# CompSci 260P
# Spring 2023 Lecture 14:
# Divide and Conquer I

# Min and Max

- We have an array of $n$ distinct numbers.
- We want to find *both* – the min and max.
- Brute force method takes $2n - 3$ comparisons.
- Find a way that uses strictly fewer.

Pair up; // $\frac{n}{2}$ comparisons

$\frac{n}{2}$ winners → find max for max $\frac{n}{2} - 1$

$\frac{n}{2}$ non-winners find min for min $\frac{n}{2} - 1$

$\frac{3n}{2} - 2$

# Could anyone do better?

- Adversary argument:
  - All queries are made to an adversary (opponent)
  - Adversary is allowed to make up answers
  - But answers must be consistent with some input
- If we compare and find $a < b$, we say:
  - $a$ lost the competition
  - $b$ won the competition
- Every non-max loses at least one
- Every non-min wins at least one
- This is $2n - 2$ units of information.

# What should the adversary do?

We compare $a$ and $b$ to gain information.

- If $a, b$ never compared (to ANY key) before?

  I gain 2 units of information

- If exactly one of them compared before?

  I gain one if adversary sets answer

- Both compared before, one won at least once?

- Both compared before, both lost before?

# How many comparisons can be forced?

- We need to gain $2n - 2$ units of information.
- $c_1 = \#$ comparisons that gave us one unit.
- $c_2 = \#$ comparisons that gave us two units.
- Total units of info available is at least $2n - 2$

$$c_1 + 2c_2 \geq 2n - 2$$

- At most $n/2$ comparisons give us two units

$$-c_2 \geq -n/2$$

$$c_1 + c_2 \geq \frac{3}{2}n - 2$$

# Counting Inversions

- $i, j$ are an *inverted pair* if $i < j$ and $A[i] > A[j]$. (the larger element appears earlier in the array)

For example:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 85 | 24 | 63 | 45 | 17 | 31 | 96 | 50 |

and more

# Counting Inversions

- $i, j$ are an *inverted pair* if $i < j$ and $A[i] > A[j]$. (the larger element appears earlier in the array)

The following is an $\Theta(n^2)$ time way to count the inversions in an array:

```
count = 0
for i = 1 ... n do
    for j = i + 1 ... n do
        if A[i] > A[j] then
            count++
return count
```

BRUTE FORCE

& POLYNOMIAL TIME

TRY D&C

# Counting Inversions Faster: a subproblem

merge - and - count

- want to count number of inverted pairs in $A$,
- we know $A[1 \ldots \frac{n}{2}]$ is sorted, as is $A[\frac{n}{2}+1 \ldots n]$.
- Can we do better than $\Theta(n^2)$?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 24 | 45 | 63 | 85 | 17 | 31 | 50 | 96 |

i j

k = 1

Count = 0; i = 1; j = n/2 + 1; declare T[1...n]

while i ≤ n/2 and j ≤ n

if A[j] < A[i] : T[k] = A[j]

Count += # elts left in 1st half

j++; k++;

else: T[k] = A[i]; i++; k++;

# Finishing the Merge Portion

- We want sorted list when done
- Let's keep the rest of the array

```
//after while loop
while j ≤ n
    T[k] = A[j]; k++; j++;
while i ≤ n/2
    T[k] = A[i]; k++; i++;
copy T to A
return count
```

# Counting Inversions Faster

also sorts A when done

- Use the algorithm from the previous question
- count number of inversions in *unsorted* array
- How fast is your algorithm? ?

count (A)

if |A| "small"

brute force count.
Then sort and return

$C_L$ = count (A [1 ... n/2])

$C_R$ = count (A [n/2+1 ... n])

