - So, this argument won't work. (Which is good, since the statement is false).

Told you so.

Recap of approach

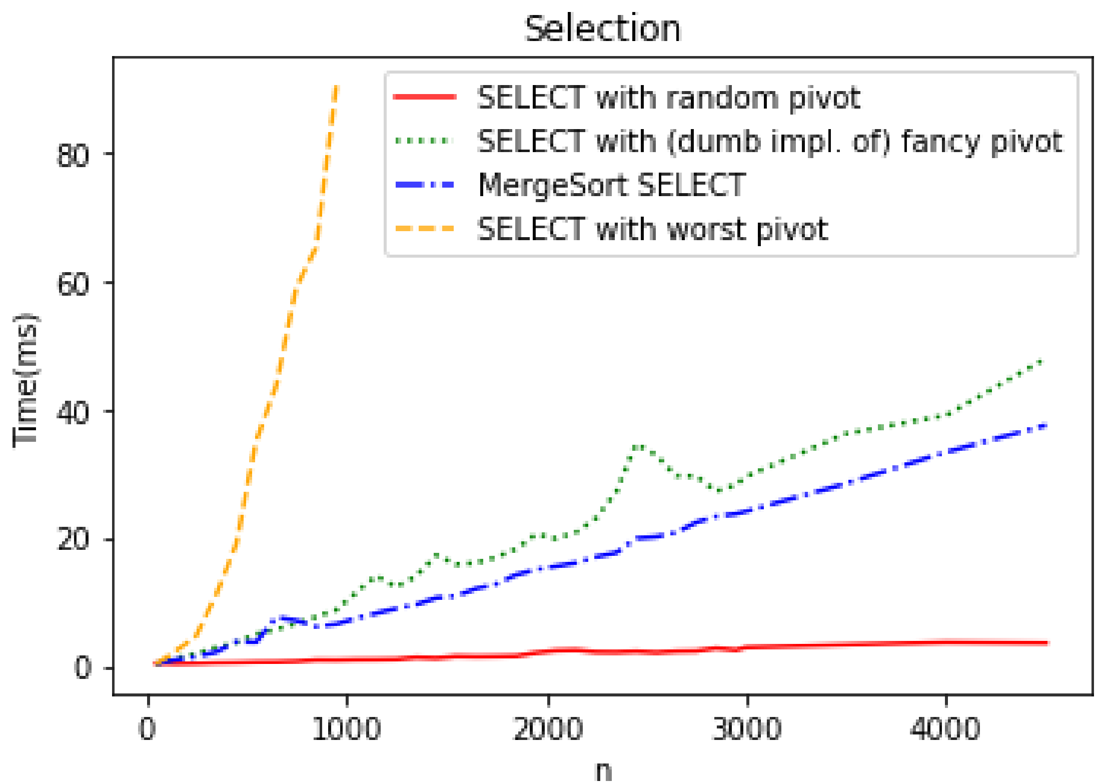
- First, we figured out what the ideal pivot would be.
  - Find the median
- Then, we figured out what a **pretty good** pivot would be.
  - An approximate median
- Finally, we saw how to get our pretty good pivot!
  - Median of medians and divide and conquer!
  - Hooray!

In practice?

- With a simple implementation, our fancy version of SELECT is worse than the MergeSort-based SELECT ☹
  - But O(n) is better than O(nlog(n))! How can that be?
  - *What's the constant in front of the n in our proof? 20? 30?*

- On **non-adversarial** inputs, random pivot choice is much better.

Moral:
Just pick a random pivot if you don't expect nefarious arrays.



Optimize the implementation of SELECT (with the fancy pivot). Can you beat MergeSort?

Siggi the Studious Stork

What have we learned?

Pending the Lemma

- It is possible to solve SELECT in time O(n).
  - Divide and conquer!

- If you want a deterministic algorithm and expect that a bad guy will be picking the list, **choose a pivot cleverly.**
  - More divide and conquer!

- If you don't expect that a bad guy will be picking the list, in practice it's better just to **pick a random pivot.**

The Plan

1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
   a) The outline of the algorithm.
   b) How to pick the pivot.
4. Return of the Substitution Method.
5. (If time) Proof of that Lemma.

Lemma: that median-of-medians thing is a good idea.

- **Lemma:** If L and R are as in the algorithm SELECT given above, then

$$|L| \leq \frac{7n}{10} + 5$$

and

$$|R| \leq \frac{7n}{10} + 5$$

Arbitrary numbers!
I thought the whole point of O() was that we'd never have to do that again.

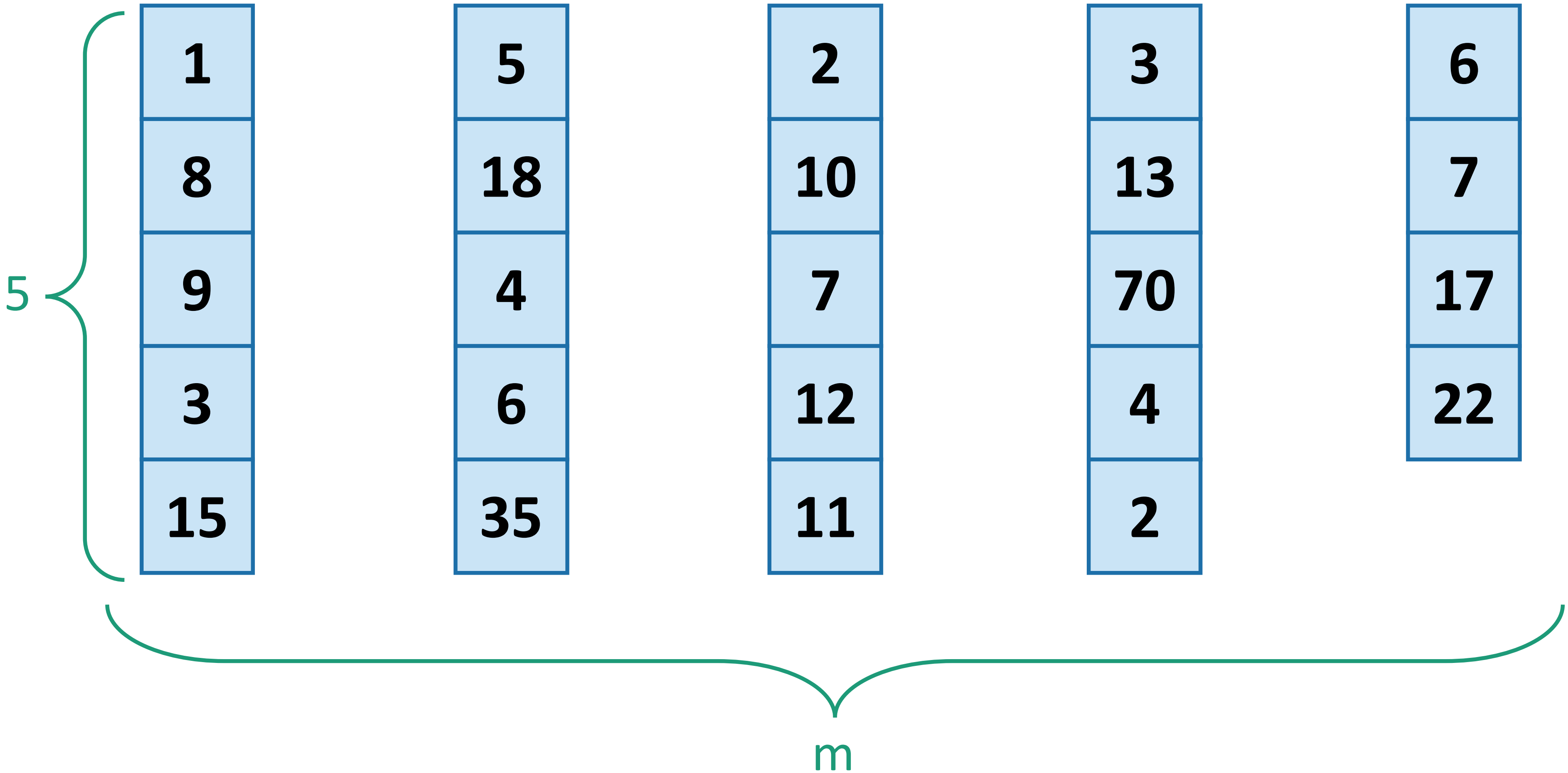
We're only doing it so we can get a (correct) O() bound later…so our choice of numbers **will** be pretty arbitrary.

- Why is this good?
  - It means that things are pretty balanced.
- We will see a proof by picture.
- See CLRS or the Lecture Notes for proof by proof.

Proof by picture generally <u>not</u> okay on exams – need at least a few words. (But pictures are encouraged if it makes things clearer!)

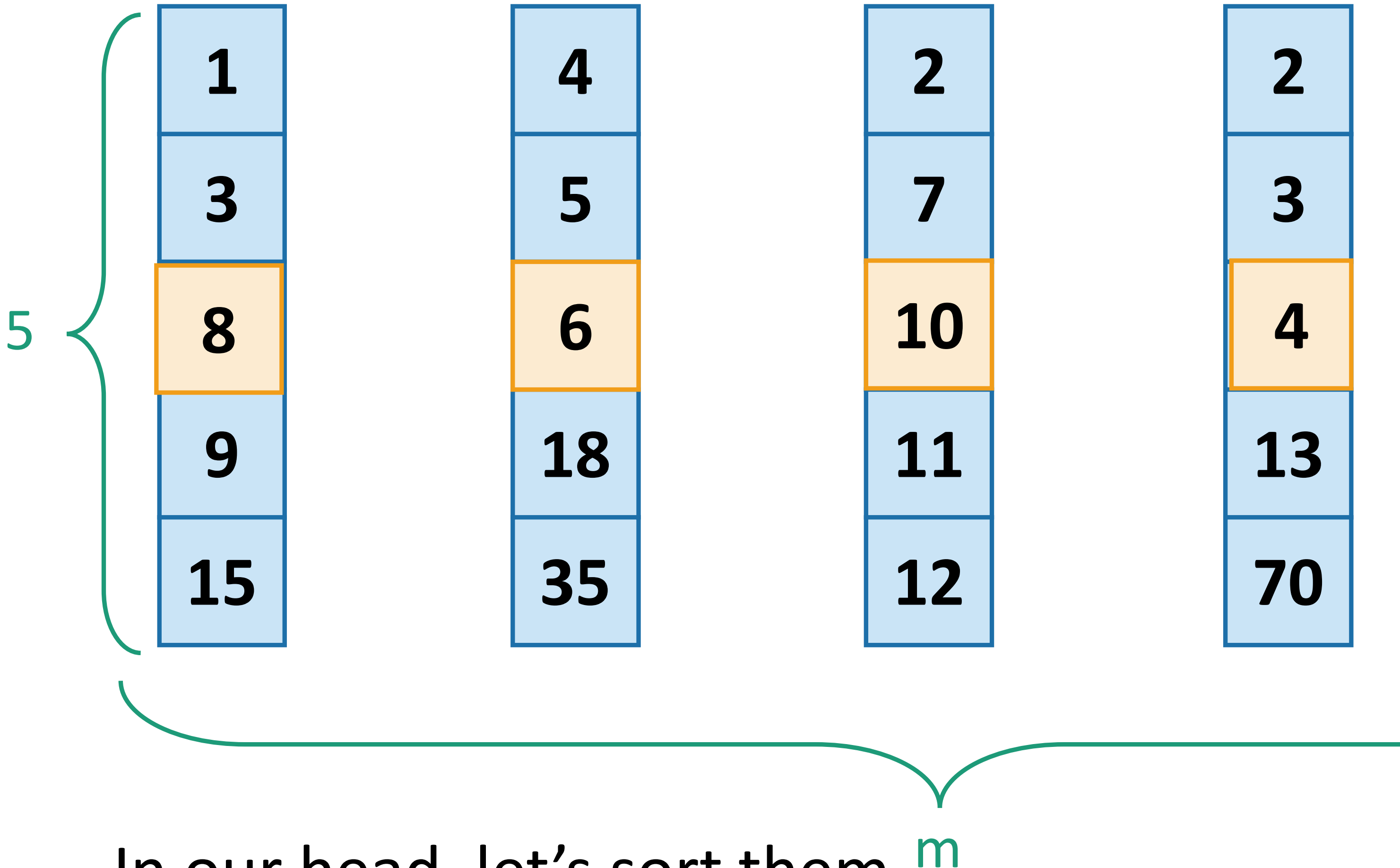
grumpy cat

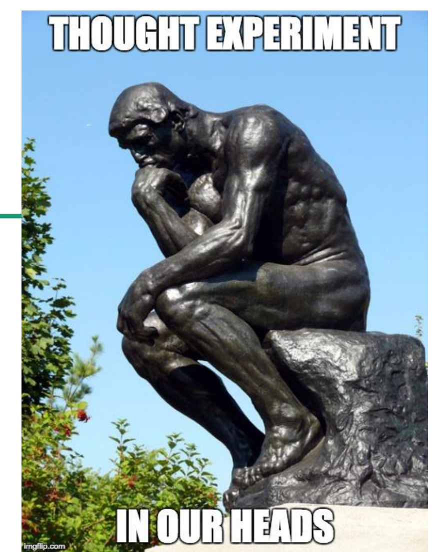
Proof by picture



Say these are our m = [n/5] sub-arrays of size at most 5.

