CS 6212 DESIGN AND ANALYSIS OF ALGORITHMS

LECTURE: DIVIDE & CONQUER – PART I

Instructor: Abdou Youssef

OBJECTIVES OF THIS LECTURE

By the end of this lecture, you will be able to:

- Describe the Divide & Conquer algorithmic design technique
- Apply the technique to designing algorithms for an important problem, Sorting, in two different ways
- Draw and appreciate the strong connection between recursion and Divide & Conquer
- Carry out time complexity analysis of Divide & Conquer algorithms, by deriving and solving recurrence relations
- Perform worst-case and average-case time complexity analysis

OUTLINE

- Template for Divide and Conquer
- First Application: Mergesort
- Second Application: Quicksort

DIVIDE & CONQUER

-- GENERAL STRATEGY AND UNDERLYING PHILOSOPHY --

- The general strategy is
  - Examine the size or magnitude of the input of the problem
  - If small enough, solve the problem directly
    - Such solutions are fairly simple, and often trivial, for small input
  - If not small, divide the input into two or more (smaller) parts
  - Solve the same problem on each part
    - by calling the algorithm recursively on each part
    - which is a huge saving in intellectual/design effort
  - Merge the subsolutions (i.e., the solutions of the parts) into a global solution
    - Merging subsolutions is usually simpler than finding a global solution from scratch

DIVIDE & CONQUER
-- TEMPLATE --

Template divide&conquer (input I)

begin

  if (size or value of input is small enough)

  then

    solve directly and **return**;

  endif

  divide input I into two or more parts I_1, I_2,…;

  $S_1 \leftarrow$ divide&conquer(I_1) ;

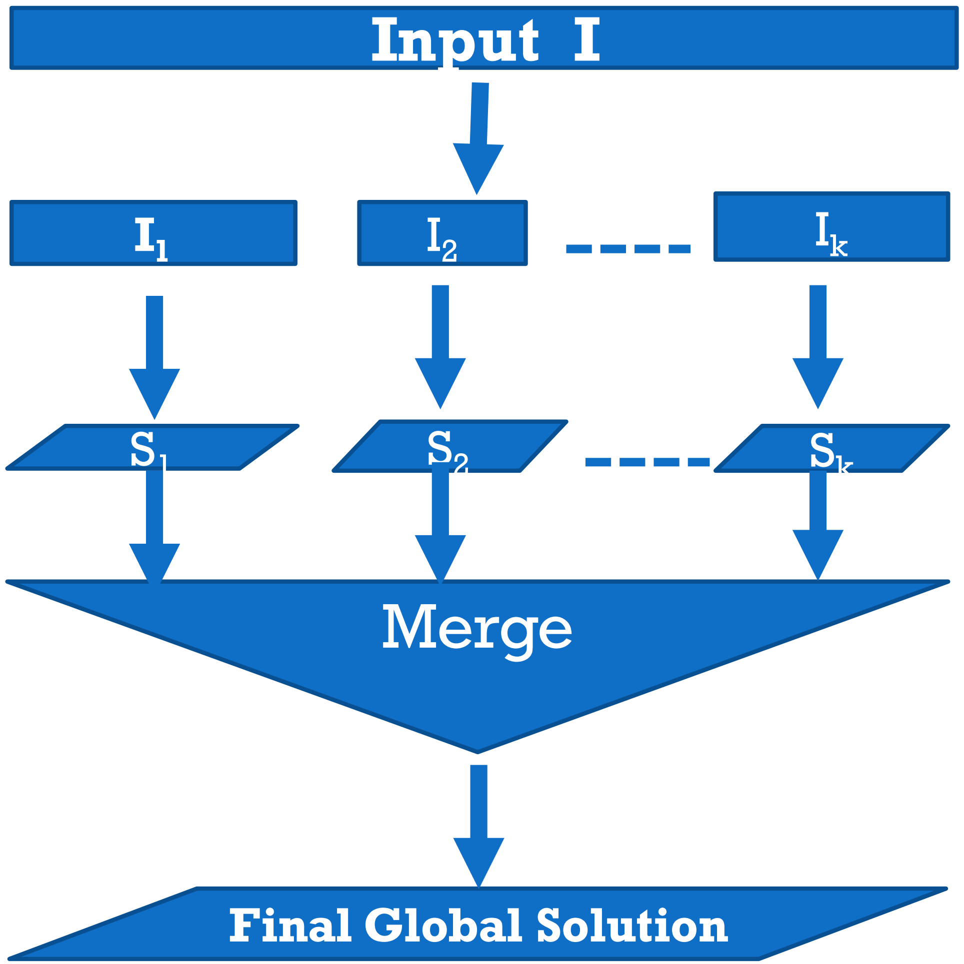
  $S_2 \leftarrow$ divide&conquer(I_2) ;

  ………………

  Merge the subsolutions S_1, S_2,…into a

    global solution S;

end



FIRST APPLICATION
-- SORTING --

- The Sorting problem:
  - **Input**: An arbitrary array of numbers
    - or array of any data type for which we have a comparator like $\leq$
  - **Output**: the same input but in increasing order (from min to max)

- **Goal**: Apply Divide & Conquer to design an algorithm for sorting

- Note: we can sort into decreasing order (from max to min)
  - simply change $\leq$ to $\geq$

FIRST APPLICATION
-- SORTING REMARKS --

- Sorting is one of the oldest problems in CS
- Sorting algorithms are among the most widely used in IT
- Many sorting algorithms have been developed
- "First-generation" sorting algorithms take $O(n^2)$ time, which is relatively slow, especially for large n
- Some 1st gen sorting algs: *insertion sort*, *selection sort*, *exchange sort*
- Divide & Conquer sorting algorithms are much faster, as will be seen in this lecture

FIRST APPLICATION
-- MERGESORT--

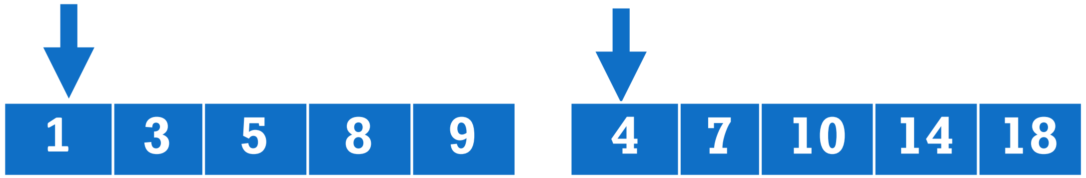
```
Proc. Mergesort (in A[1:n],i,j; out B[1:n])
// sorts A[i:j] to B[i:j]
begin
      generic C[1:n]; // same type as A
      if i==j then B[i] = A[i]; Return; endif
      Mergesort (A,i,(i+j)/2;C); // sorts 1st half
      Mergesort(A,(i+j)/2 +1,j;C); // sorts 2nd half
      Merge(C,i,j;B);  // merges the two sorted
                  // halves into a single sorted array
end Mergesort

To sort whole array: call Mergesort (A,1,n;B)
```

```
Procedure Merge(in C i,j; out B)
// merges C[i:k] and C[k+1:j] into B[i:j]
// k=(i+j)/2
begin
      int k=(i+j)/2, u=i, v=k+1, w=u;
      // u scans C[i:k], v scans C[k+1:j]
      // w indexes B
      while  (u <= k and v <= j) do
            if C[u] <= C[v] then B[w++]=C[u++];
            else   B[w++]=C[v++];
            endif
      endwhile
      if u > k then   B[w:j] = C[v:j];
      elseif v>j then  B[w:j] = C[u:k];
      endif
end Merge
```

