



# Recitation 03

# Problem 1

# My DNA



Question 1

# Problem

**Input:** String (sequence of characters) e.g. DNA sequence

**Output:** Ordered string

**Constraint:** Can only reverse a subsequence

## Example

**Input :** ATTTTAAAA

**Output :** AAAAATTTT

# Example

C	A	G	T	G	A	C	A	A	T
0	1	2	3	4	5	6	7	8	9

After reversing [2,5]

C	A	A	G	T	G	C	A	A	T
0	1	5	4	3	2	6	7	8	9

## Problem 1.a.

First assume your string is binary: only made up of letters **'A'** and **'T'**.

Devise a divide-and-conquer algorithm for sorting. What is the recurrence? What is the running time?

Caveat: The only legal operations are reversals. You cannot simply count the **'A'**s and **'T'**s and rebuild the string! Inspecting/examining the string is free.

# Guiding question

What are the divide and conquer algorithms we have seen so far and which one of them is appropriate?

# Guiding question

What are the divide and conquer algorithms we have seen so far and which one of them is appropriate?

**Answer:** We have seen Merge Sort and Quick Sort. Since we cannot easily implement Quick Sort with the operation constraint of subsequence flipping, Merge Sort is probably a better choice.



# Guiding question

What do we achieve by sorting a sequence of only 2 element types?

# Guiding question

What do we achieve by sorting a sequence of only 2 element types?

**Answer:** We achieve a *binary partitioning* where each partition is a sequence of homogeneous type.

# Guiding question

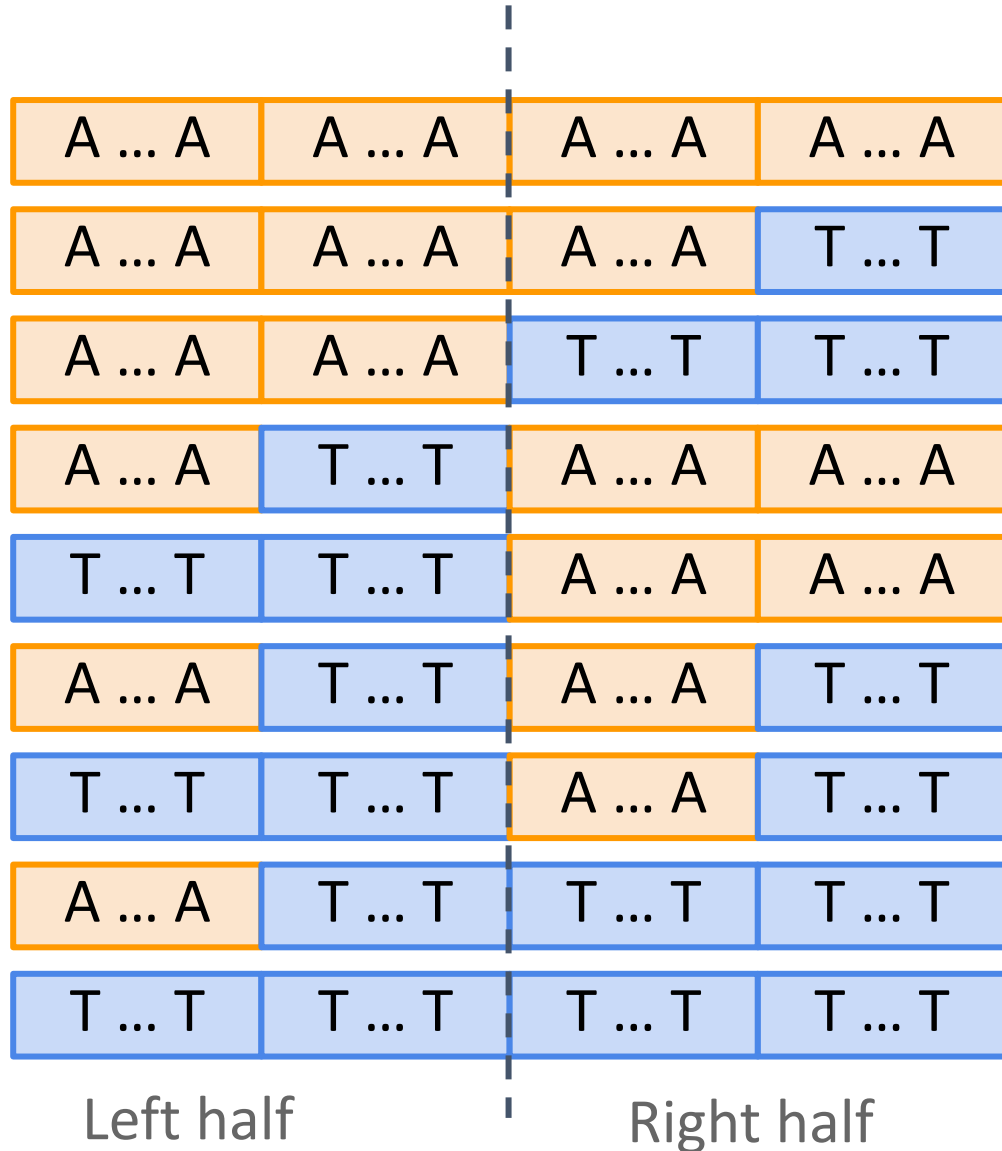
After dividing the string into two halves and sorting each half, what do you have?