$C_m$ = merge-and-count (A); return $C_L + C_R + C_m$

# Running Time for Counting Inversions

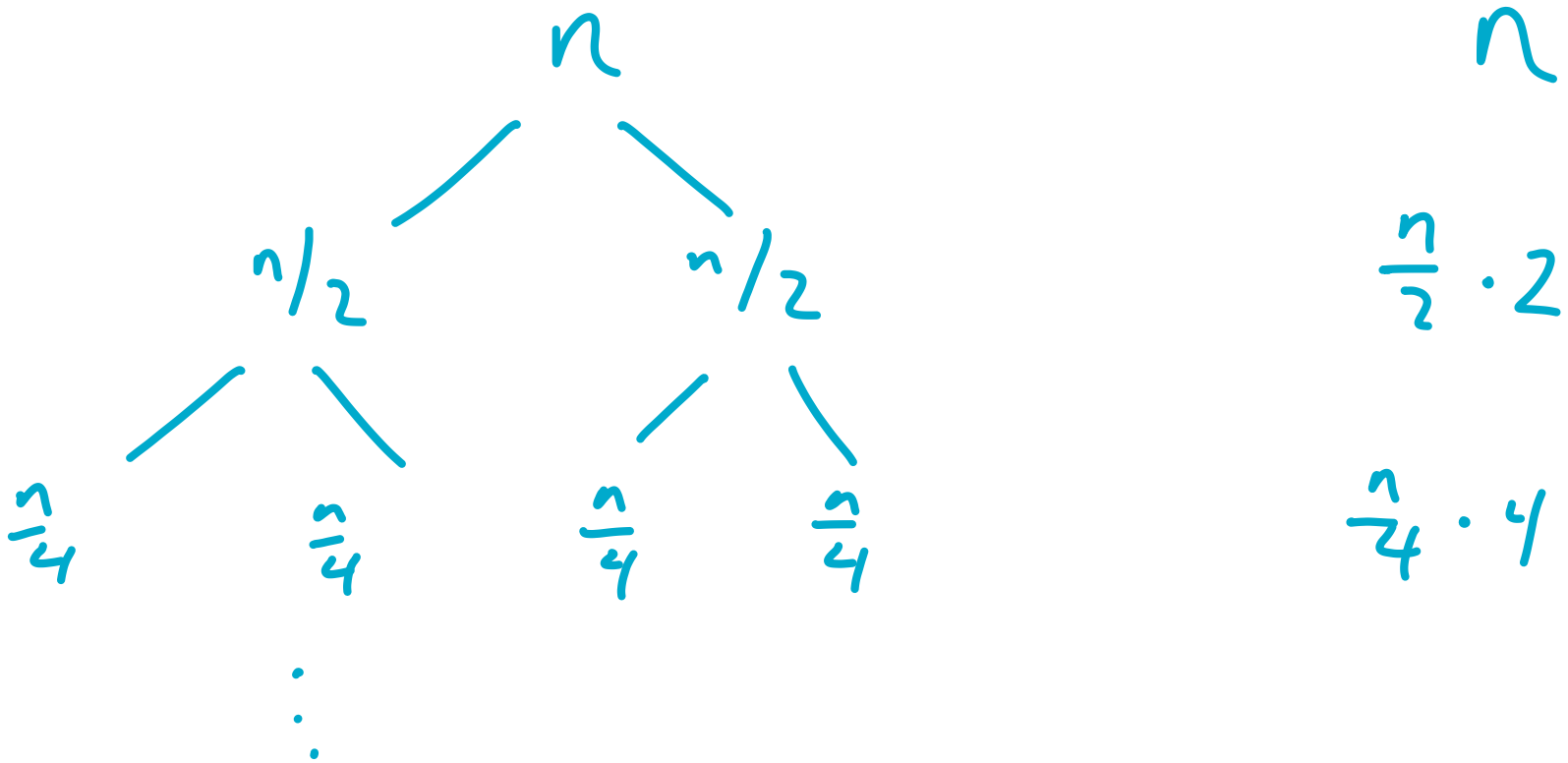
**if** list has one or zero elements **then**
  **return** no inversions
Divide into $L = A[1 \ldots \frac{n}{2}]$ and $R = A[\frac{n}{2} + 1 \ldots n]$
// Should this be slices or parameters?
Recursively solve on $L$; count is $c_L$
Recursively solve on $R$; count is $c_R$
Run earlier subproblem on $L, R$; count is $c_M$
**return** $c_L + c_R + c_M$

How long does this take?

$T(n) = 2T(n/2) + \Theta(n)$

# Running Time for Counting Inversions

- Two recursive of size $n/2$, plus local linear work
- $T(n) = 2T(n/2) + n$



# The Master Theorem

A common running time for a D&C algorithm:
$T(n) = aT(n/b) + f(n)$
($a \geq 1$, $b > 1$, $f(n)$ is asymptotically positive)

- If there is a small constant $\varepsilon > 0$ such that $f(n)$ is $\mathcal{O}(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
- For example, express $T(n) = 4T(n/2) + n$

$a = 4$ $b = 2$ $f(n) = n$

$\log_b a = 2$

is $n$ $O(n^{\log_2 4 - \varepsilon})$

$n$ is $O(n^{1.9})$? ☺ Yes.

# The Master Theorem

A common running time for a D&C algorithm:
$T(n) = aT(n/b) + f(n)$
($a \geq 1$, $b > 1$, $f(n)$ is asymptotically positive)

- If there is a constant $k \geq 0$, such that $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
- For example, express $T(n) = 2T(n/2) + n \log n$

$a = 2$ $b = 2$ $f(n) = n \log n$ $k = 1$?

$\log_2 2 = 1$

is $n \log n$ $\Theta(n^1 \cdot \log n)$ ✓

So $T(n)$ is $\Theta(n \log^2 n)$

# The Master Theorem

A common running time for a D&C algorithm:
$T(n) = aT(n/b) + f(n)$
($a \geq 1$, $b > 1$, $f(n)$ is asymptotically positive)

- If there is a small constant $\varepsilon > 0$ such that $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$.
- For example, express $T(n) = T(n/3) + n$

$a = 1$ $b = 3$ $f(n) = n$

$\log_b a = 0$

is $n$ $\Omega(n^{0+\varepsilon})$? ✓

So $T(n)$ is $\Theta(n)$

# The Master Theorem

- $T(n) = 9T(n/3) + n^{2.5}$

$a = 9$ $b = 3$ $f(n) = n^{2.5}$

$\log_3 9 = 2$

is $n^{2.5}$ $\Omega(n^{2+\varepsilon})$? Yes.

$T(n)$ is $\Theta(n^{2.5})$