Proof by picture

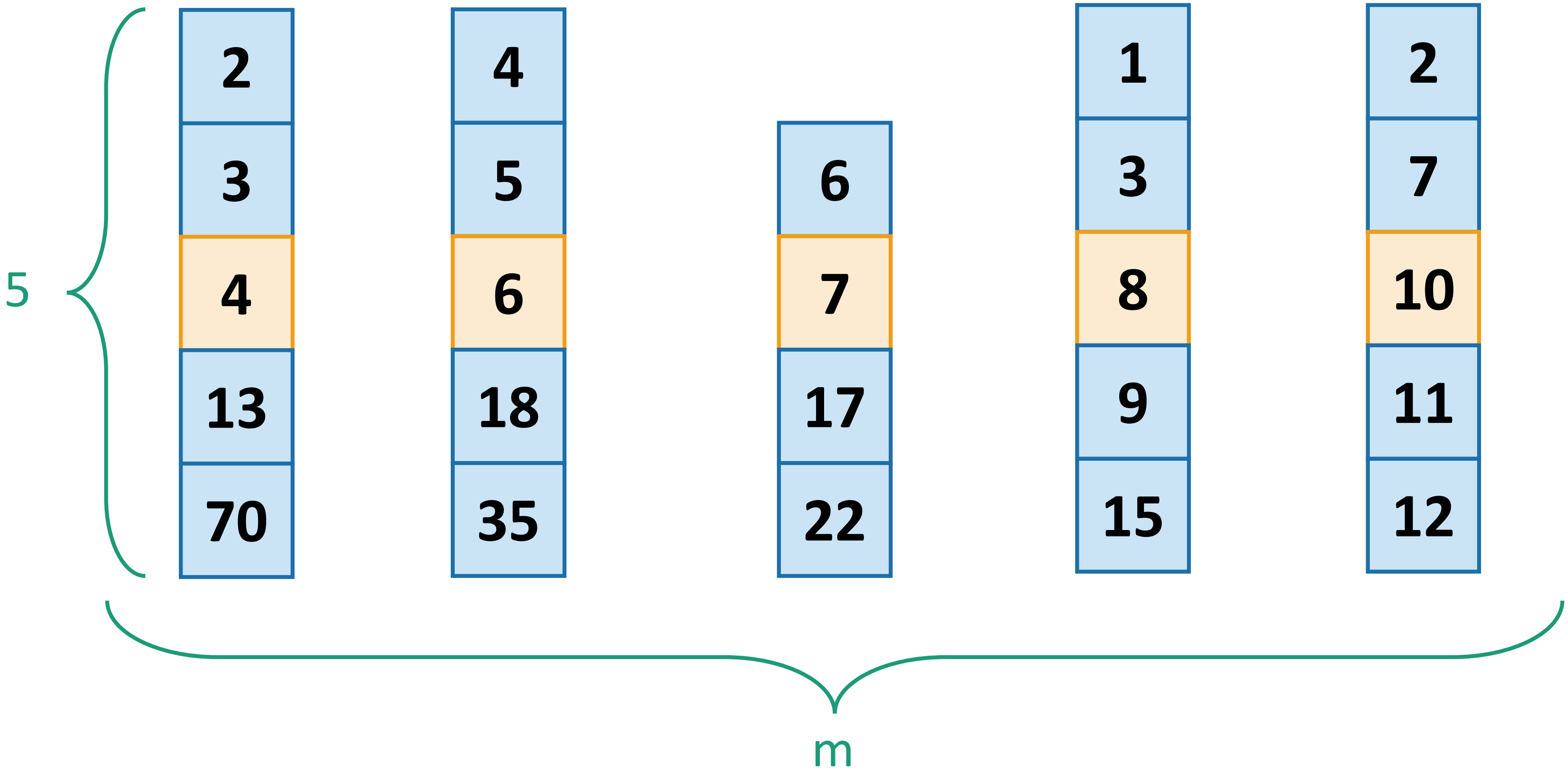




In our head, let's sort them.

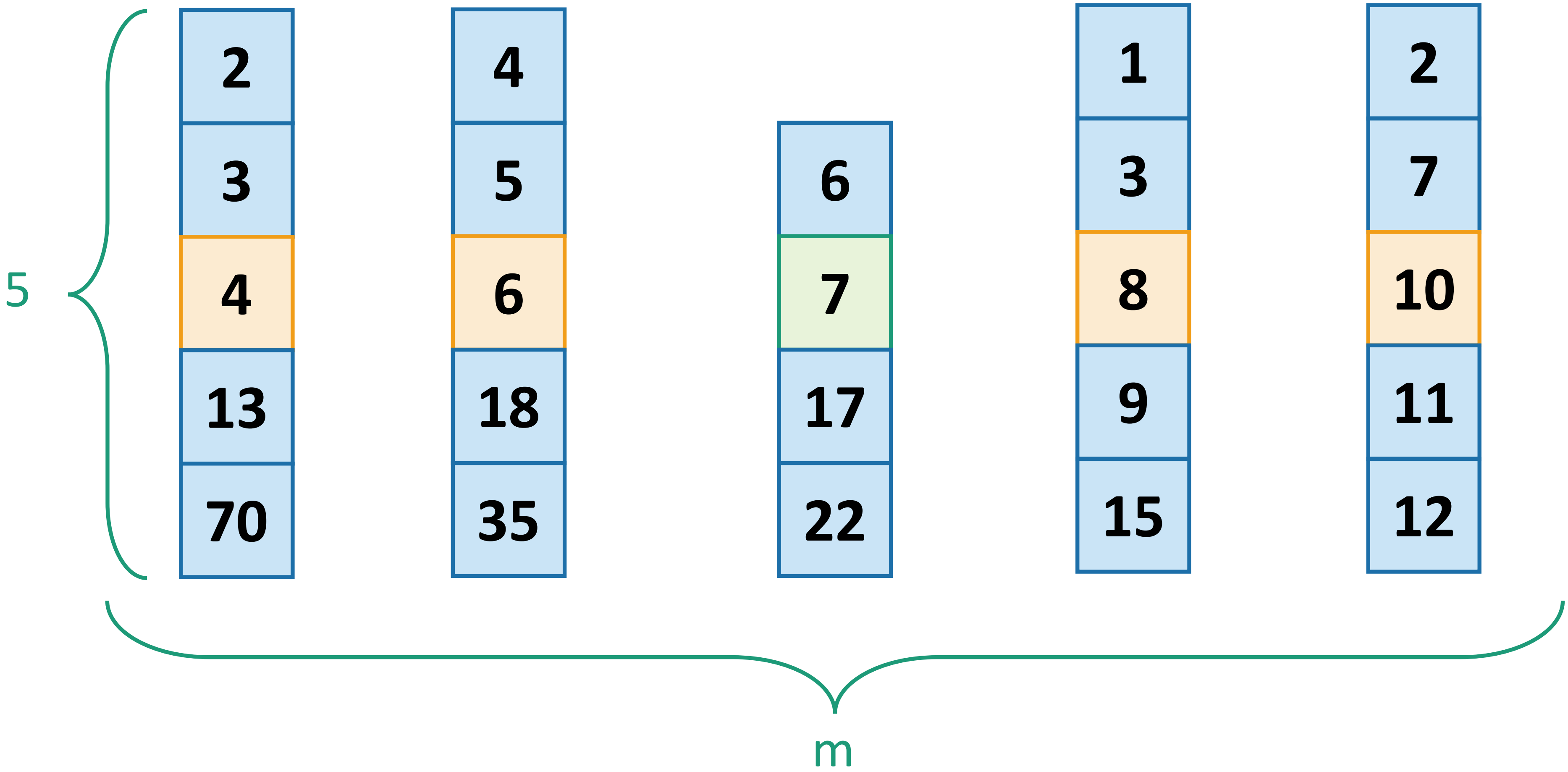
Then find medians.

Proof by picture



Then let's sort them by the median

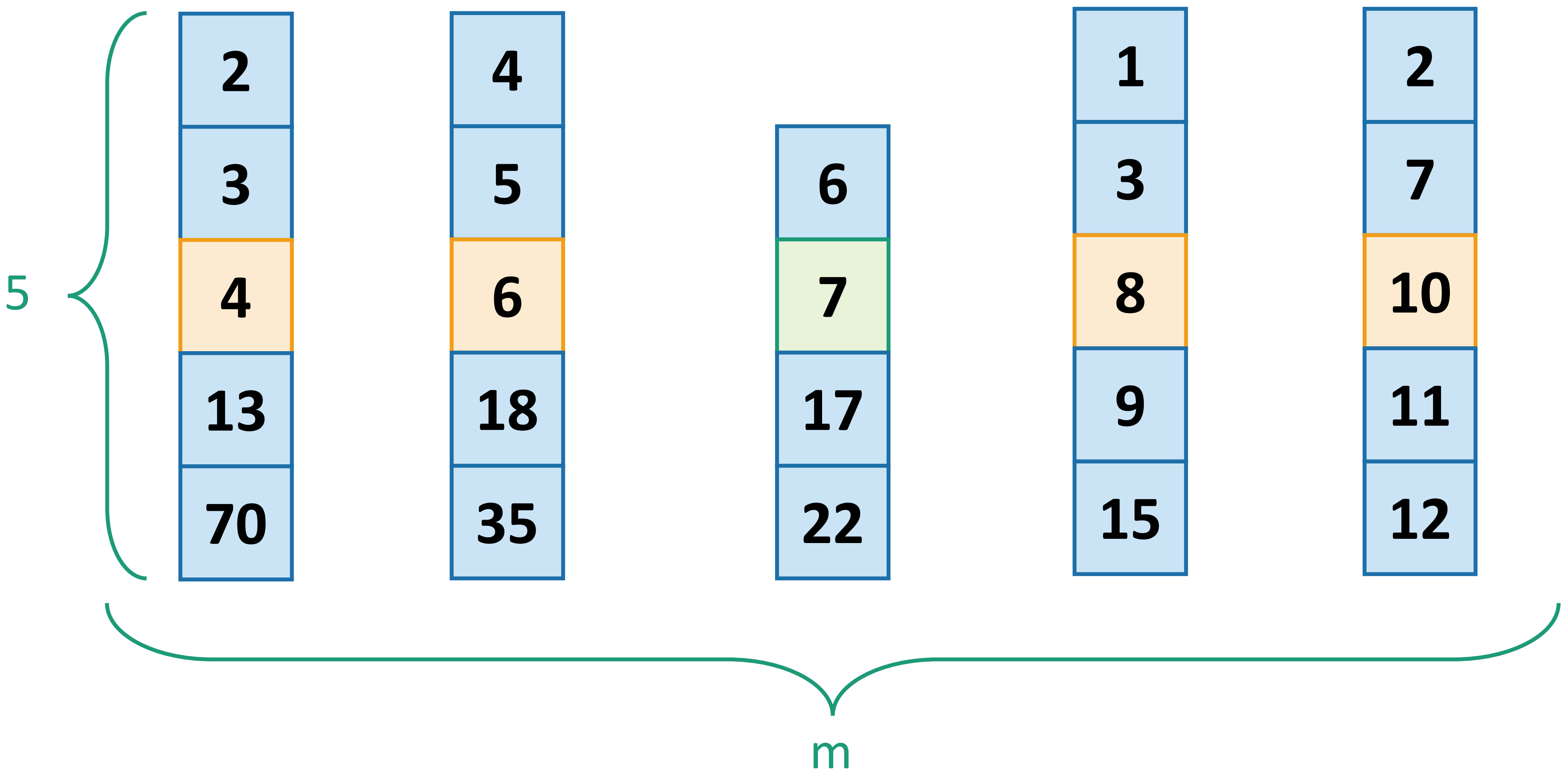
Proof by picture



The median of the medians is 7. That's our pivot!