EXPLANATION OF MERGE

- Input: Two sorted arrays



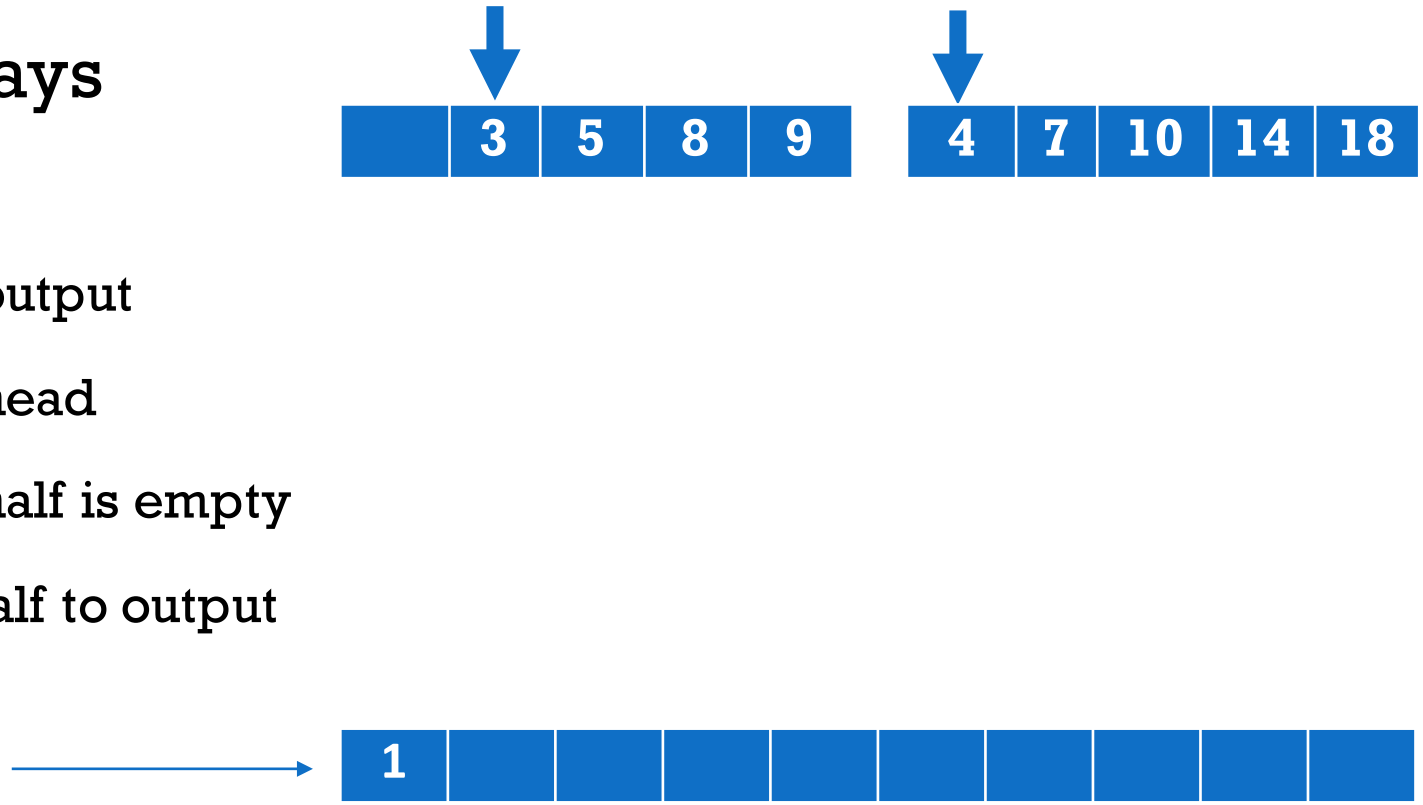
1. Compare heads
2. Move smaller value to the output
3. Move forward the smaller head
4. Repeat 1-3 until one input half is empty
5. Move remainder of other half to output

EXPLANATION OF MERGE

- Input: Two sorted arrays

1. Compare heads
2. Move smaller value to the output
3. Move forward the smaller head
4. Repeat 1-3 until one input half is empty
5. Move remainder of other half to output