# Guiding question

After dividing the string into two halves and sorting each half, what do you have?

**Answer:** After sorting, each half will entail at most 2 homogeneous partitions each. We would end up with 9 possible distinct partition sequences (see next slide). Note that although the total number of possible combinations for 4 homogenous partitions is  $2^4 = 16$ , not all these are valid since each half must be itself sorted.

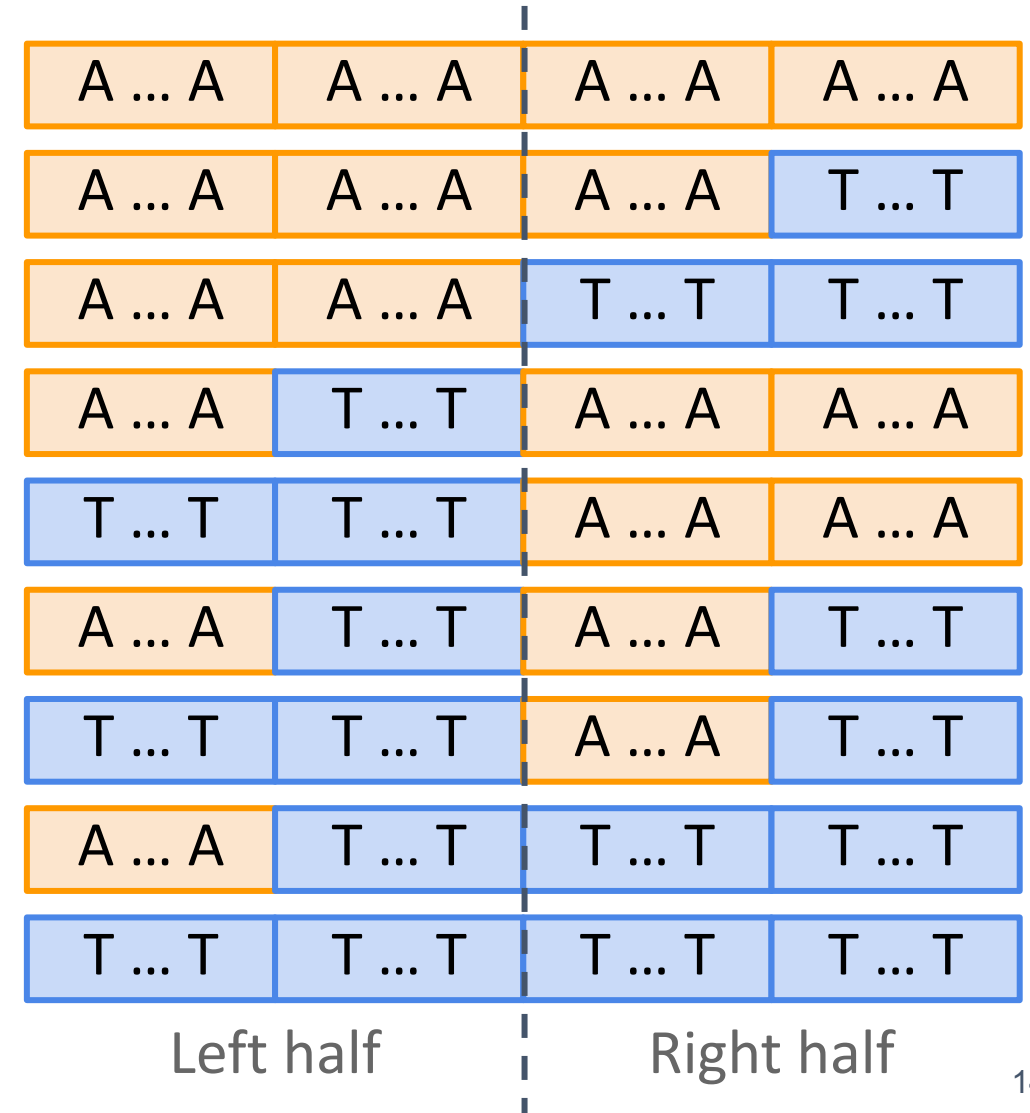
# 9 distinct sequences possible



$A^*$  refers to 0 or more number of  $A$ s.