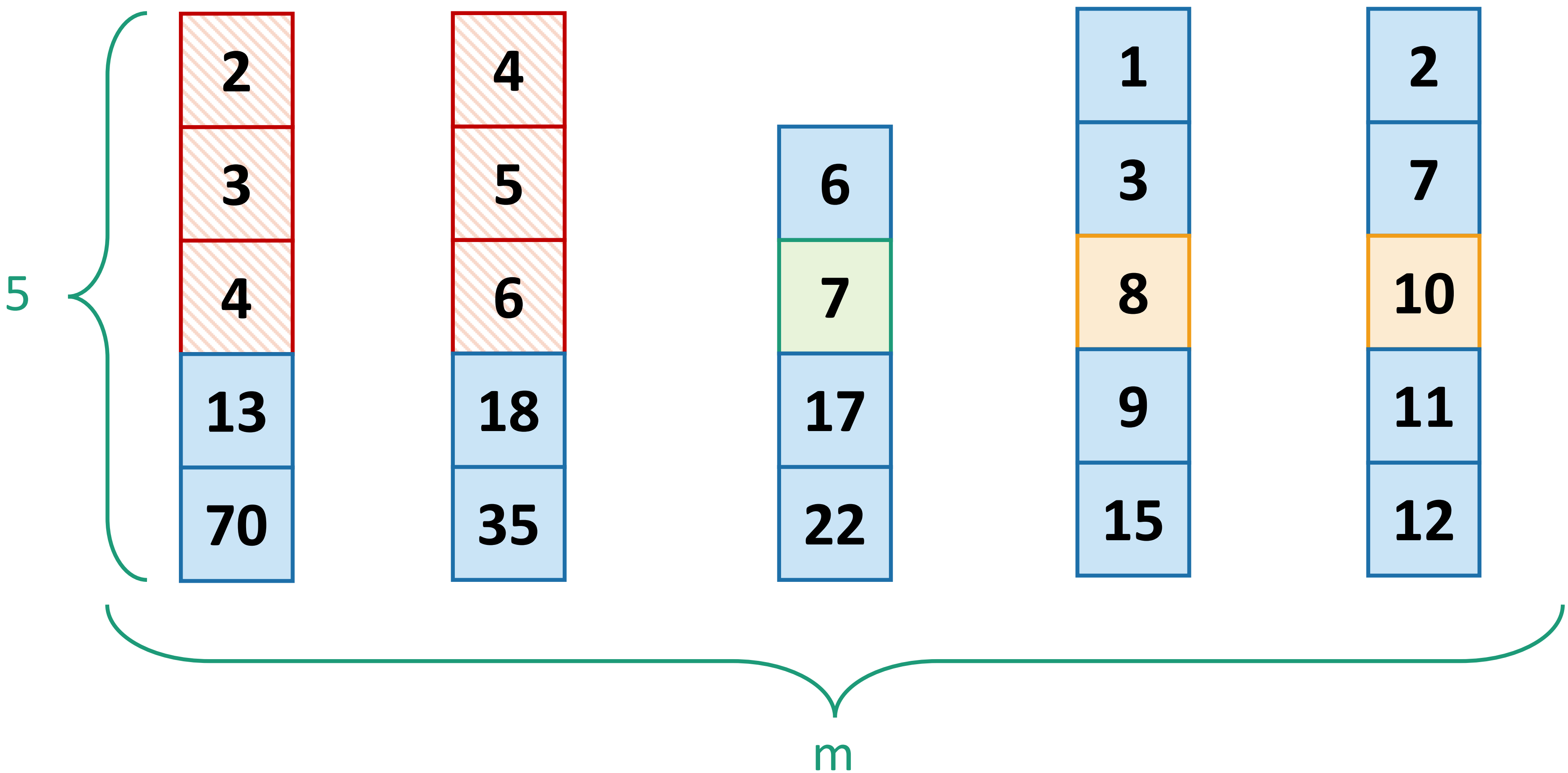
Proof by picture

We will show that lots of elements are smaller than the pivot, hence not too many are larger than the pivot.



How many elements are SMALLER than the pivot?

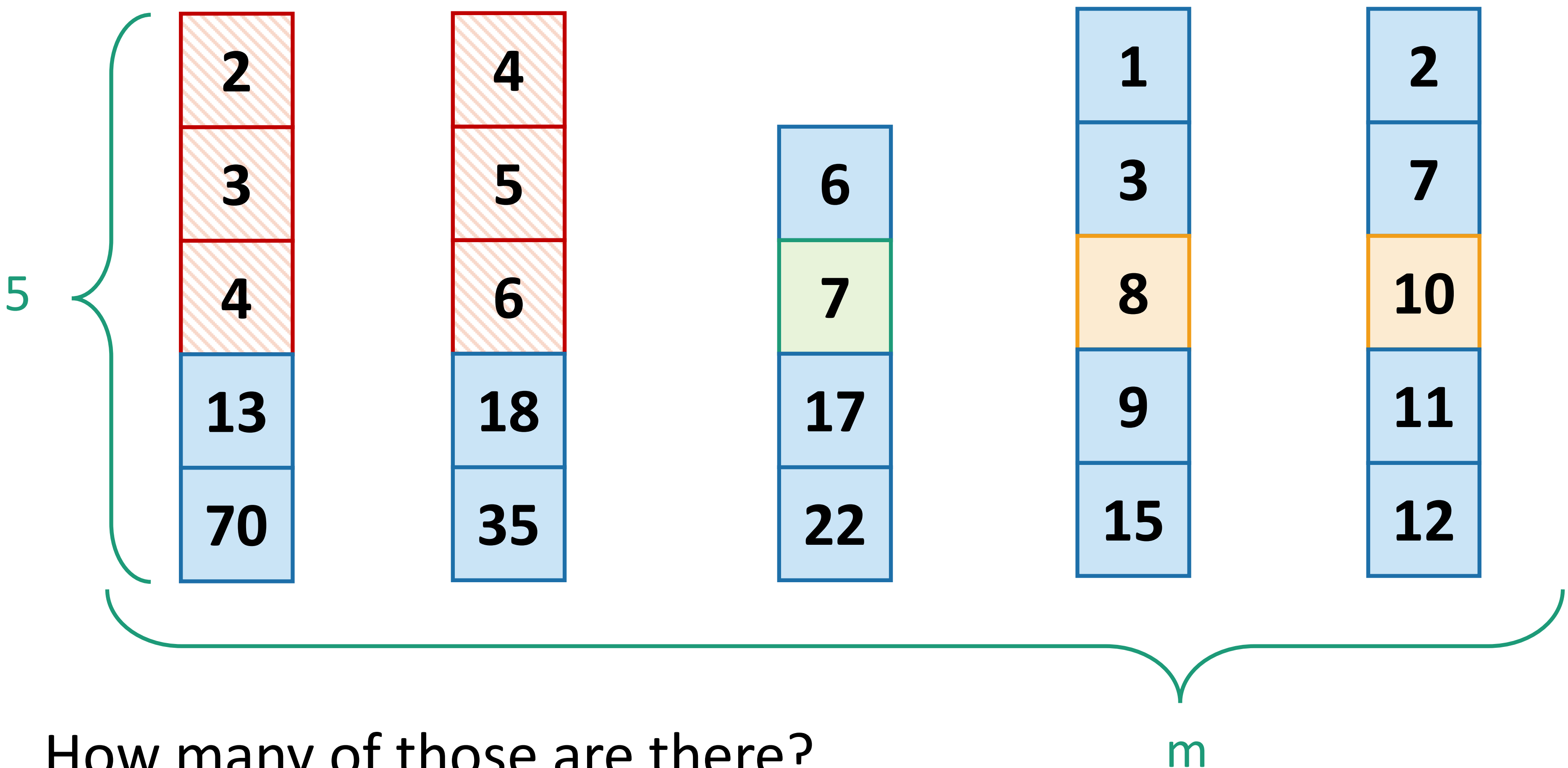
Proof by picture



At least these ones: everything above and to the left.

Proof by picture

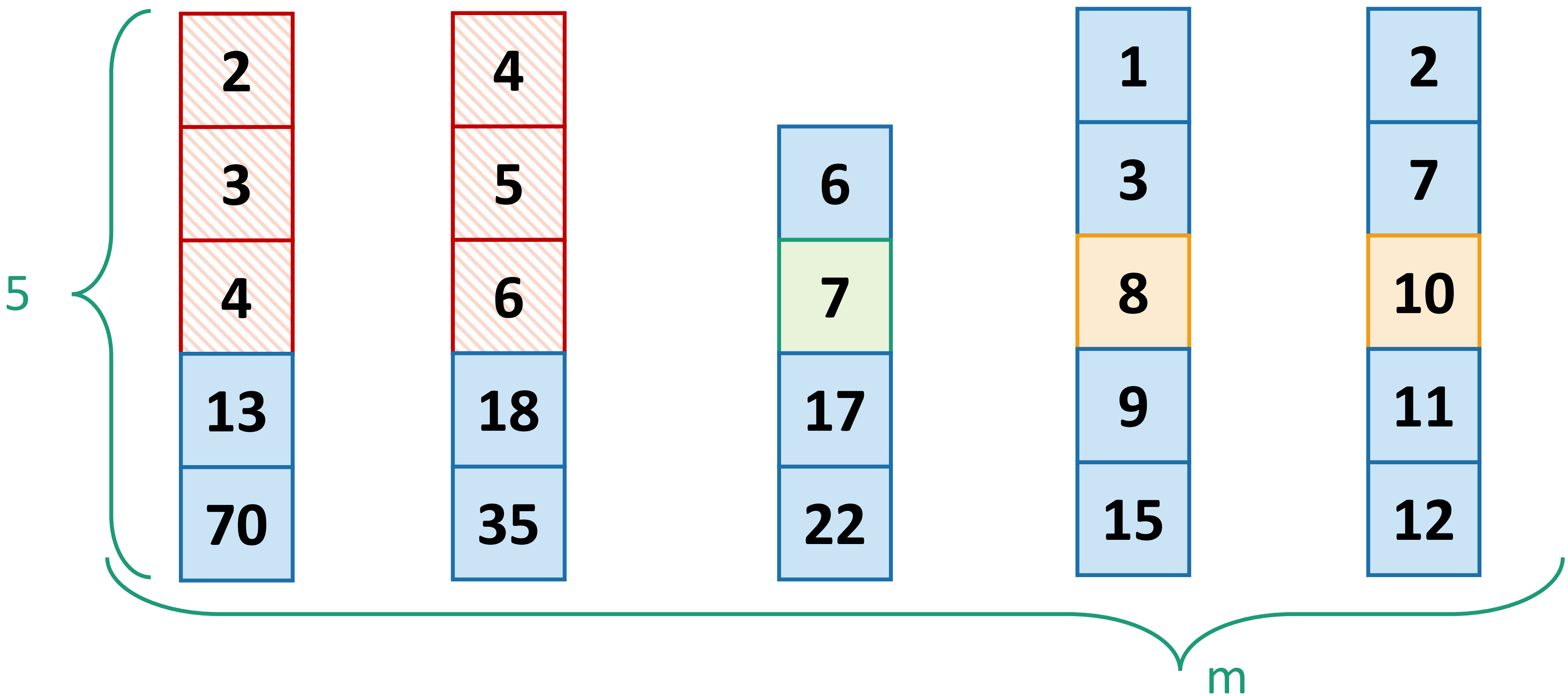
$3 \cdot \left(\left\lceil \frac{m}{2} \right\rceil - 1\right)$ of these, but then one of them could have been the "leftovers" group.