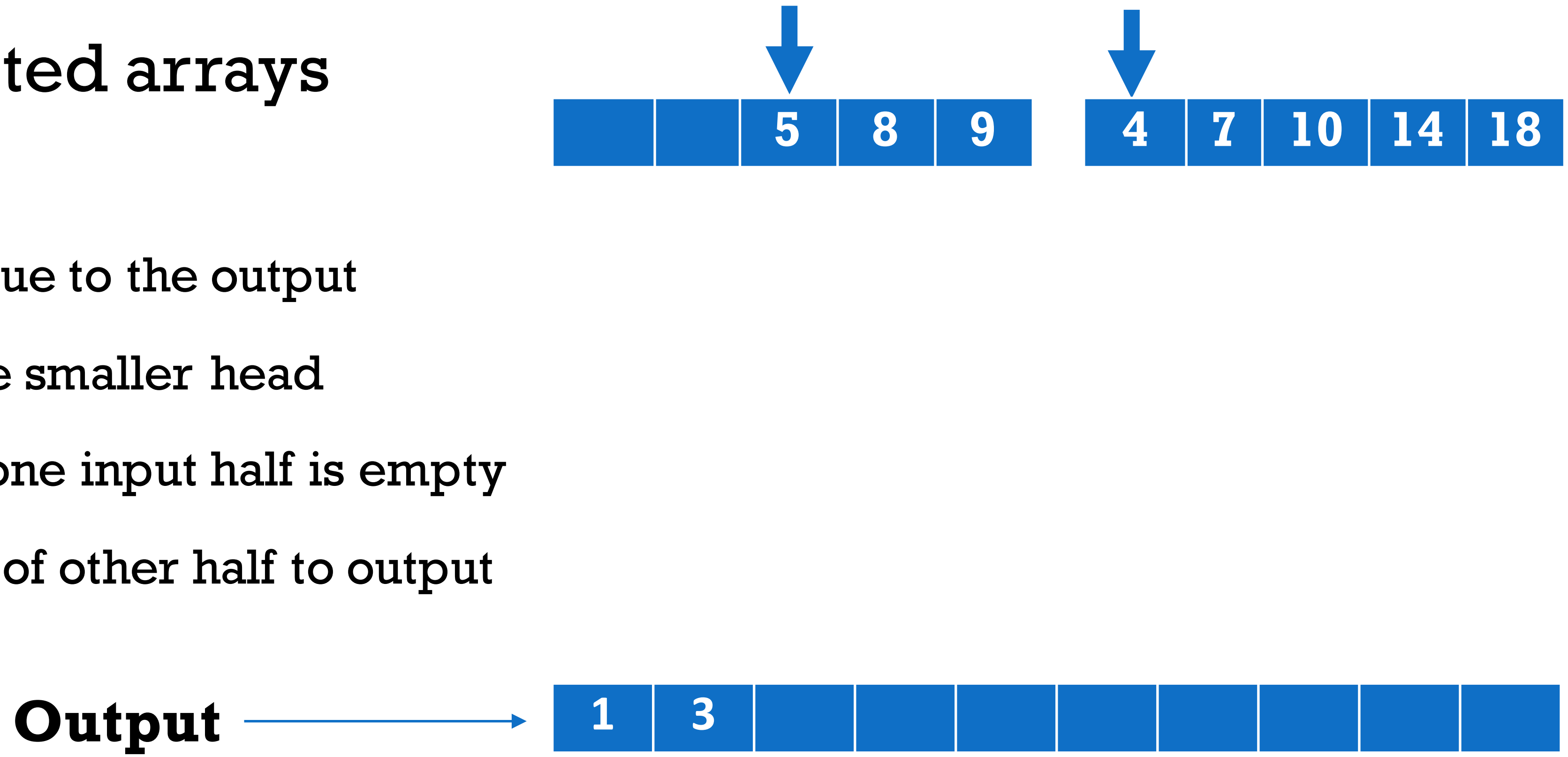
Output



EXPLANATION OF MERGE

- Input: Two sorted arrays

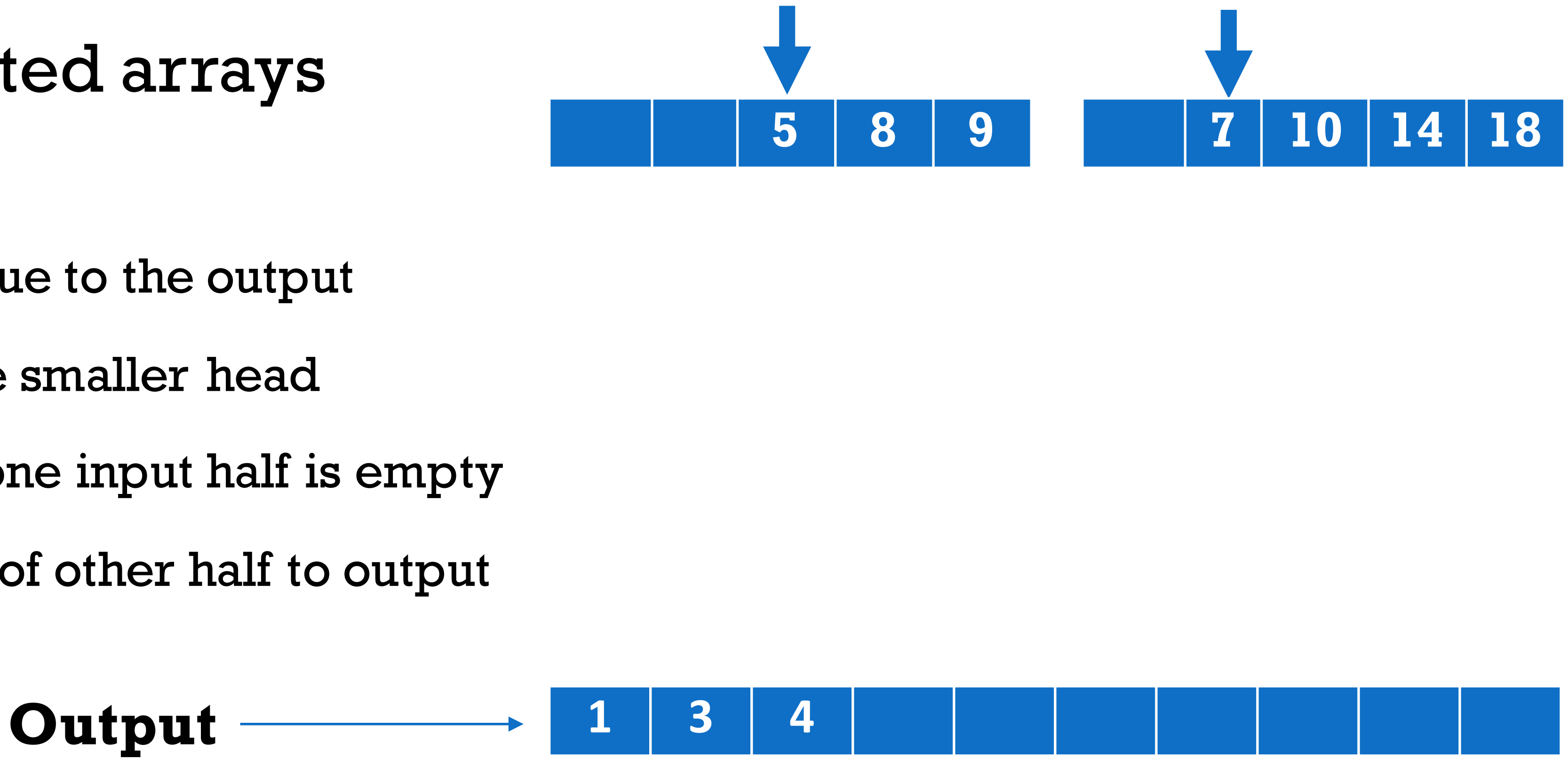
1. Compare heads
2. Move smaller value to the output
3. Move forward the smaller head
4. Repeat 1-3 until one input half is empty
5. Move remainder of other half to output



EXPLANATION OF MERGE

- Input: Two sorted arrays

1. Compare heads
2. Move smaller value to the output
3. Move forward the smaller head
4. Repeat 1-3 until one input half is empty
5. Move remainder of other half to output



EXPLANATION OF MERGE

- Input: Two sorted arrays

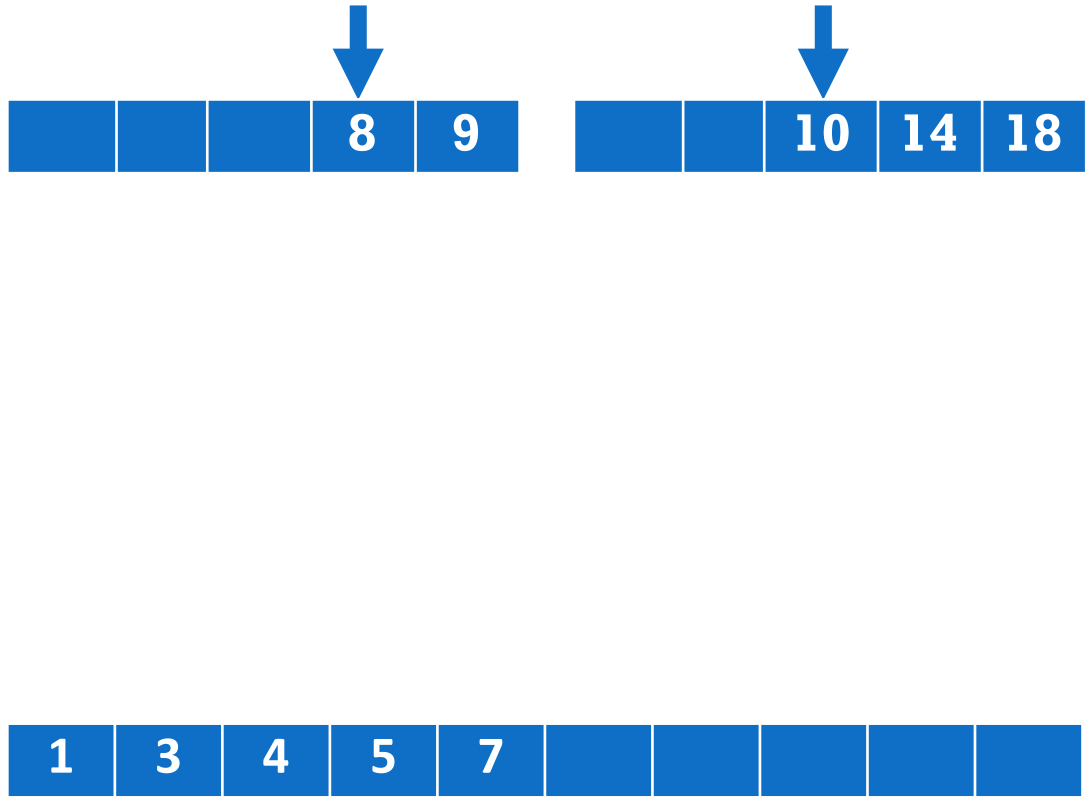
1. Compare heads
2. Move smaller value to the output
3. Move forward the smaller head
4. Repeat 1-3 until one input half is empty
5. Move remainder of other half to output