# Guiding question

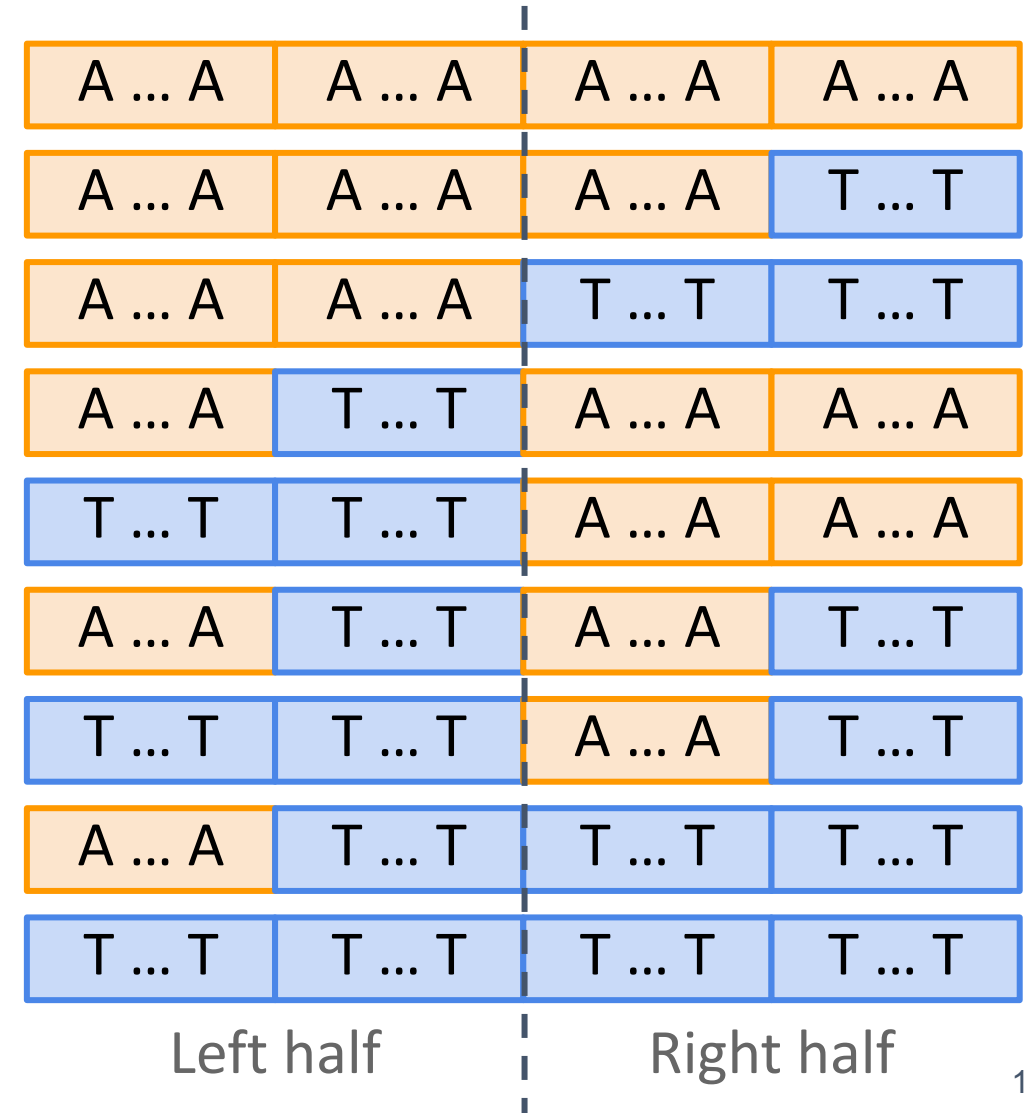
Given these cases, how do we implement the modified merge step?  
Can you come up with a generalizable algorithm to do so? What is its complexity?



# Guiding question

Given these cases, how do we implement the modified merge step? Can you come up with a generalizable algorithm to do so? What is its complexity?

**Answer:** Reverse sequence [starts at first T in left half, finishes with last A in right half].  $O(n)$  time.



# Guiding question

What is the overall time complexity of our binary partitioning algorithm using our modified Merge Sort?



## Guiding question

What is the overall time complexity of our binary partitioning algorithm using our modified Merge Sort?

**Answer:** Since we only modified the merge step and its complexity is still  $O(n)$ . The overall complexity of our modified Merge Sort is still  $O(n \log n)$ .