How many of those are there?

at least $3 \cdot \left(\left\lceil \frac{m}{2} \right\rceil - 2\right)$

Proof by picture



So how many are LARGER than the pivot? At most

$$n - 1 - 3\left(\left\lceil\frac{m}{2}\right\rceil - 2\right) \leq \frac{7n}{10} + 5$$

Remember $m = \left\lceil\frac{n}{5}\right\rceil$

(derivation on board)

That was one part of the lemma

- **Lemma:** If L and R are as in the algorithm SELECT given above, then

$$|L| \leq \frac{7n}{10} + 5$$

and

$$|R| \leq \frac{7n}{10} + 5$$

The other part is exactly the same.

The Plan

1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
    a) The outline of the algorithm.
    b) How to pick the pivot.
4. Return of the Substitution Method.
5. (If time) Proof of that Lemma.

Recap

Recap

- We introduced the Substitution Method to with problems with heterogeneous split of tasks.
- We saw a (pretty clever) algorithm to do SELECT in time O(n).
- We proved that it worked using the Substitution Method.
- The Master Theorem wouldn't have worked for this.
- In practice for this algorithm, it's often better to just choose the pivot randomly. You'll see an analysis of that if you take EECS278 (randomized algorithms).
- We'll also see some randomized algorithms…

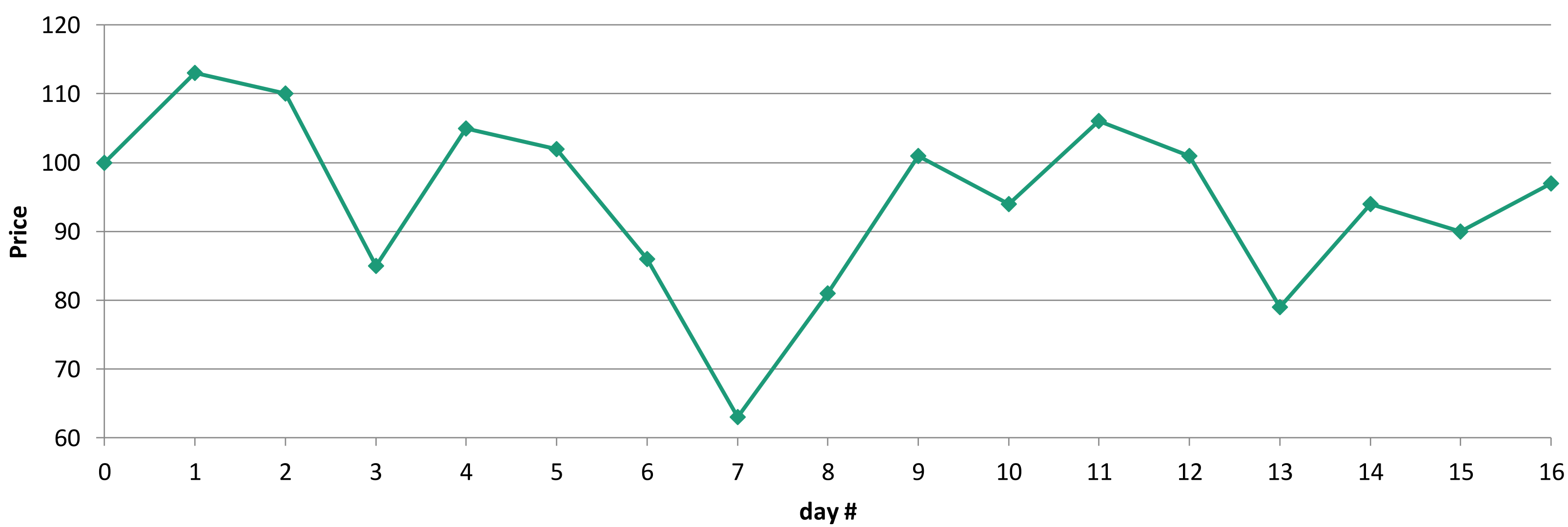
…in a few lectures

Today (part 2): more divide and conquer algorithms.

- The Maximum Subarray problem:
  - find the contiguous subarray within an array whose values have the largest sum.
- The Matrix Multiplication problem:
  - how do we efficiently multiply 2 large matrices?

Maximum-subarray problem background

- If you know the price of certain stock from day *i* to day *j*;
- You can only buy and sell one share once
- How to maximize your profit?



Brute Force

- What is the **brute-force** solution?

```
max = −∞;
for each day pair p {
   if(p.priceDifference > max)
         max=p.priceDifference;
}
```

Time complexity? $\binom{n}{2}$ pairs, so O(n^2)

Maximum-subarray problem

- If we know the price difference of each 2 contiguous days
- The solution can be found from the **maximum-subarray**
- **Maximum-subarray** of array A is:
  - A subarray of A
  - Nonempty
  - Contiguous
  - Whose values have the largest sum among all other possible subarrays

Example

Day	0	1	2	3	4
Price	10	11	7	10	6
Difference		1	-4	3	-4

What is the solution? Buy on day 2, sell on day 3

Can it be solved by the maximum-subarray of difference array?

Sub-array	0-1	0-2	0-3	0-4	1-2	1-3	1-4	2-3	2-4	3-4
Sum	1	-3	0	-4	-4	-1	-5	3	-1	-4

Divide-and-conquer algorithm

- How to divide?
  - Divide into 2 arrays
- What is the base case?
- How to combine the sub problem solutions to the current solution?
  - The process:
    - Divide array A[i, …, j] into A[i, …, mid] and A[mid+1, …, j]
    - A sub array must be in one of them (or across them!)
      - A[i, …, mid] // the left array
      - A[mid+1, …, j] // the right array
      - A[…, mid, mid+1,…] // the array across the midpoint
  - The maximum subarray is the largest sub-array among maximum subarrays of those 3

Basic Pseudocode

- Input: array A[i, …, j]
- Ouput: sum of maximum-subarray, start point of maximum-subarray, end point of maximum-subarray
- **FindMaxSubarray**:

1. if(j<=i) return (A[i], i, j);
2. mid = floor(i+j);
3. (sumLeft, startLeft, endLeft) = **FindMaxSubarray**(A, i, mid);
4. (sumRight, startRight, endRight) = **FindMaxSubarray**(A, mid+1, j);
5. (sumCross, startCross, endCross) = **FindMaxCrossingSubarray**(A, i, j, mid);
6. Return the largest one from those 3

Basic Pseudocode II

FindMaxCrossingSubarray(A, i, j, mid)

1. Scan A[i, mid] once, find the largest A[left, mid]
2. Scan A[mid+1, j] once, find the largest A[mid+1, right]
3. Return (sum of A[left, mid] and A[mid+1, right], left, right)

Let's do an example to make it clearer…

Target array : [1, -4, 3, 2]

All the sub arrays:

[1]

[-4]

[3]

[2]

[1, -4]

[-4, 3]

[3, 2]

[1, -4, 3]

[-4, 3, 2]

[1, -4, 3, 2]

Target array : [1, -4, 3, 2]

All the sub arrays:

[1] 1

[-4]

[3]

[2]

[1, -4]

[-4, 3]

[3, 2]

[1, -4, 3]

[-4, 3, 2]

[1, -4, 3, 2]

Target array :

1	-4	3	2

All the sub arrays:

1 — 1

-4 — -4

3

2

1, -4

-4, 3

3, 2

1, -4, 3

-4, 3, 2

1, -4, 3, 2

Target array :

1	-4	3	2

All the sub arrays:

1				1
	-4			-4
		3		3
			2	
1	-4			
	-4	3		
		3	2	
1	-4	3		
	-4	3	2	
1	-4	3	2	

Target array :

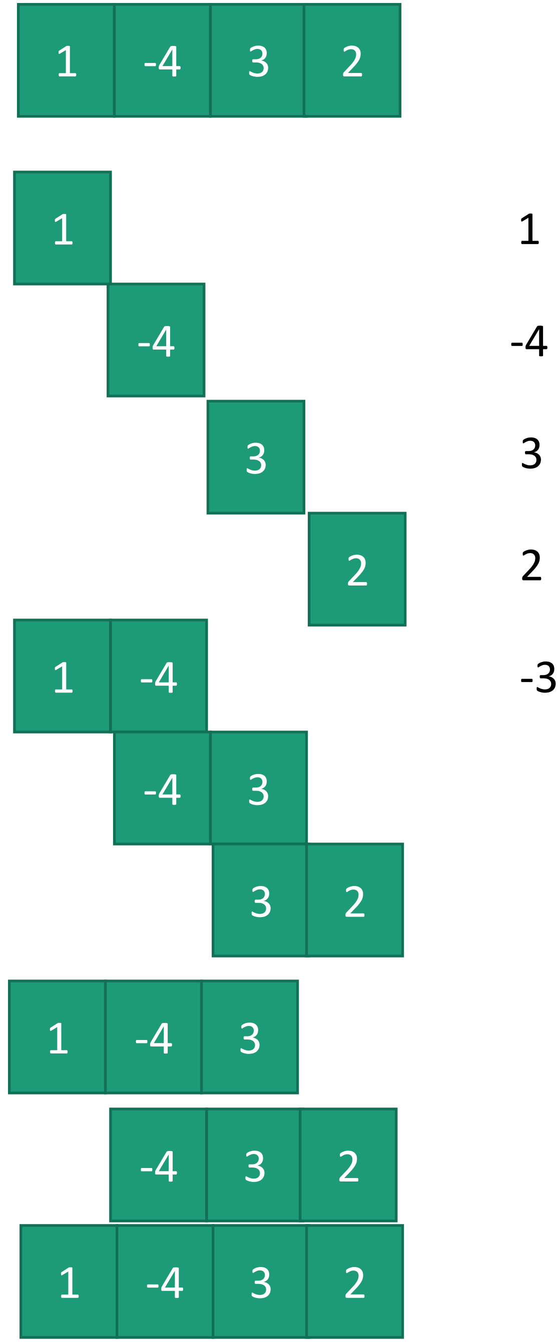
1	-4	3	2

All the sub arrays:

[1]	1
[-4]	-4
[3]	3
[2]	2
[1, -4]	
[-4, 3]	
[3, 2]	
[1, -4, 3]	
[-4, 3, 2]	
[1, -4, 3, 2]	

Target array :

All the sub arrays:



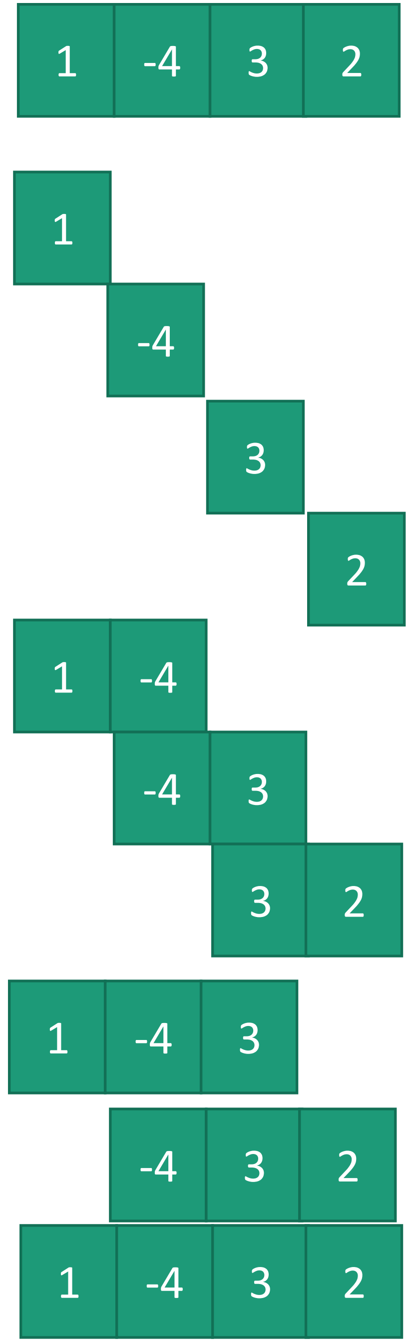
Target array : | 1 | -4 | 3 | 2 |

All the sub arrays:

1				1
	-4			-4
		3		3
			2	2
1	-4			-3
	-4	3		-1
		3	2	
1	-4	3		
	-4	3	2	
1	-4	3	2	

Target array :

All the sub arrays:



1

-4

3

2

-3

-1

5

Target array : | 1 | -4 | 3 | 2 |

All the sub arrays:

Sub array				Sum
1				1
	-4			-4
		3		3
			2	2
1	-4			-3
	-4	3		-1
		3	2	5
1	-4	3		0
	-4	3	2	
1	-4	3	2	

Target array : [1, -4, 3, 2]

All the sub arrays:

1	1
-4	-4
3	3
2	2
1, -4	-3
-4, 3	-1
3, 2	5
1, -4, 3	0
-4, 3, 2	1
1, -4, 3, 2	

Target array : [1, -4, 3, 2]

All the sub arrays:

Sub array	Sum
1	1
-4	-4
3	3
2	2
1, -4	-3
-4, 3	-1
3, 2	5
1, -4, 3	0
-4, 3, 2	1
1, -4, 3, 2	2

Target array :

1	-4	3	2

All the sub arrays:

Sub array	Sum
1	1
-4	-4
3	3
2	2
1, -4	-3
-4, 3	-1
3, 2 (Max!)	5
1, -4, 3	0
-4, 3, 2	1
1, -4, 3, 2	2

Target array :

1	-4	3	2

All the sub arrays:

The problem can be then solved by:

1				1
	-4			-4
		3		3
			2	2
1	-4			-3
	-4	3		-1
		3	2	5
1	-4	3		0
	-4	3	2	1
1	-4	3	2	2

Divide target array into 2 arrays

Target array : [1, -4, 3, 2]

All the sub arrays:

The problem can be then solved by:

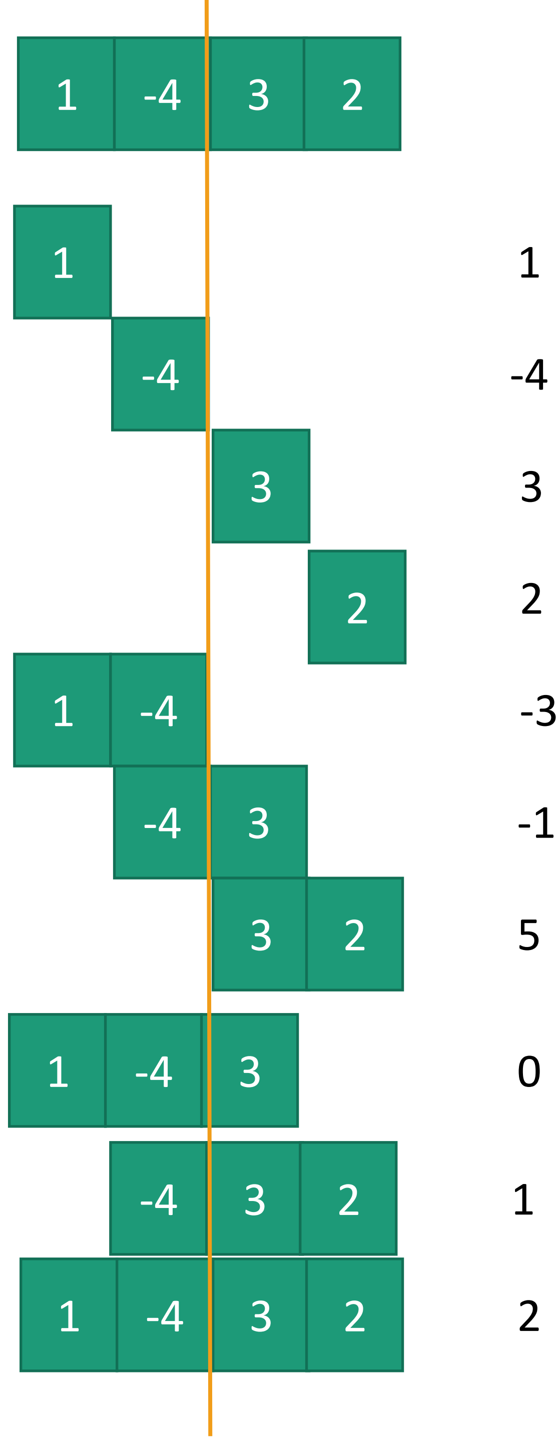
Sub array	Sum
1	1
-4	-4
3	3
2	2
1, -4	-3
-4, 3	-1
3, 2	5
1, -4, 3	0
-4, 3, 2	1
1, -4, 3, 2	2

Target array :

Divide target array into 2 arrays

All the sub arrays:

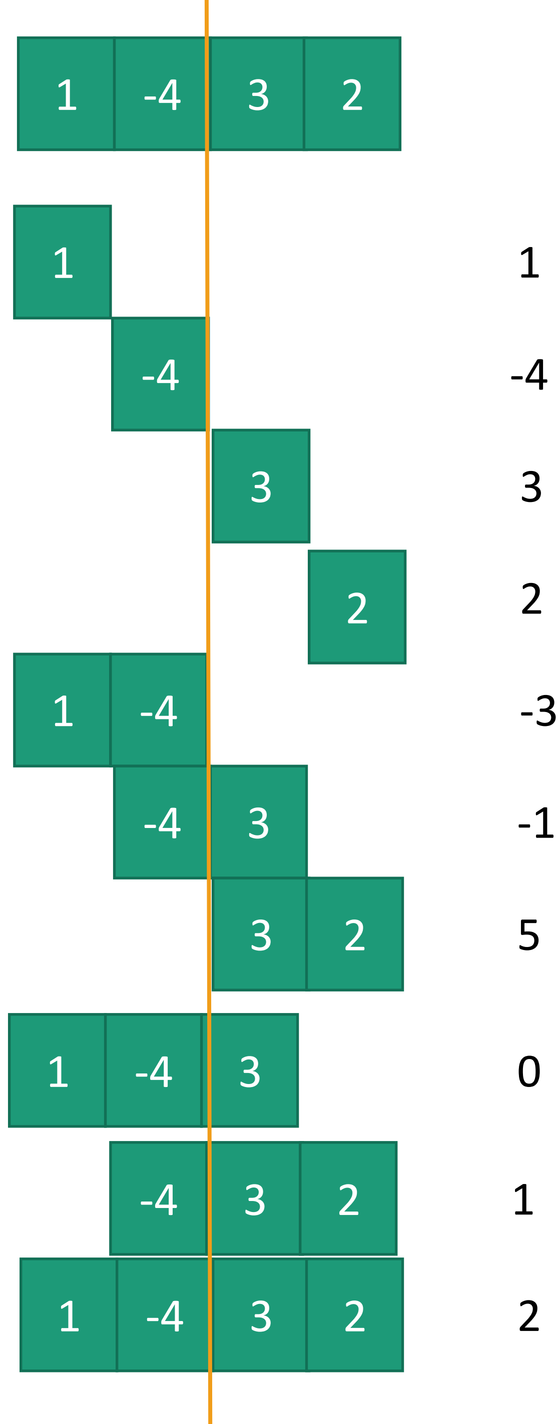
The problem can be then solved by:



Target array :

All the sub arrays:

The problem can be then solved by:



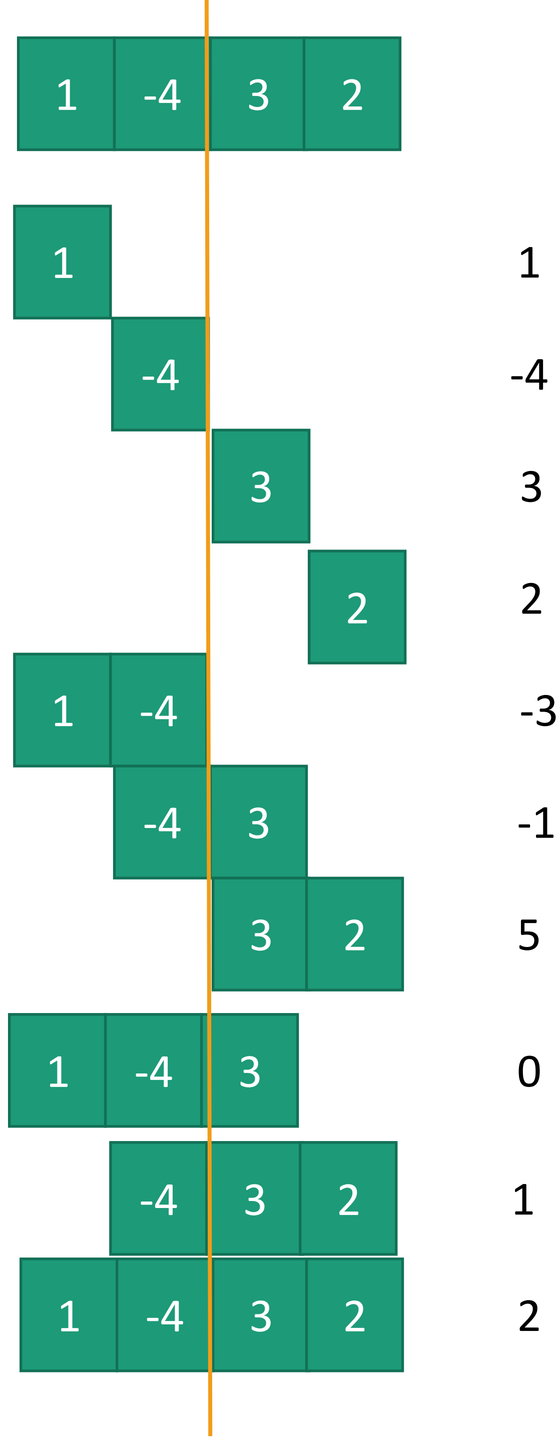
Divide target array into 2 arrays

We then have 3 types of subarrays:

Target array :

All the sub arrays:

The problem can be then solved by:



Divide target array into 2 arrays

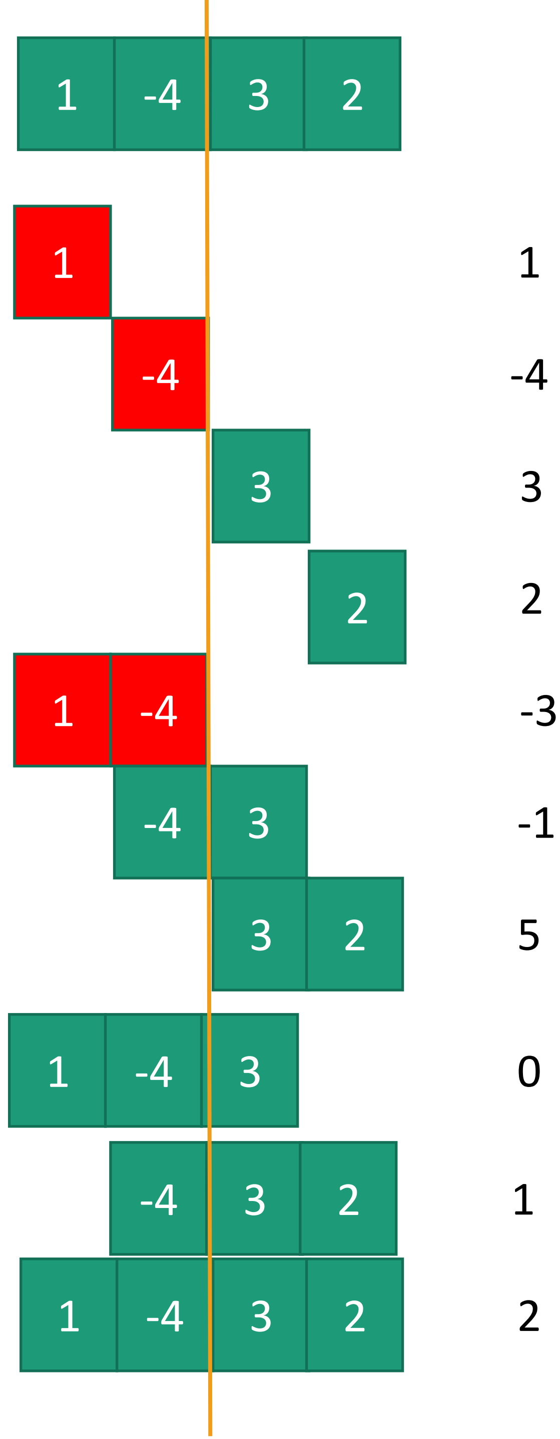
We then have 3 types of subarrays:

The ones belong to the left array

Target array :

All the sub arrays:

The problem can be then solved by:



Divide target array into 2 arrays

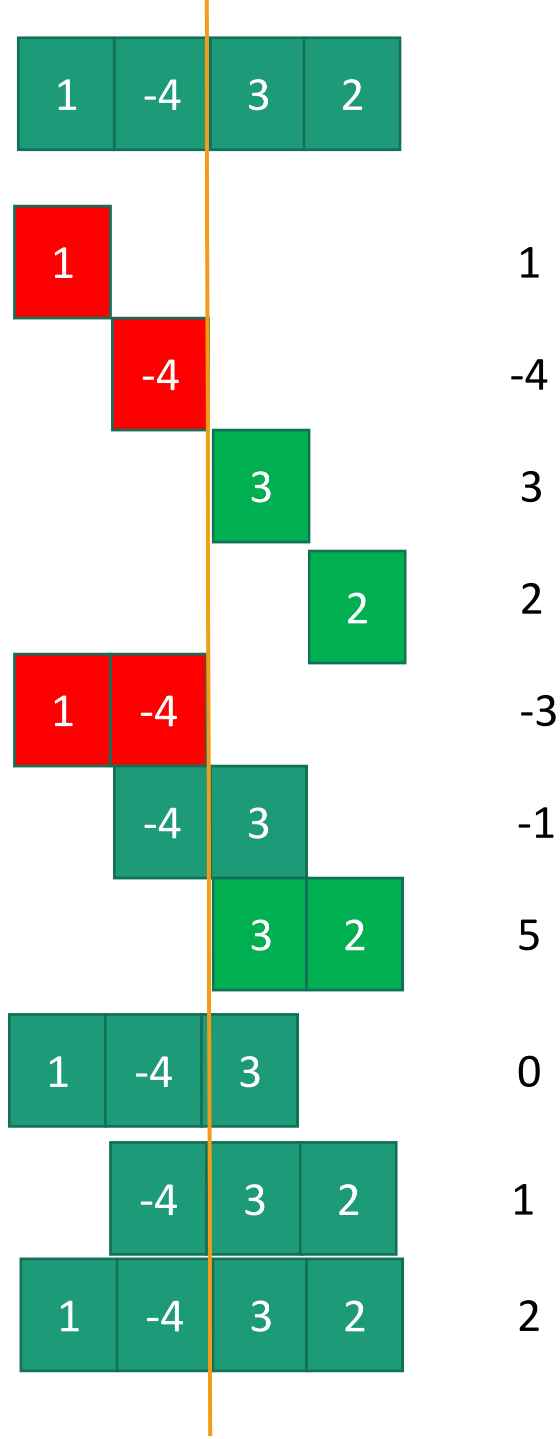
We then have 3 types of subarrays:

The ones belong to the left array

Target array :

All the sub arrays:

The problem can be then solved by:



Divide target array into 2 arrays

We then have 3 types of subarrays:

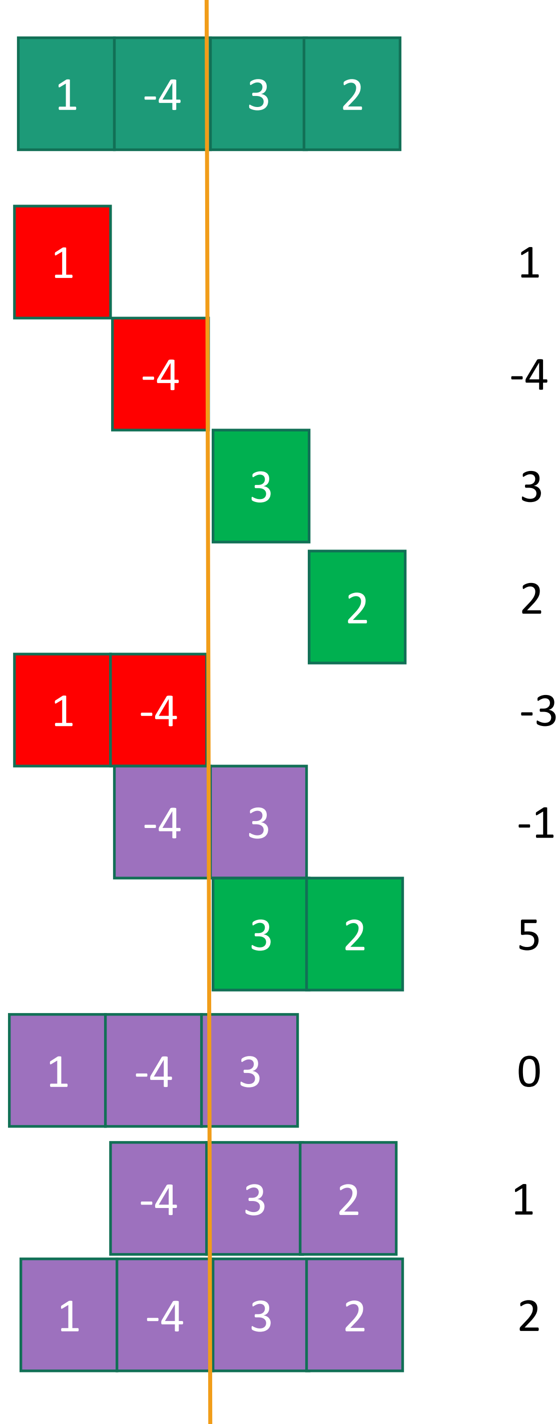
The ones belong to the left array

The ones belong to the right array

Target array :

All the sub arrays:

The problem can be then solved by:



Divide target array into 2 arrays

We then have 3 types of subarrays:

The ones belong to the left array

The ones belong to the right array

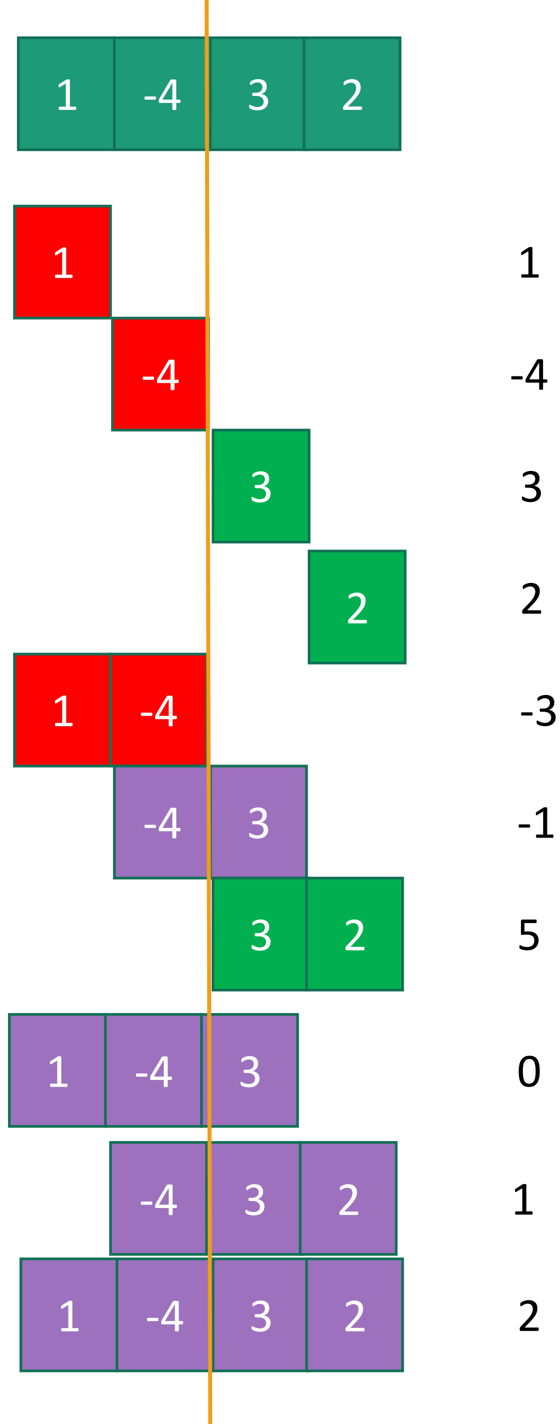
The ones crossing the mid point

Target array :

All the sub arrays:

The problem can be then solved by:

1. Find the max in left sub arrays



Divide target array into 2 arrays

We then have 3 types of subarrays:

The ones belong to the left array

The ones belong to the right array

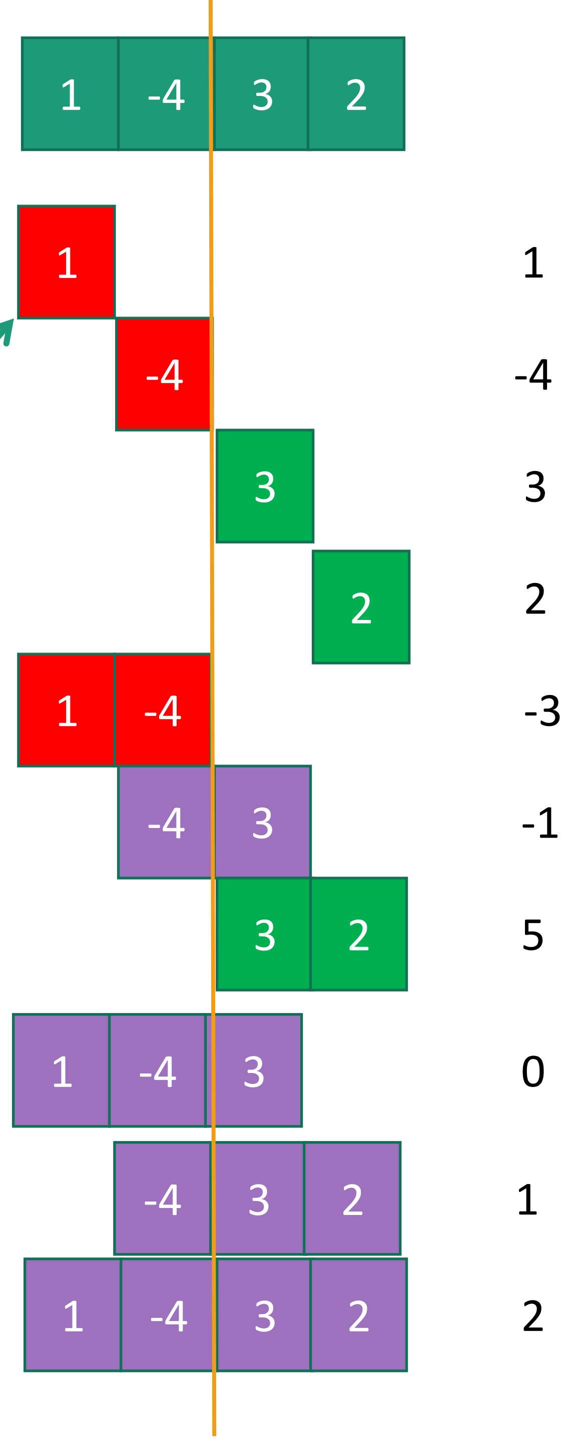
The ones crossing the mid point

Target array :

All the sub arrays:

The problem can be then solved by:

1. Find the max in left sub arrays



Divide target array into 2 arrays

We then have 3 types of subarrays:

The ones belong to the left array

The ones belong to the right array

The ones crossing the mid point

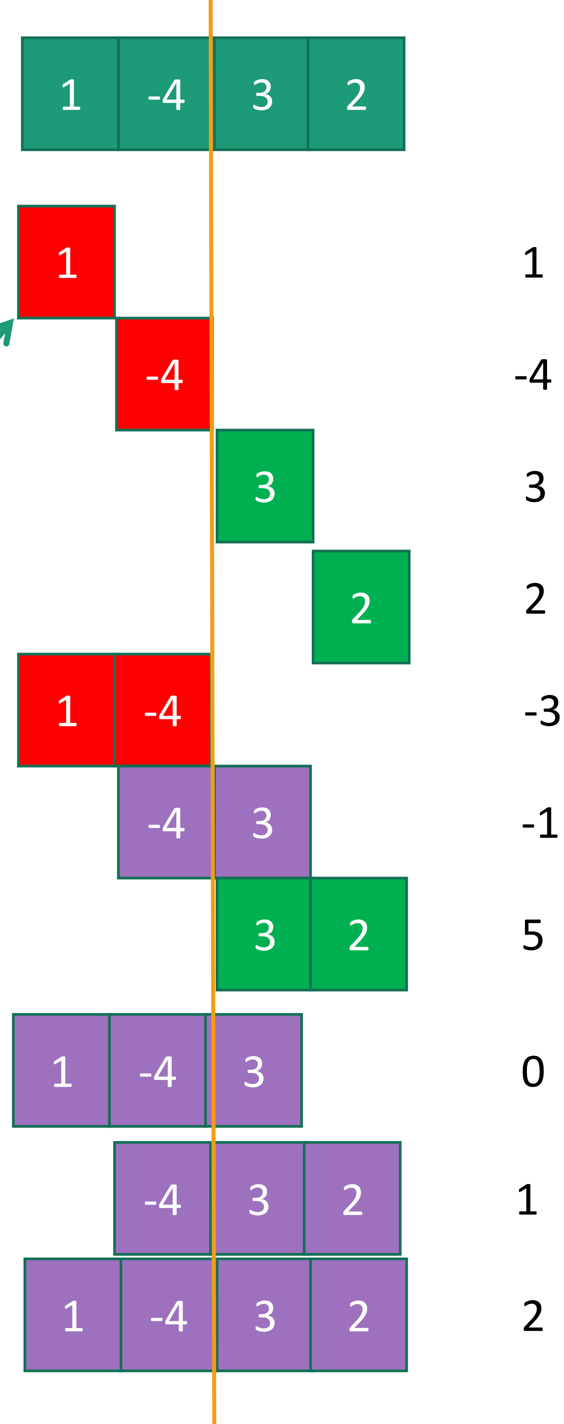
Target array :

All the sub arrays:

The problem can be then solved by:

1. Find the max in left sub arrays

2. Find the max in right sub arrays



Divide target array into 2 arrays

We then have 3 types of subarrays:

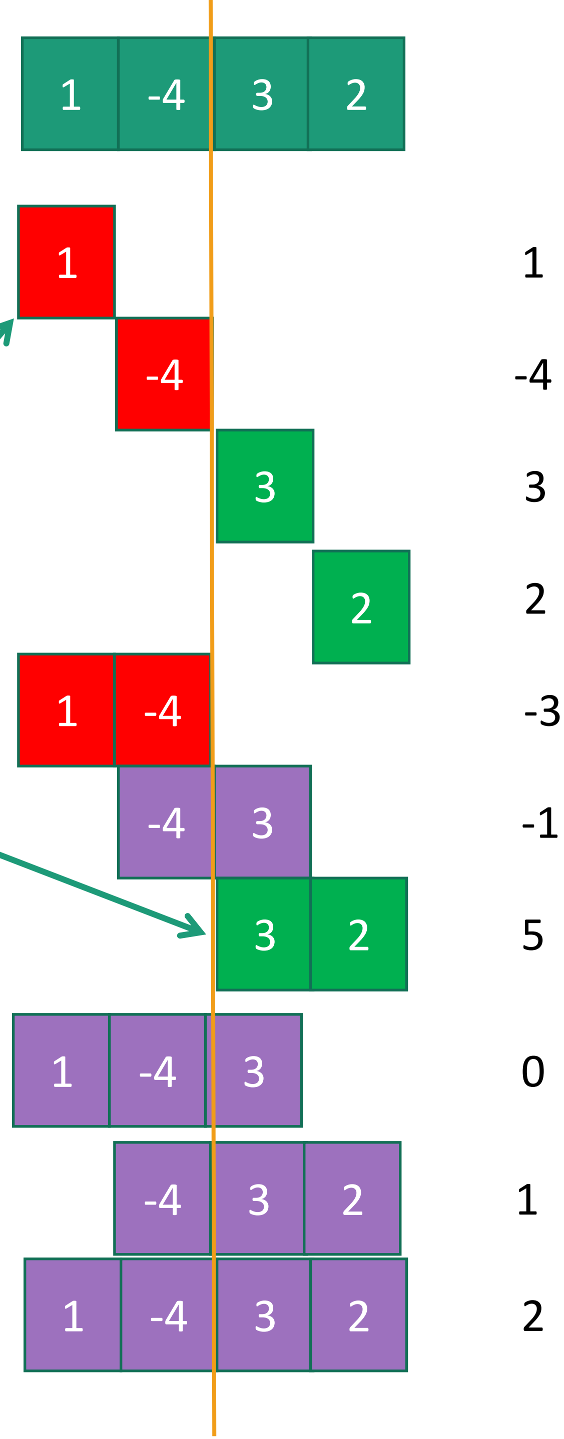
The ones belong to the left array

The ones belong to the right array

The ones crossing the mid point

Target array :

All the sub arrays:



The problem can be then solved by:

1. Find the max in left sub arrays

2. Find the max in right sub arrays

Divide target array into 2 arrays

We then have 3 types of subarrays:

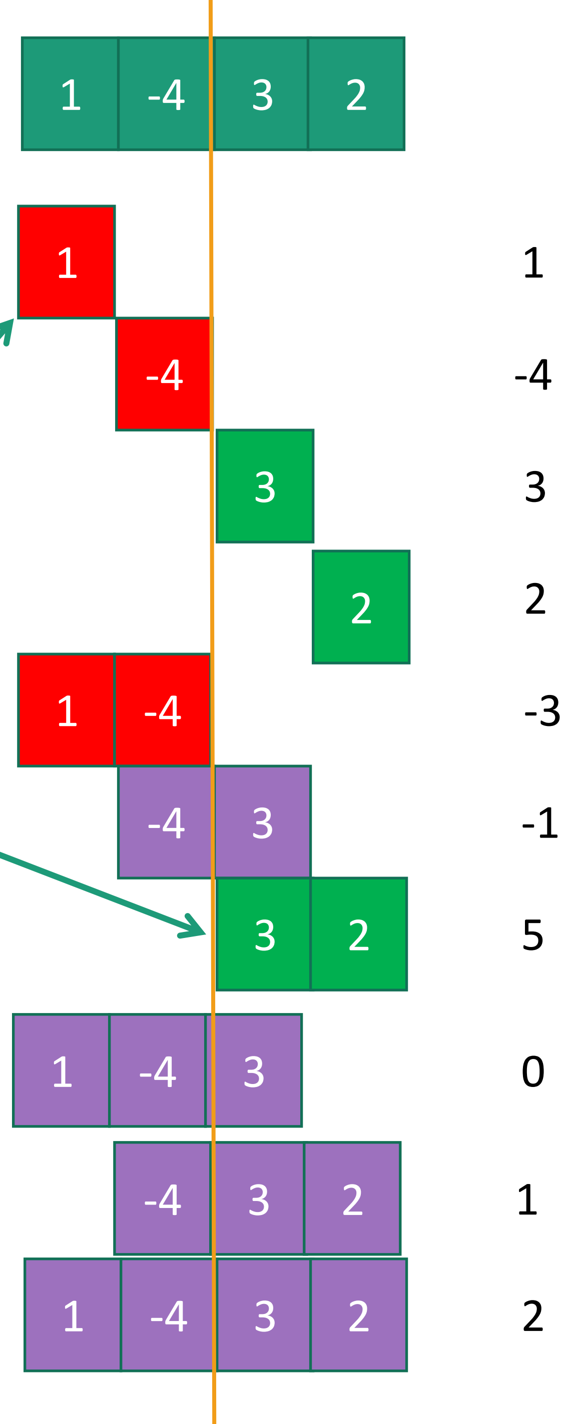
The ones belong to the left array

The ones belong to the right array

The ones crossing the mid point

Target array :

All the sub arrays:



The problem can be then solved by:

1. Find the max in left sub arrays

2. Find the max in right sub arrays

3. Find the max in crossing sub arrays

Divide target array into 2 arrays

We then have 3 types of subarrays:

The ones belong to the left array

The ones belong to the right array

The ones crossing the mid point

Target array :