EXPLANATION OF MERGE

- Input: Two sorted arrays

1. Compare heads
2. Move smaller value to the output
3. Move forward the smaller head
4. Repeat 1-3 until one input half is empty
5. Move remainder of other half to output

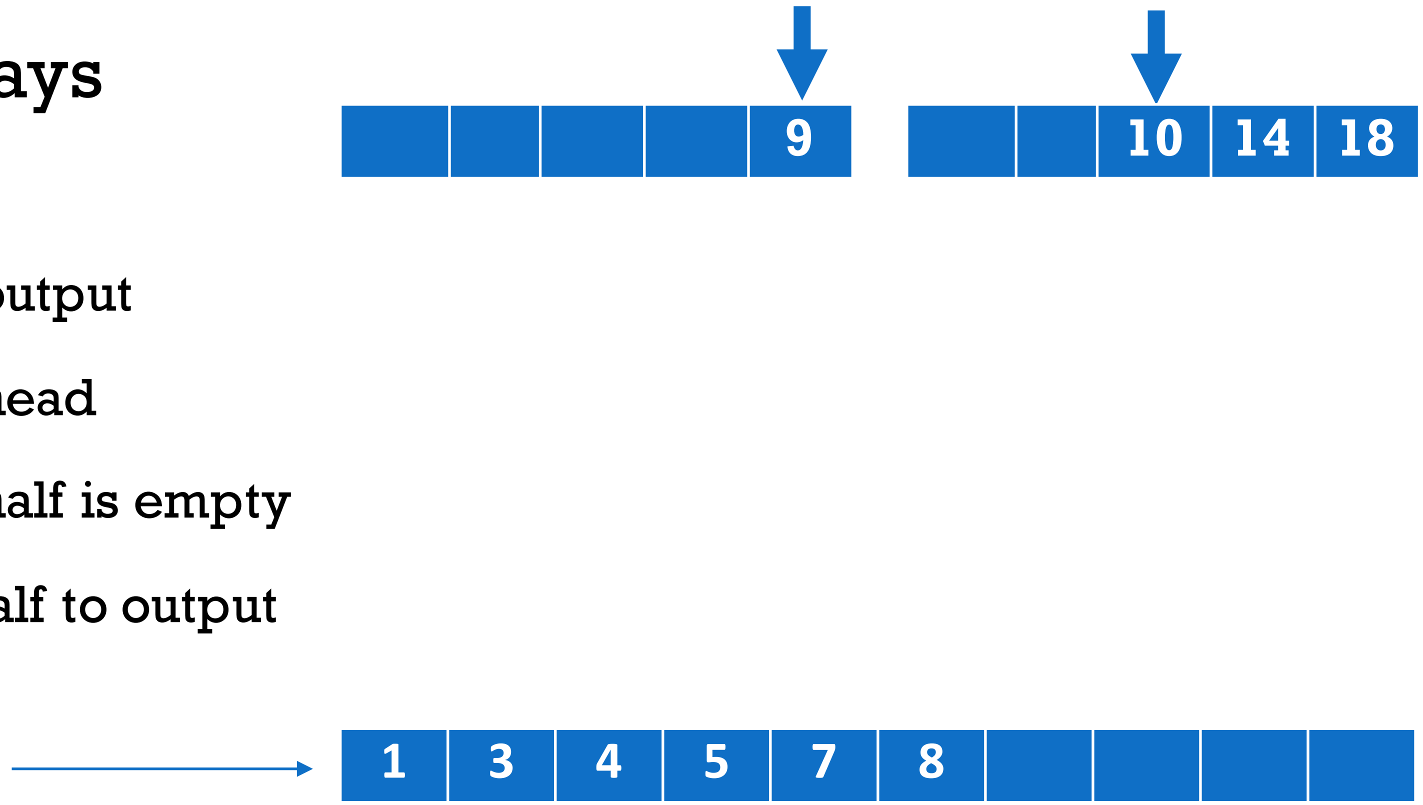
Output →



EXPLANATION OF MERGE

- Input: Two sorted arrays

1. Compare heads
2. Move smaller value to the output
3. Move forward the smaller head
4. Repeat 1-3 until one input half is empty
5. Move remainder of other half to output

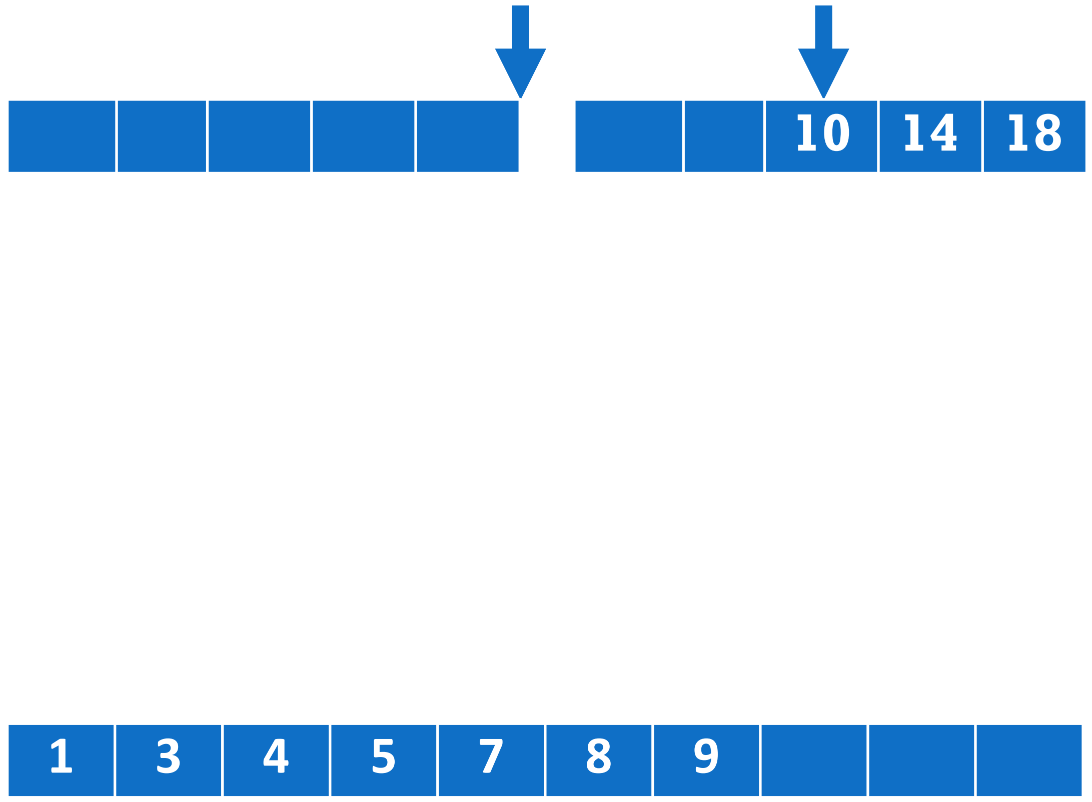


Output

EXPLANATION OF MERGE

- Input: Two sorted arrays

1. Compare heads
2. Move smaller value to the output
3. Move forward the smaller head
4. Repeat 1-3 until one input half is empty
5. Move remainder of other half to output



Output

EXPLANATION OF MERGE

- Input: Two sorted arrays

1. Compare heads
2. Move smaller value to the output
3. Move forward the smaller head
4. Repeat 1-3 until one input half is empty
5. Move remainder of other half to output