## Problem 1.b.

Suppose now, our input string comprise of arbitrary characters without duplicates. Your new task is to devise a QuickSort-like algorithm for sorting the string but still obeying the same rules as before:

- Only substring reversals are permitted
- Inspecting the string is free

*Hint:* Use our modified MergeSort from before to help you.

What is the recurrence? What is the running time?

# QuickSort overview

1. Pick a random pivot
2. Partition based on pivot
3. Recursive step:
  - QuickSort on left half
  - QuickSort on right half

# Guiding question

Which step in QuickSort can we implement using our 1.a. solution with minimal modifications?



Q 1.a. - This can be done in  $O(n \log n)$

# Difference in the new Problem

- More than two characters. (Or more than two numeric values)

23	32	2	15	7	13	5	42	15
----	----	---	----	---	----	---	----	----

# Guiding question

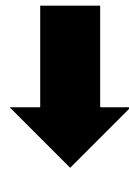
Which step in QuickSort can we implement using our 1.a. solution with minimal modifications?

**Answer:** Step 2 (partitioning step). We can transform it into a binary partitioning problem by labeling each element (using a bit) based on whether it is greater or lesser than pivot. Thereafter, we can solve as per 1.a. solution

# Example

- More than two characters. (Or more than two numeric values)

23	32	2	15	7	13	5	42	15
----	----	---	----	---	----	---	----	----



Pivot - 23

23	32	2	15	7	13	5	42	15
----	----	---	----	---	----	---	----	----

X	1	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---

Now, we can use the previous solution for this partitioning.



# Guiding question

Assuming we pick a good pivot on average which partitions into two even halves, what is the time complexity  $T(n)$  of our solution expressed as a recurrence relation?

# Guiding question

Assuming we pick a good pivot on average which partitions into two even halves, what is the time complexity  $T(n)$  of our solution expressed as a recurrence relation?

**Answer:**  $T(n) = 2T(n/2) + O(n \log n)$

## Complexity analysis: loose but intuitive

Quicksort:  $T(n) = 2T(n/2) + O(n) = O(n \log n)$

Ours:  $T(n) = 2T(n/2) + O(n \log n)$

Since there is a multiple of  $\log n$  at every level, the overall time complexity should also be a factor of Quick Sort's time complexity by a multiple of  $\log n$ , thus giving us  $O(n(\log n)^2)$ .

# Complexity analysis: Rigorous [Optional]

$$\begin{aligned}T(n) &= 2T(n/2) + O(n \log n) \\&= 2[2T(n/4) + O(\frac{n}{2} \log \frac{n}{2})] + O(n \log n) && \text{Expand } T(n/2) \\&= 4T(n/4) + O(n \log \frac{n}{2}) + O(n \log n) \\&= O(n \log n + n \log \frac{n}{2} + n \log \frac{n}{4} + n \log \frac{n}{8} + \cdots + n \log \frac{n}{n}) && \text{Expanding from pattern} \\&= O(n[\log \frac{n}{1} + \log \frac{n}{2} + \log \frac{n}{4} + \log \frac{n}{8} + \cdots + \log \frac{n}{n}]) && \text{Factorize out } n \\&= O(n[\log n - \log 1 + \log n - \log 2 + \cdots + \log n - \log n]) && \text{Applying log law} \\&= O(n[\sum_{1}^{\log n} \log n - (\log 1 + \log 2 + \log 4 + \log 8 + \cdots + \log n)]) && \text{Since sequence length is } \log n\end{aligned}$$

Continued next page..

# Complexity analysis: rigorous (cont'd)

$$= O(n[\sum_1^{\log n} \log n - (0 + 1 + 2 + 3 + \cdots + \log n)])$$

$$= O(n[\sum_1^{\log n} \log n - \sum_{i=1}^{\log n} i])$$

$$= O(n[\log^2 n - (\log n)(\log n + 1)/2])$$

Open up summations

$$= O(n[\log^2 n - (\log^2 n + \log n)/2])$$

Expand bracket

$$= O(n[\log^2 n - \frac{1}{2} \log^2 n - \frac{1}{2} \log n])$$

$$= O(n[\frac{1}{2} \log^2 n - \frac{1}{2} \log n])$$

$$= O(n[\log^2 n - \log n])$$

Constant factors ignored by big O

$$= O(n \log^2 n - n \log n)$$

Opening up bracket

$$= O(n \log^2 n)$$

Taking big O

## Problem 1.c

What if there are duplicate characters in the string?

*(Hint: think the most extreme case for duplicate elements)*

Can you still use the same routine from the previous part?

If not, what would you have to do differently now?

# Guiding question

What's the worst-case input for the standard QuickSort which uses binary partitioning? Why?