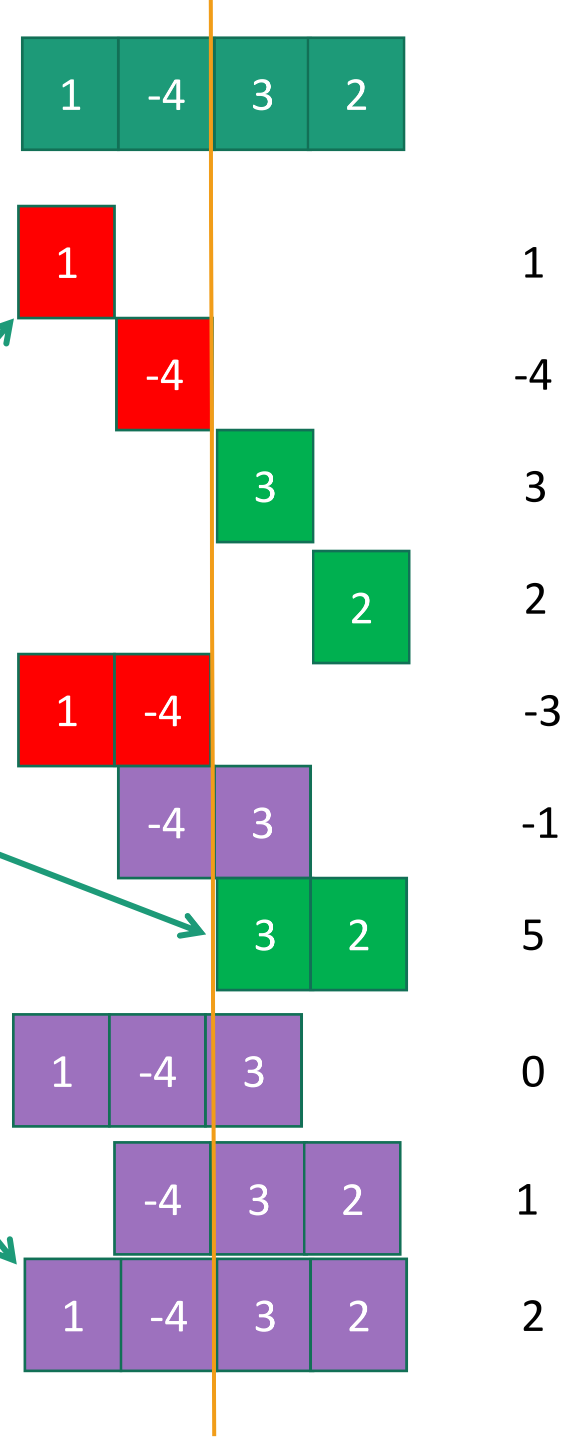
All the sub arrays:

The problem can be then solved by:

1. Find the max in left sub arrays

2. Find the max in right sub arrays

3. Find the max in crossing sub arrays



Divide target array into 2 arrays

We then have 3 types of subarrays:

The ones belong to the left array

The ones belong to the right array

The ones crossing the mid point

Target array :

All the sub arrays:

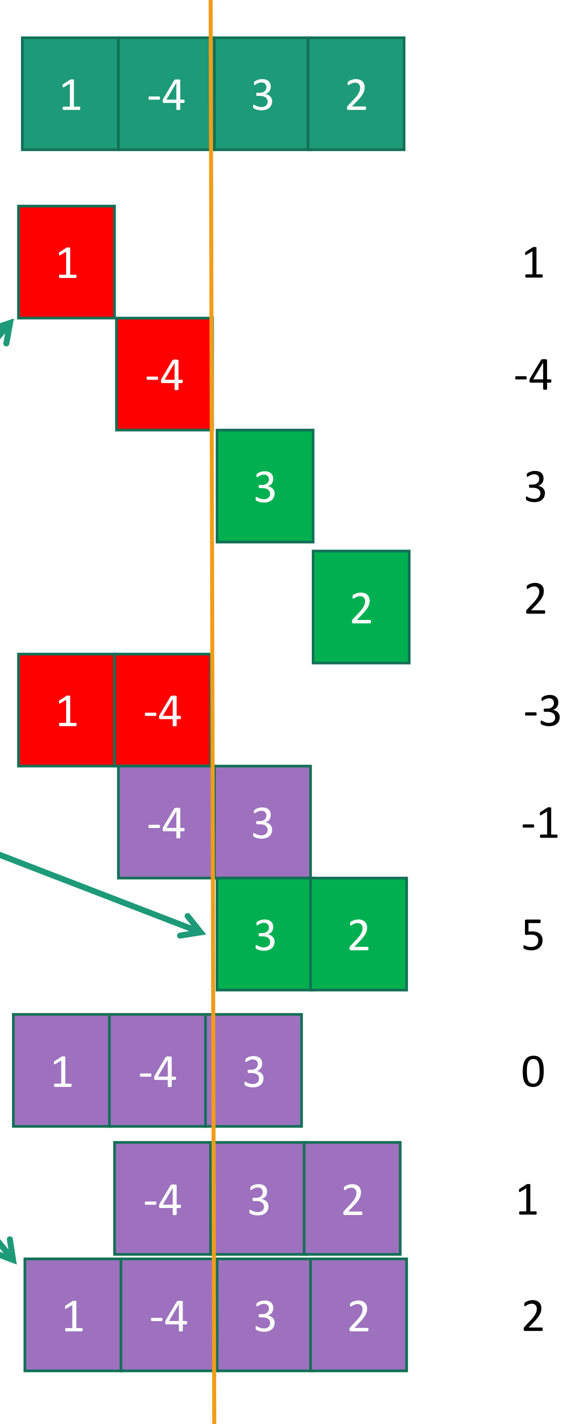
The problem can be then solved by:

1. Find the max in left sub arrays

2. Find the max in right sub arrays

3. Find the max in crossing sub arrays

4. Choose the largest one from those 3 as the final result



Divide target array into 2 arrays

We then have 3 types of subarrays:

The ones belong to the left array

The ones belong to the right array

The ones crossing the mid point

Target array :

All the sub arrays:

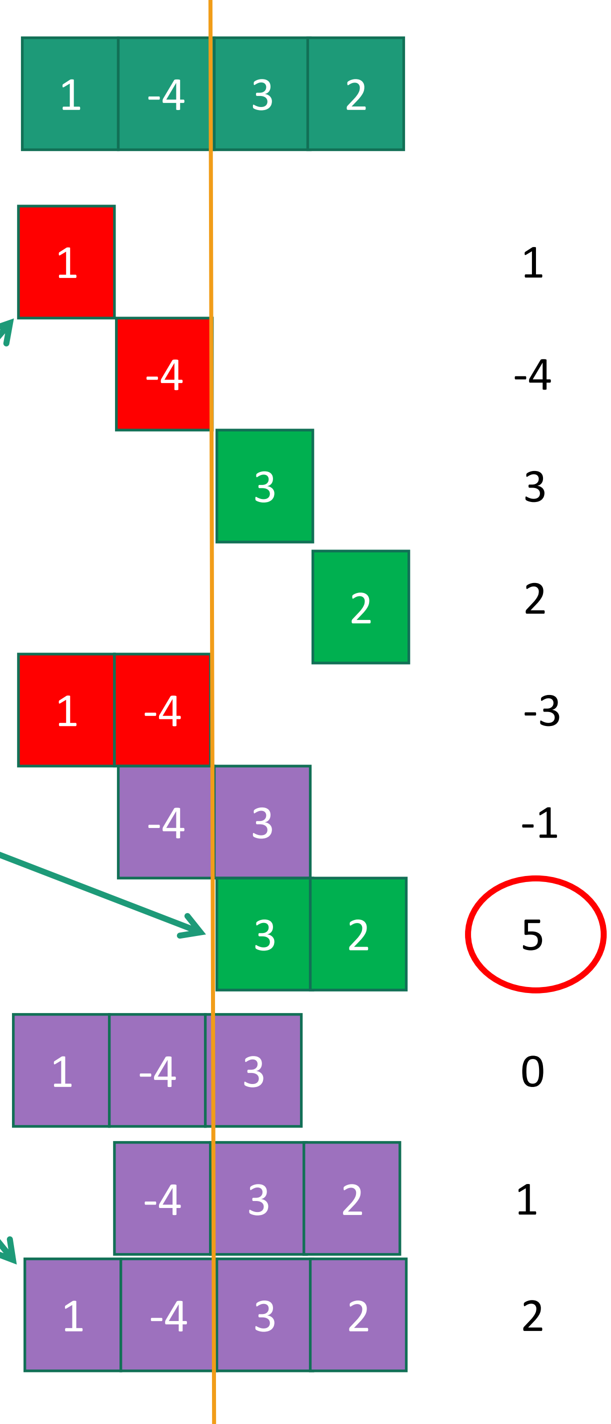
The problem can be then solved by:

1. Find the max in left sub arrays

2. Find the max in right sub arrays

3. Find the max in crossing sub arrays

4. Choose the largest one from those 3 as the final result



Divide target array into 2 arrays

We then have 3 types of subarrays:

The ones belong to the left array

The ones belong to the right array

The ones crossing the mid point

FindMaxSub (1 | -4 | 3 | 2)

1. Find the max in left sub arrays **FindMaxSub** (1 | -4)

2. Find the max in right sub arrays **FindMaxSub** (3 | 2)

3. Find the max in crossing sub arrays

   Scan 1 | -4 once, and scan 3 | 2 once

4. Choose the largest one from those 3 as the final result

3. Find the max in crossing sub arrays

1	-4	3	2

3. Find the max in crossing sub arrays

1	-4	3	2

3. Find the max in crossing sub arrays

1	-4	3	2

-4

Sum=-4

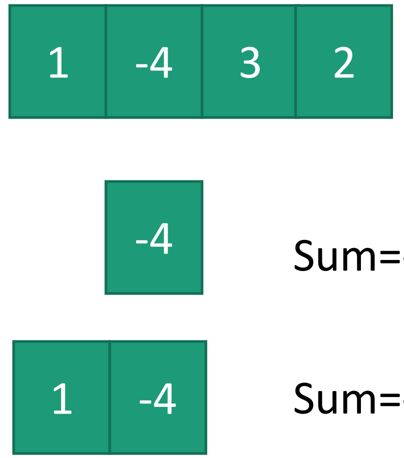
3. Find the max in crossing sub arrays

1	-4	3	2

-4

Sum=-4

3. Find the max in crossing sub arrays



Sum=-4

Sum=-3

3. Find the max in crossing sub arrays

1	-4	3	2

-4

Sum=-4

1	-4

Sum=-3 largest

3. Find the max in crossing sub arrays

1	-4	3	2

-4

Sum=-4

1	-4

Sum=-3 largest

3. Find the max in crossing sub arrays

1	-4	3	2

-4	Sum=-4

1	-4	Sum=-3	largest

3	Sum=3

3. Find the max in crossing sub arrays

1	-4	3	2

-4

Sum=-4

1	-4

Sum=-3 largest

3

Sum=3

3. Find the max in crossing sub arrays

1	-4	3	2

-4

Sum=-4

1	-4

Sum=-3 largest

3

Sum=3

3	2

Sum=5

3. Find the max in crossing sub arrays

1	-4	3	2

-4

Sum=-4

1	-4

Sum=-3 largest

3

Sum=3

3	2

Sum=5 largest

3. Find the max in crossing sub arrays

1	-4	3	2

-4 Sum=-4

1 -4 Sum=-3 largest

3 Sum=3

3 2 Sum=5 largest

The largest crossing subarray is :

3. Find the max in crossing sub arrays

1	-4	3	2

-4

Sum=-4

Sum=-3 largest

3

Sum=3

3	2

Sum=5 largest

The largest crossing subarray is :

1	-4

3. Find the max in crossing sub arrays

1	-4	3	2

-4 Sum=-4

Sum=-3 largest

3 Sum=3

Sum=5 largest

The largest crossing subarray is :

1	-4	3	2

3. Find the max in crossing sub arrays

1	-4	3	2

-4

Sum=-4

Sum=-3 largest

3

Sum=3

Sum=5 largest

The largest crossing subarray is :

1	-4	3	2

Sum=2

Time Complexity

- What is the time complexity?
- **FindMaxSubarray**:

1. if(j<=i) return (A[i], i, j);
2. mid = floor(i+j);
3. (sumCross, startCross, endCross) = **FindMaxCrossingSubarray**(A, i, j, mid);
4. (sumLeft, startLeft, endLeft) = **FindMaxSubarray**(A, i, mid);
5. (sumRight, startRight, endRight) = **FindMaxSubarray**(A, mid+1, j);
6. Return the largest one from those 3

Linear work in n, Θ(n)

T(n/2)

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

Time Complexity II

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

- Remember the Master Theorem
- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

- Our recurrence formula has the appropriate format!

  a : number of subproblems

  b : factor by which input size shrinks

  d : need to do n^d work to create all the subproblems and combine their solutions.

  - a=2, b=2, d=1 (top case)
- T(n) = O(n.log(n))