Output →

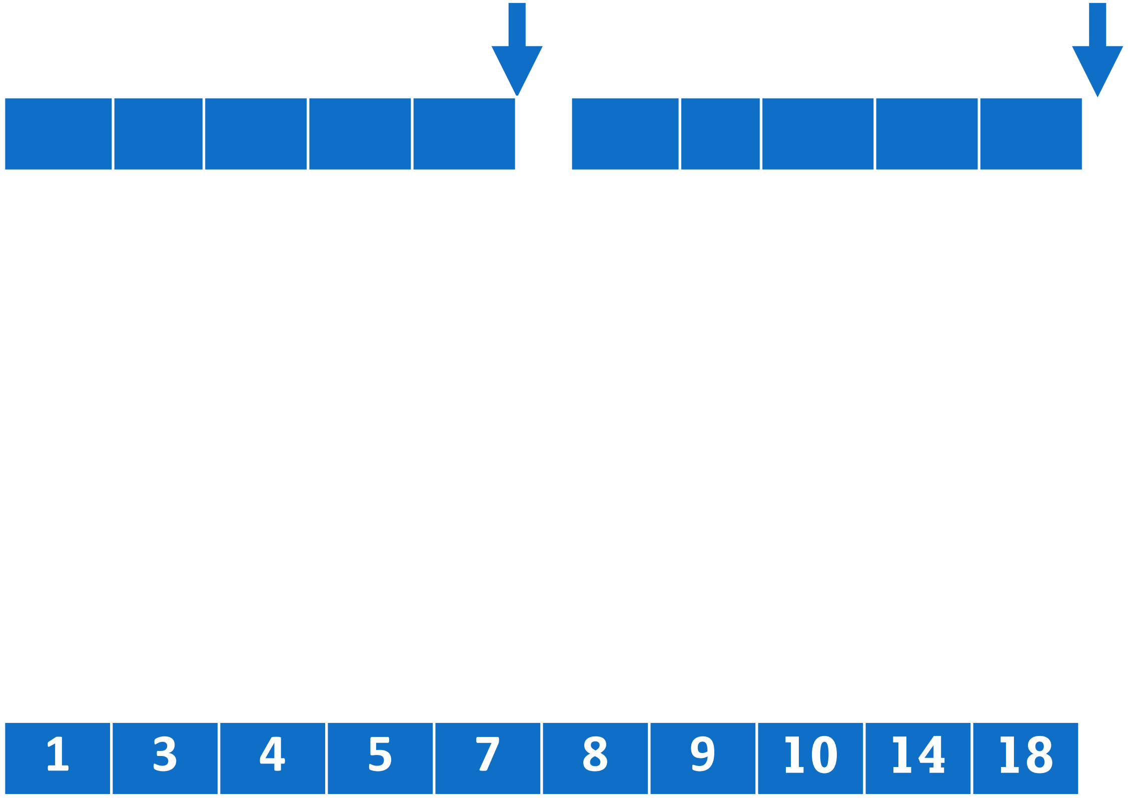
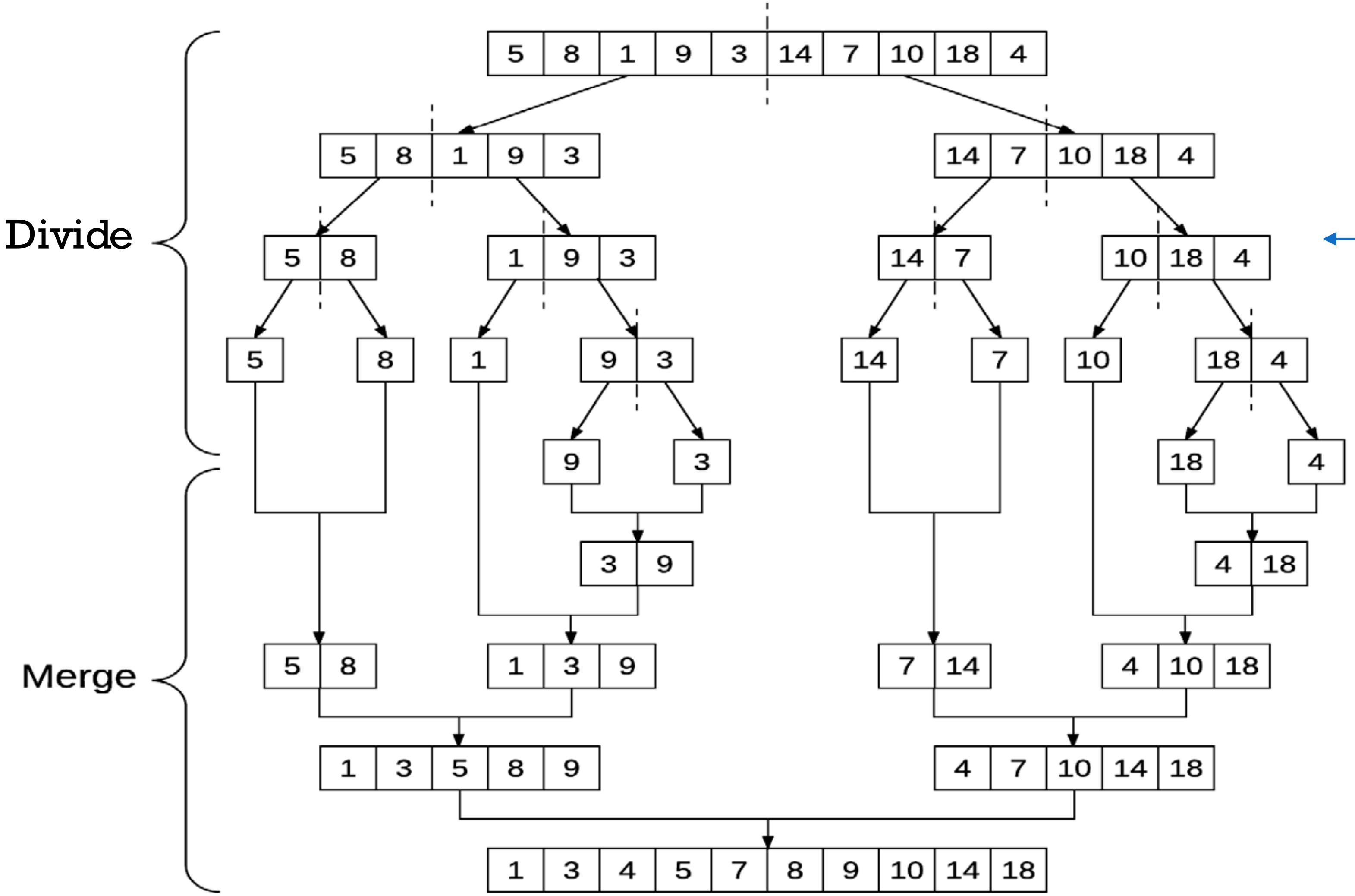