# Guiding question

What's the worst-case input for the standard QuickSort which uses binary partitioning? Why?

**Answer:** A list of identical items. Every partitioning step only reduces the problem size by exactly 1.





# Guiding question

How might we turn the worst-case input into the best-case input?

# Guiding question

How might we turn the worst-case input into the best-case input?

**Answer:** Enhanced QuickSort creates 3 partitions: (1) values less than pivot, (2) values equal to pivot, (3) values greater than pivot.



Recurse into partitions (1) and (3) as per usual.

For the list of  $n$  identical items, (1) and (3) would be empty, so we are done at the first level of enhanced QuickSort:  $O(n)$  time.

# Guiding question

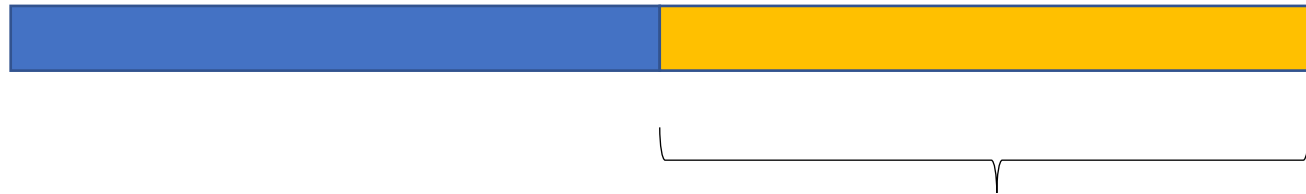
How might we create 3 partitions using our modified MergeSort which only achieves binary partitioning?

# Guiding question

How might we create 3 partitions using our modified MergeSort which only achieves binary partitioning?

**Answer:** Run it twice, with the second run on the suffix with adjusted criteria for tagging the bits :)

After first run



After second run



# Challenge yourself!

Enhanced QuickSort clearly requires extensive modifications to the procedure.

Can you come up with another solution (with minimal modifications), a “quick” enhanced QuickSort of sorts, that turns the worst-case input into an *average* case input?

# Problem 2

# Description

Given an array  $A$  of  $n$  items (they might be integers, or they might be larger objects.), we want to come up with an algorithm which produces a random permutation of  $A$  on every run.

## Problem 2.a.

Recall the problem solving process in recitation 1. Before we come up with a solution, we should be clear about what the objectives are.

What are our objectives here and what should be our metrics to evaluate how well a permutation-generation algorithm performs?



# Guiding question

What is our objective here? How do we quantify that?

# Guiding question

What is our objective here? How do we quantify that?

**Answer:** We want to generate permutations with *good randomness*. For an array with  $n$  items, we need to ensure *every* of the  $n!$  possible permutations will be producible by our algorithm with probability exactly  $1/n!$

Realize this objective entails a hard-constraint.

## Problem 2.b.

Come up with a simple permutation-generation algorithm which meets the metrics defined in the previous part.

What is the time and space complexity of your algorithm?

*Note:* It doesn't have to be an in-place algorithm.

Can you use sorting to do the shuffle?

# Discussion template

*A*

a	b	c	d	e	f	g	h	i	j
1	2	3	4	5	6	7	8	9	10

*B*

1	2	3	4	5	6	7	8	9	10

# Sorting shuffle

Step 1: Assign each element with a random number in range  $[0,1]$

<i>A</i>	a	b	c	d	e	f	g	h	i	j
	0.23	0.19	0.71	0.02	0.83	0.64	0.63	0.12	0.91	0.55

Step 2: Sort based on assigned random numbers

<i>B</i>	d	h	b	a	j	g	f	c	e	i
	0.02	0.12	0.19	0.23	0.55	0.63	0.64	0.71	0.83	0.91

## Problem 2.c.

Does the following algorithm work?

```
for ( $i$  from 1 to  $n$ ) do  
    Choose  $j = \text{random}(1, n)$   
    Swap( $A, i, j$ )  
end
```

# Running on a small example

- Let's try to run the above code over a small array " $A = \{1, 2\}$ "
- Since there are only 2 elements in the array, the number of iterations of the "for loop" in the previous code would be 2.
- Now, suppose in the first iteration (i.e.  $i = 1$ ) the random number generated, when we call " $\text{random}(1,2)$ " is 1.
- So, in the next line we will do " $\text{Swap}(A, 1, 1)$ " and thus the condition of the array after the first iteration is " $A = \{1, 2\}$ "

# Running on a small example

- Now, we come to the second iteration (i.e.  $i = 2$ ) of the “for loop”.
- Suppose in the second iteration the random number generated, when we call “random (1,2)” is 1.
- So, in the next line we do “Swap(A, 2, 1)” and thus the condition of the array after the second iteration is “A = {2, 1}”
- Our algorithm stops here and produces the array “A = {2, 1}” as output.
- This completes one run of the algorithm. The set of random numbers generated during this run is [1, 1] ( 1 in the first iteration and again 1 in the second iteration).



# Running on a small example

- Now, suppose we pass the array “ $A = \{1, 2\}$ ” as input to this algorithm again.
- But this time the random number generated in the first iteration is 1 and the random number generated in the second iteration is 2.
- So, the condition of the array  $A$  after the first iteration is “ $A = \{1, 2\}$ ” and after the second iteration is “ $A = \{1, 2\}$ ”
- This would complete our second run of the algorithm. The set of random numbers generated during this run is  $[1, 2]$  ( 1 in the first iteration and 2 in the second iteration).

# Running on a small example

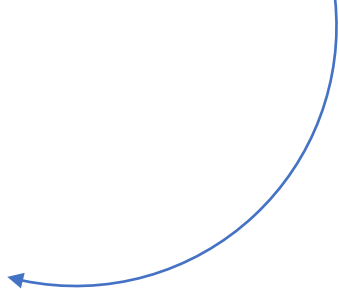
- Now, suppose we do “Four Runs” of the algorithm but each time the set of random numbers generated is different.

Runs	Condition of the array after 1st iteration (i = 1)	Condition of the array after 2nd iteration	Set of Random Numbers Generated
1	$A = \{1, 2\}$	$A = \{2, 1\}$	[1, 1]
2	$A = \{1, 2\}$	$A = \{1, 2\}$	[1, 2]
3	$A = \{2, 1\}$	$A = \{1, 2\}$	[2, 1]
4	$A = \{2, 1\}$	$A = \{2, 1\}$	[2, 2]

# Running on a small example

Runs	Condition of the array after 1st iteration (i = 1)	Condition of the array after 2nd iteration	Set of Random Numbers Generated
1	A = {1, 2}	A = {2, 1}	[1, 1]
2	A = {1, 2}	A = {1, 2}	[1, 2]
3	A = {2, 1}	A = {1, 2}	[2, 1]
4	A = {2, 1}	A = {2, 1}	[2, 2]

This column is called as the outcome space/ sample space, that is the set of all possible Random Number Generations.



# Running on a small example

Runs	Condition of the array after 1st iteration (i = 1)	Condition of the array after 2nd iteration	Set of Random Numbers Generated
1	$A = \{1, 2\}$	$A = \{2, 1\}$	[1, 1]
2	$A = \{1, 2\}$	$A = \{1, 2\}$	[1, 2]
3	$A = \{2, 1\}$	$A = \{1, 2\}$	[2, 1]
4	$A = \{2, 1\}$	$A = \{2, 1\}$	[2, 2]

These are called Outcomes

# Running on a small example

Runs	Condition of the array after 1st iteration (i = 1)	Condition of the array after 2nd iteration	Set of Random Numbers Generated
1	$A = \{1, 2\}$	$A = \{2, 1\}$	[1, 1]
2	$A = \{1, 2\}$	$A = \{1, 2\}$	[1, 2]
3	$A = \{2, 1\}$	$A = \{1, 2\}$	[2, 1]
4	$A = \{2, 1\}$	$A = \{2, 1\}$	[2, 2]

Now, a permutation generating algorithm will satisfy our objective only if the outcomes distribute uniformly over the permutations (i.e. the number of occurrences of each permutation should be same). Like we have in this example, both the permutations  $\{1, 2\}$  and  $\{2, 1\}$  occurred 2 times at the end of the last (second for this example) iteration.

# Guiding question

What would be a simple sanity check for any proposed permutation-generating algorithms?

# Guiding question

What would be a simple sanity check for any proposed permutation-generating algorithms?

**Answer:** We can check if it is even *possible* for the number of outcomes to distribute over each permutation *uniformly*. I.e. each permutation having equal representation in the outcome space.

Just calculate  $\text{\#outcomes} / \text{\#permutations}$  and see if we obtain a whole number. If we don't, we can confirm a true negative and can reject the algorithm. However if we do, it can either be a true positive or false positive. I.e. additional checks are needed.