ILLUSTRATION OF MERGESORT
-- WHAT GOES ON INSIDE THE COMPUTER --



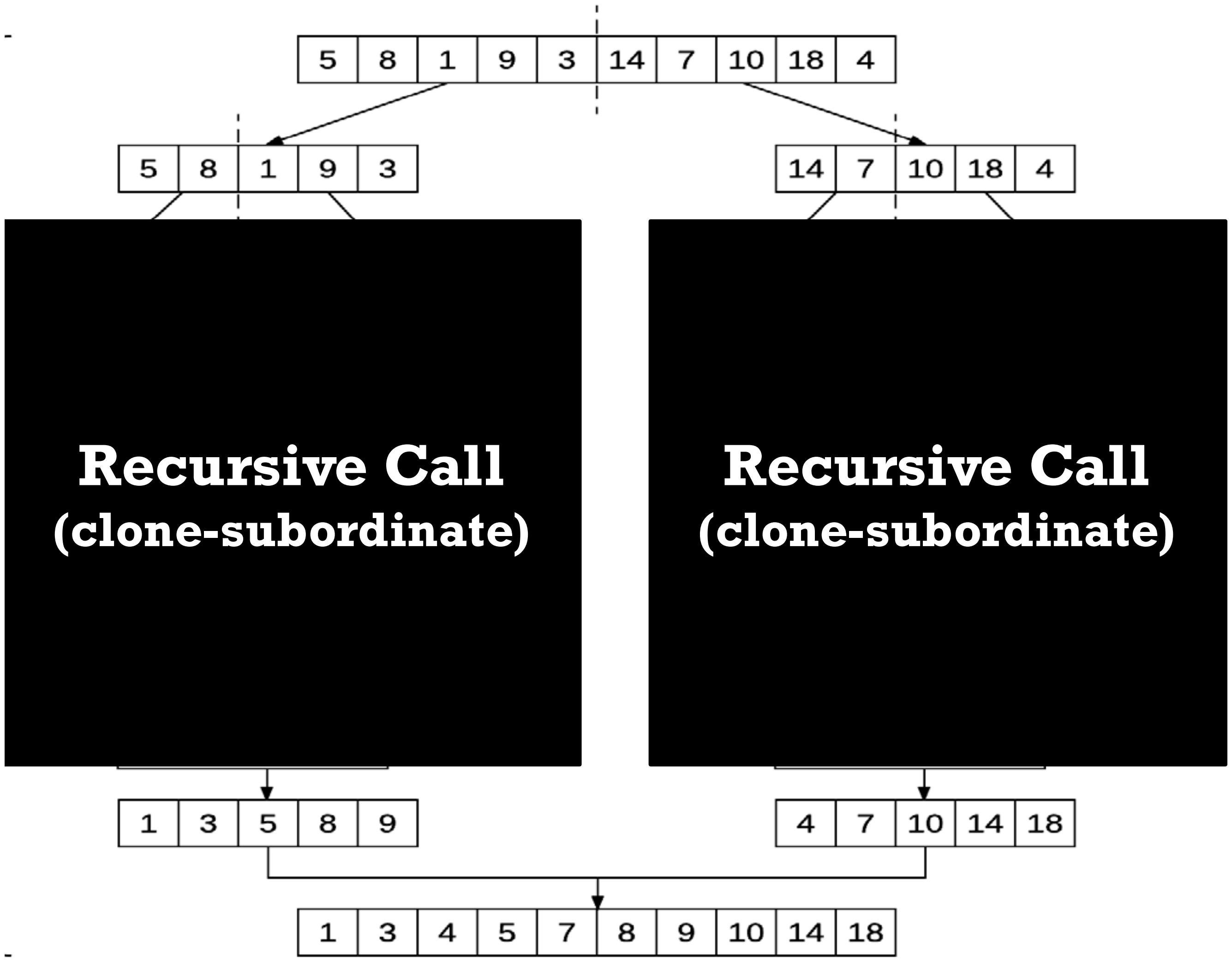
- This is what goes on inside the computer when executing Metgesort
- But, **<u>don't do</u>** that "at home"
- Rather, … (see next slide)

THE “BOSS-VIEW” OF MERGESORT

Proper Mindset:

1. Divide data into parts
2. Then, as a boss, hand each part to a clone-subordinate
3. Wait for each subordinate to come back with its sub-solution
4. Then, as the boss, you take the sub-solutions and merge into a global solution
5. As as boss, you take the credit!

- NEVER MICROMANAGE your subordinates



TIME COMPLEXITY OF MERGESORT
-- DERIVING A RECURRENCE RELATION --

- Time of Merge: O(n)=cn, for some constant c, because:
  - After each comparison, the input loses one element
  - Once the input loses all its elements (after $\leq$ n comparisons), it is done
- Time of Mergesort:
  - Let $T(n)$ be the time of Mergesort of n elements
  - $T(n)$ = (time of each Mergesort on n/2 elements)+(time of Merge)
  - $T(n) = 2T\left(\frac{n}{2}\right) + cn,$ T(1)=constant=c
  - The above is called a recurrence relation

TIME COMPLEXITY OF MERGESORT
-- SOLVING THE RECURRENCE RELATION --

- Can be solved with the Master Theorem
- But we will solve it here more informally/easily

- $T(n) = 2T\left(\frac{n}{2}\right) + cn$, T(1)=constant=c. **Assume $n = 2^k$**

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$
$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + c\frac{n}{2}$$
$$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}$$
$$\dots$$
$$T\left(\frac{n}{2^{k-1}}\right) = 2T\left(\frac{n}{2^k}\right) + c\frac{n}{2^{k-1}}$$

- Each line above came from applying the recurrence relation on some $\frac{n}{2^i}$, for $i = 0, 1, 2, \dots, k-1$
- Multiply each i^{th} line above by 2^i

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$
$$2T\left(\frac{n}{2}\right) = 2^2T\left(\frac{n}{2^2}\right) + 2c\frac{n}{2}$$
$$2^2T\left(\frac{n}{2^2}\right) = 2^3T\left(\frac{n}{2^3}\right) + 2^2c\frac{n}{2^2}$$
$$\dots$$
$$2^{k-1}T\left(\frac{n}{2^{k-1}}\right) = 2^kT\left(\frac{n}{2^k}\right) + 2^{k-1}c\frac{n}{2^{k-1}}$$

TIME COMPLEXITY OF MERGESORT
-- SOLVING THE RECURRENCE RELATION (2) --

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$
$$2T\left(\frac{n}{2}\right) = 2^2T\left(\frac{n}{2^2}\right) + 2c\frac{n}{2}$$
$$2^2T\left(\frac{n}{2^2}\right) = 2^3T\left(\frac{n}{2^3}\right) + 2^2c\frac{n}{2^2}$$
$$\ldots$$
$$2^{k-1}T\left(\frac{n}{2^{k-1}}\right) = 2^kT\left(\frac{n}{2^k}\right) + 2^{k-1}c\frac{n}{2^{k-1}}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$
$$2T\left(\frac{n}{2}\right) = 2^2T\left(\frac{n}{2^2}\right) + cn$$
$$2^2T\left(\frac{n}{2^2}\right) = 2^3T\left(\frac{n}{2^3}\right) + cn$$
$$\ldots$$
$$2^{k-1}T\left(\frac{n}{2^{k-1}}\right) = 2^kT\left(\frac{n}{2^k}\right) + cn$$

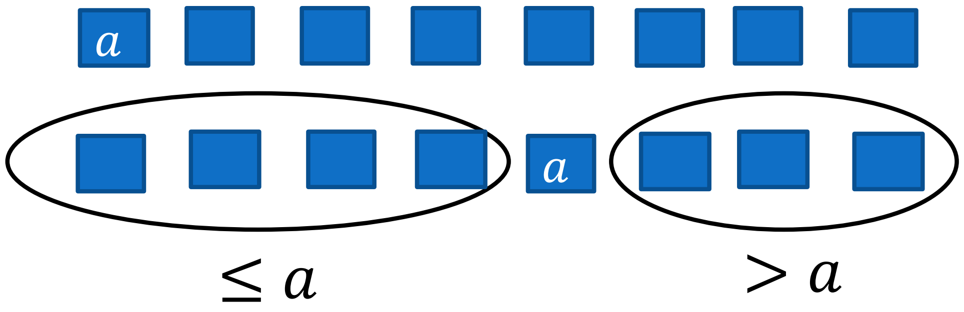
- Sum of left terms = sum of right terms
- Cancel terms that occur on both sides of "="
- What remains on the left is: $T(n)$
- What remains on the right: $2^kT\left(\frac{n}{2^k}\right) + cnk = nT(1) + cnk$
- Therefore: $T(n) = nT(1) + cnk = cn + cn\log n = O(n\log n)$
- $\boldsymbol{T(n) = O(n\log n)}$