# Guiding question

Will this pass our sanity check?

```
for ( $i$  from 1 to  $n$ ) do  
    Choose  $j = \text{random}(1, n)$   
    Swap( $A, i, j$ )  
end
```



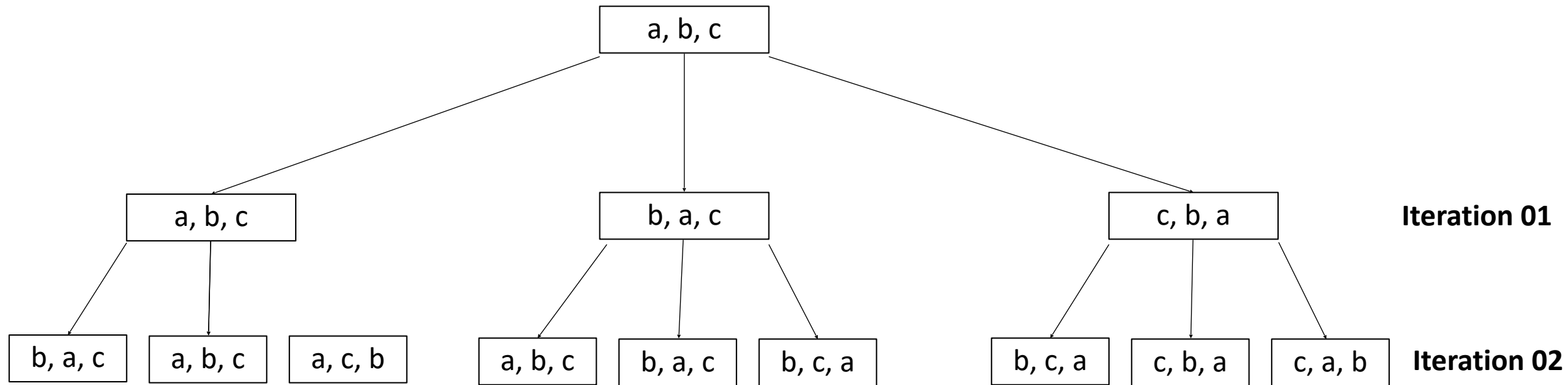
# Guiding question

Will this pass our sanity check?

```
for ( $i$  from 1 to  $n$ ) do  
    Choose  $j = \text{random}(1, n)$   
    Swap( $A, i, j$ )  
end
```

**Answer:** No it will not! This has  $n^n$  outcomes. There is no guarantee that  $n^n/n!$  will produce a whole number. Take for instance when  $n = 3 : 3^3/3! = 27/6 = 4.5$  which is not an integer!

# Example



In iteration 3, we will have 27 leaf nodes(Here  $n^n = 3^3 = 27$ ). But there are only  $n!(3! = 6)$  different permutations. Hence, we will not have equal probability for every unique permutation.

## Problem 2.d.

Consider the Fisher-Yates / Knuth Shuffle algorithm:

```
KnuthShuffle( $A[1..n]$ )
```

```
for ( $i$  from 2 to  $n$ ) do  
    Choose  $r = \text{random}(1, i)$   
    Swap( $A, i, r$ )  
end
```

What is the idea behind this algorithm?

Will this produce good permutations?

If so, are you able to come up with a simple proof of correctness?

“The Fisher–Yates shuffle is named after Ronald Fisher and Frank Yates, who first described it, and is also known as the Knuth shuffle after Donald Knuth.”

Source: [Wikipedia](#)

# Compare and contrast

## KnuthShuffle( $A[1..n]$ )

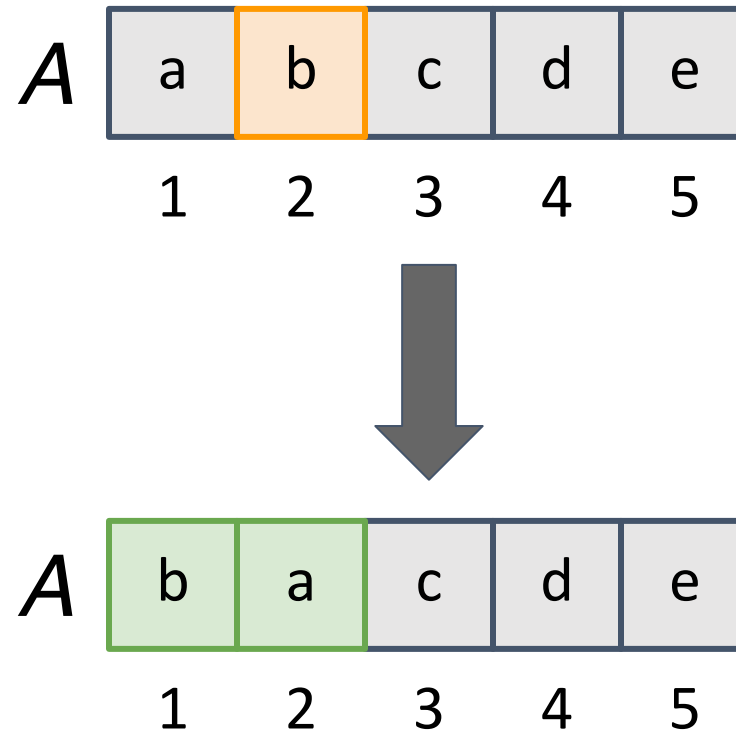
```
for ( $i$  from 2 to  $n$ ) do  
    Choose  $r = \text{random}(1, i)$   
    Swap( $A, i, r$ )  
end
```

*Note:* we can skip  $i$  from 1 since it's a redundant step where the first item swaps with itself.