SECOND APPLICATION OF D&C
-- QUICKSORT --

- This time we partition the input A[1:n] around an element in input A[1:n], say $a = A[1]$, such that, after the partitioning:
  - All the input elements that are $\leq a$ are put in the first (left) partition
  - All the input elements that are $> a$ are put in the second (right) partition

Input A:

After partitioning around a:



- Partitioning takes O(n) time

SECOND APPLICATION OF D&C
-- QUICKSORT ALGORITHM --

Procedure Quicksort(in/out A[1,n];in: p,q)   // sorts A[p:q]
// The sorting is *in situ*, i.e., in place (within the same input array A
begin
    int r;
    if (p==q) **then return**; **endif** // if one element to sort, then sorted
    r := partition(A[p:q]);   // r is the index where the partitioning
                                          // element (p.a.) lands, i.e., now A[r]==p.a.
    Quicksort(A[p:r-1]);   // now A[p:r-1] is sorted, and all are $\le a$
    Quicksort(A[r+1:q]);   // now A[r+1:q] is sorted, and all are $> a$
end

At the end of the algorithm, A[p:q] is sorted, because:

- A[p:r-1] is sorted and all are $\le a$ => $A[p] \le A[p+1] \le \ldots \le A[r-1] \le a = A[r]$
- A[r+1:q] is sorted, and all are $> a$ => $a < A[r+1] \le A[r+2] \le \ldots \le A[q]$
- Therefore: $A[p] \le A[p+1] \le \ldots \le A[r-1] \le a = A[r] < A[r+1] \le A[r+2] \le \ldots \le A[q]$

TIME COMPLEXITY OF QUICKSORT

- Let $T(n)$ be the time of Quicksort(A[1,n];1,n)
- $T(n)$= (time of partition)+(time of Quicksort(A[1:n];1,r-1])) +

  (time of Quicksort(A[1:n];r+1,n]))
- $T(n) = cn + T(r-1) + T(n-r)$
- This is a recurrence relation, but we don't know r
- Worst-case time complexity:
  - $r = 1$ (i.e., partitioning is extremely unbalanced)
  - $T(n) = cn + T(1-1) + T(n-1) = cn + T(0) + T(n-1)$
  - $T(n) = T(n-1) + cn$

TIME COMPLEXITY OF QUICKSORT
-- WORST-CASE ANALYSIS --

- Cannot be solved with the Master Theorem b/c the latter doesn't apply to this kind of recurrence relation
- We'll solve it using the informal unfolding method

- $T(n) = T(n-1) + cn$

$$
\begin{aligned}
T(n) &= T(n-1) + cn \\
T(n-1) &= T(n-2) + c(n-1) \\
T(n-2) &= T(n-3) + c(n-2) \\
&\ldots\ldots \\
T(1) &= T(0) + c.1 = c.1
\end{aligned}
$$

The lines in the left box are all derived by applying the top recurrence relations at different values: $T(m) = T(m-1) + cm$ for $m = n, n-1, n-2, \ldots, 1$.

- Sum of left terms = sum of right terms
- Cancel terms that occur on both sides of "="
- What remains on the left is: $T(n)$
- What remains on the right: $c(1 + 2 + \cdots + (n-1) + n)$
- Therefore: $T(n) = c(1 + 2 + \cdots + (n-1) + n) = cn(n+1)/2$
- Conclusion: $\boldsymbol{T(n) = O(n^2)}$, which is bad!

TIME COMPLEXITY OF QUICKSORT
-- AVERAGE-CASE ANALYSIS --

- **Irony**: Quicksort is slow in the worst case ($O(n^2)$) yet it is called **Quick**sort
- **Reality**: In practice, Quicksort is the fastest sorting algorithm around, faster even than Mergesort (which takes O(n log n) time < $O(n^2)$)
- So, what is going on?
- Well, the worst case occurs when the input happens to be already sorted (or nearly sorted), but that rarely happens
- In practice, the input is in random order
  - So, the question is: What happens if we have **average input**
- We need to perform "average-case" time complexity analysis

AVERAGE-CASE ANALYSIS OF QUICKSORT (1)

- Recall the general recurrence relation:

$$T(n) = T(r-1) + T(n-r) + cn$$

where r can be 1 or 2 or … or n

- Thus, $T(n)$ can be:
  - $T(n) = T(0) + T(n-1) + cn$, or
  - $T(n) = T(1) + T(n-2) + cn$, or
  - $T(n) = T(2) + T(n-3) + cn$, or
  - ………
  - $T(n) = T(n-1) + T(0) + cn$

- So, the average value of T(n) is the average of those n possible values, i.e., (the sum of those values)/n
- As you sum, group the terms as shown left
- Thus, the sum is: $2[T(0) + T(1) + T(2) + \cdots T(n-1)] + cn.n$

AVERAGE-CASE ANALYSIS OF QUICKSORT (2)

- Therefore, the average of $T(n)$, denoted $T_A(n)$, is:
  - $T_A(n) = \mathrm{sum}/n$
  - $T_A(n) = \left[2\big(T(1) + T(2) + \cdots + T(n-1)\big) + cn.n\right]/n$
  - $T_A(n) = \left[2\big(T(1) + T(2) + \cdots + T(n-1)\big) + cn^2\right]/n$
- Multiplying both sides by n, we get
  - $nT_A(n) = 2\big(T(1) + T(2) + \cdots + T(n-1)\big) + cn^2$

AVERAGE-CASE ANALYSIS OF QUICKSORT (3)

- $nT_A(n) = 2\big(T(1) + T(2) + \cdots + T(n-1)\big) + cn^2$
- Since we are considering average time, we can assume that each $T(i)$ on the right (which is a recursive call on an average part) to be an average time $T_A(i)$
  - $nT_A(n) = 2\big(T_A(1) + T_A(2) + \cdots + T_A(n-1)\big) + cn^2$
- Applying the formula above at $n-1$, we get
  - $(n-1)T_A(n-1) = 2\big(T_A(1) + T_A(2) + \cdots + T_A(n-2)\big) + c(n-1)^2$
- Subtracting the last two equations, we obtain:
  - $nT_A(n) - (n-1)T_A(n-1) = 2T_A(n-1) + cn^2 - c(n-1)^2$
- Performing some arithmetic, we get:
  - $nT_A(n) = (n-1)T_A(n-1) + 2T_A(n-1) + 2cn - c$
  - $nT_A(n) = (n+1)T_A(n-1) + 2cn - c$
  - $nT_A(n) \leq (n+1)T_A(n-1) + 2cn$ (because we got rid of $-c$)

AVERAGE-CASE ANALYSIS OF QUICKSORT (4)

- $nT_A(n) \le (n+1)T_A(n-1) + 2cn$
- Divide both sides by $n(n+1)$, we get:
  - $\frac{nT_A(n)}{n(n+1)} \le \frac{(n+1)T_A(n-1)}{n(n+1)} + \frac{2cn}{n(n+1)}$
  - $\frac{T_A(n)}{n+1} \le \frac{T_A(n-1)}{n} + \frac{2c}{n+1}$
- Calling $\boldsymbol{f(n) = \frac{T_A(n)}{n+1}}$, and thus $f(n-1) = \frac{T_A(n-1)}{n}$, the above equation becomes:
  - $f(n) \le f(n-1) + \frac{2c}{n+1}$

AVERAGE-CASE ANALYSIS OF QUICKSORT (5)

- $f(n) \le f(n-1) + \frac{2c}{n+1}$, where $f(n) = \frac{T_A(n)}{n+1}$ $\quad (f(0) = \frac{T_A(0)}{0+1} = 0)$

$$
\begin{aligned}
f(n) &\le f(n-1) + \frac{2c}{n+1} \\
f(n-1) &\le f(n-2) + \frac{2c}{n} \\
f(n-2) &\le f(n-3) + \frac{2c}{n-1} \\
&\cdots\cdots\cdots\cdots \\
f(1) &\le f(0) + \frac{2c}{2}
\end{aligned}
$$

The lines in the left box are all derived by applying the top recurrence relations at different values: $f(m) \le f(m-1) + \frac{2c}{m+1}$ for $m = n, n-1, n-2, \ldots, 1$.

- Sum of left terms $\le$ sum of right terms
- Cancel terms that occur on both sides of "$\le$"
- What remains on the left is: $f(n)$
- What remains on the right: $f(0) + 2c(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} + \frac{1}{n+1})$
- Therefore: $f(n) \le 2c(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} + \frac{1}{n+1})$ $\quad$ note: $f(0)=0$

AVERAGE-CASE ANALYSIS OF QUICKSORT (6)

- $f(n) \le 2c(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} + \frac{1}{n+1})$, where $f(n) = \frac{T_A(n)}{n+1}$
- From Calculus, $\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} + \frac{1}{n+1} \le \text{Ln}\,(n+1) = O(\log n)$
- Therefore, $f(n) \le 2c\ \text{Ln}\,(n+1)$
- Since, $f(n) = \frac{T_A(n)}{n+1}$ and hence $T_A(n) = (n+1)f(n)$, we get
  - $T_A(n) \le (n+1)f(n) \le 2c(n+1)\text{Ln}\,(n+1) = O(n \log n)$
- Conclusion: $\boldsymbol{T_A(n) = O(n\, log\, n)}$
- Because the constant factor in the above big-O is < the constant factors of the Big-O of other sorting algorithms, Quicksort is faster on average than other sorting algorithms

THE PARTITION ALGORITHM

- Quicksort did some fancy partitioning
- Now we give an O(n) time *in situ* partition algorithm

THE PARTITION ALGORITHM (2)

```
Function Partition(in/out  A[p:q])
begin
    int i,j;
    real a=A[p];            // a is the partitioning element
    i=p; j=q;
    while (i < j)  do
        while (A[i] <= a && i<q) do i++;  endwhile
        while (A[j] > a && j>p)  do j--; endwhile
        if i < j then
            swap (A[i],A[j]);   i++;   j--;
        endif
    endwhile
    swap(A[p],A[j]);
    return (j);
end
```

ILLUSTRATION OF PARTITION

Partitioning Element	Operation	Array									
A[1]=5	Partition(A[1:10])	5	8	1	9	3	14	7	10	18	4
		5	8	1	9	3	14	7	10	18	4
		i									j

ILLUSTRATION OF PARTITION

Partitioning Element	Operation	Array									
A[1]=5	Partition(A[1:10])	5	8	1	9	3	14	7	10	18	4
	8>5, 4<5	**5**	8	1	9	3	14	7	10	18	4
			↑ i								↑ j

ILLUSTRATION OF PARTITION

Partitioning Element	Operation	Array									
A[1]=5	Partition(A[1:10])	5	8	1	9	3	14	7	10	18	4
	8>5, 4<5 Swap(A[2],A[10])	**5**	8	1	9	3	14	7	10	18	4
			↑ i								↑ j
		5	4	1	9	3	14	7	10	18	8
				↑ i						↑ j	

ILLUSTRATION OF PARTITION

Partitioning Element	Operation	Array									
A[1]=5	Partition(A[1:10])	5	8	1	9	3	14	7	10	18	4
	8>5, 4<5 Swap(A[2],A[10])	**5**	8	1	9	3	14	7	10	18	4
			↑ i								↑ j
		5	4	1	9	3	14	7	10	18	8
					↑ i					↑ j	

ILLUSTRATION OF PARTITION

Partitioning Element	Operation	Array									
A[1]=5	Partition(A[1:10])	5	8	1	9	3	14	7	10	18	4
	8>5, 4<5 Swap(A[2],A[10])	5	8	1	9	3	14	7	10	18	4
			↑ i								↑ j
		5	4	1	9	3	14	7	10	18	8
					↑ i				↑ j		

ILLUSTRATION OF PARTITION

Partitioning Element	Operation	Array									
A[1]=5	Partition(A[1:10])	5	8	1	9	3	14	7	10	18	4
	8>5, 4<5 Swap(A[2],A[10])	5	8	1	9	3	14	7	10	18	4
			↑ i								↑ j
		5	4	1	9	3	14	7	10	18	8
					↑ i			↑ j			

ILLUSTRATION OF PARTITION

Partitioning Element	Operation	Array									
A[1]=5	Partition(A[1:10])	5	8	1	9	3	14	7	10	18	4
	8>5, 4<5 Swap(A[2],A[10])	5	8	1	9	3	14	7	10	18	4
			↑ i								↑ j
		5	4	1	9	3	14	7	10	18	8
					↑ i		↑ j				

ILLUSTRATION OF PARTITION

Partitioning Element	Operation	Array									
A[1]=5	Partition(A[1:10])	5	8	1	9	3	14	7	10	18	4
	8>5, 4<5 Swap(A[2],A[10])	**5**	8	1	9	3	14	7	10	18	4
			↑ i								↑ j
		5	4	1	9	3	14	7	10	18	8
					↑ i	↑ j					

ILLUSTRATION OF PARTITION

Partitioning Element	Operation	Array									
A[1]=5	Partition(A[1:10])	5	8	1	9	3	14	7	10	18	4
	8>5, 4<5 Swap(A[2],A[10])	5	8	1	9	3	14	7	10	18	4
			i								j
	9>5, 3<5 Swap(A[4],A[5])	5	4	1	9	3	14	7	10	18	8
					i	j					
		5	4	1	3	9	14	7	10	18	8
					j	i					

ILLUSTRATION OF PARTITION

Partitioning Element	Operation	Array									
A[1]=5	Partition(A[1:10])	5	8	1	9	3	14	7	10	18	4
	8>5, 4<5 Swap(A[2],A[10])	5	8	1	9	3	14	7	10	18	4
			↑ i								↑ j
	9>5, 3<5 Swap(A[4],A[5])	5	4	1	9	3	14	7	10	18	8
					↑ i	↑ j					
	i>j Swap(A[1],A[4])	5	4	1	3	9	14	7	10	18	8
					↑ j	↑ i					

ILLUSTRATION OF PARTITION

Partitioning Element	Operation	Array									
A[1]=5	Partition(A[1:10])	5	8	1	9	3	14	7	10	18	4
	8>5, 4<5 Swap(A[2],A[10])	5	8	1	9	3	14	7	10	18	4
			i								j
	9>5, 3<5 Swap(A[4],A[5])	5	4	1	9	3	14	7	10	18	8
					i	j					
	i>j Swap(A[1],A[4])	5	4	1	3	9	14	7	10	18	8
					j	i					
	Now A is partitioned	3	4	1	5	9	14	7	10	18	8

TIME COMPLEXITY OF PARTITION

- At every step, either i moves one step right or j moves one step left
- After i and j meet and cross by at most one step, only constant-time work is done and the algorithm terminates
- So the time is proportional to the "total distance" traveled by i and j combined
  - That traveled distance is the length of the array (no matter where i and j meet)
- Therefore, the time of partition is O(n)

NEXT LECTURE

- We finish Divide and Conquer
- We apply it to the Order Statistics problem:
  - Finding the k^{th} smallest element of an arbitrary (unsorted) array
- We will see a simple way of applying D&C to that problem, yielding a slow algorithm
- Then we apply D&C to that problem in a more sophisticated way, yielding a much faster algorithm