## Problem 2.c.

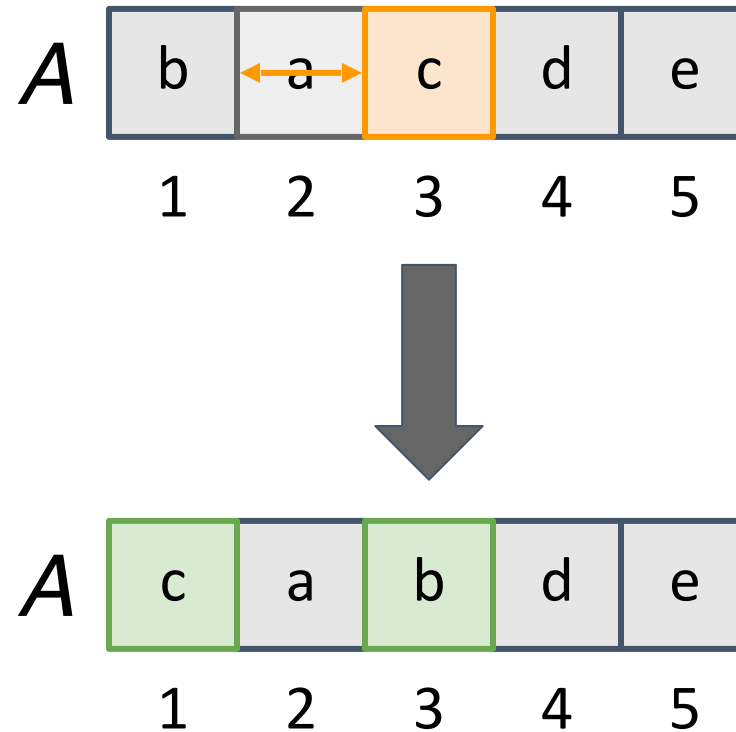
```
for ( $i$  from 1 to  $n$ ) do  
    Choose  $j = \text{random}(1, n)$   
    Swap( $A, i, j$ )  
end
```

# Behaviour trace



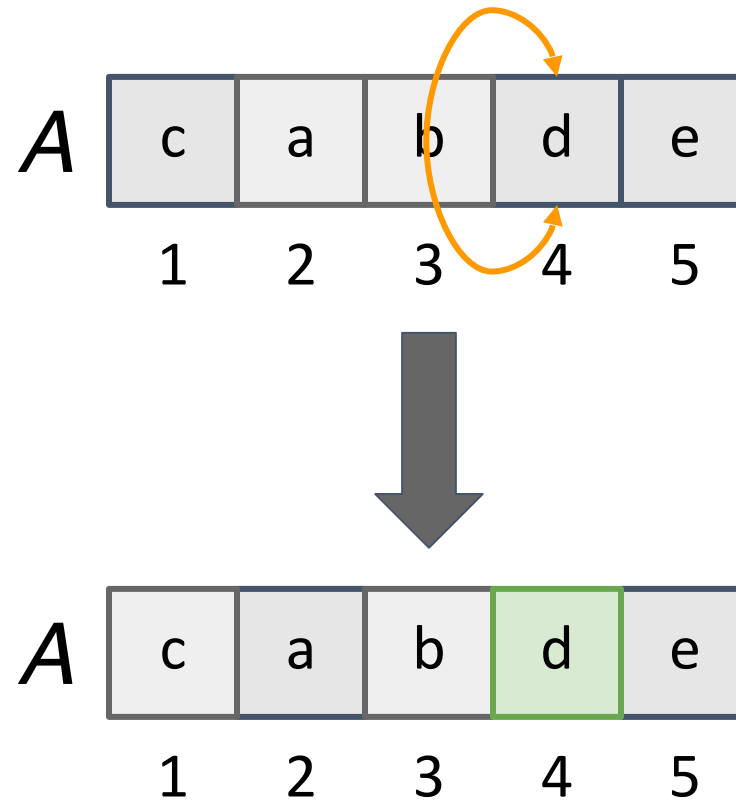
$i=2$ ,  $\text{random}(1,2)$  returned  $1$ , we swap index  
2 with 1

# Behaviour trace



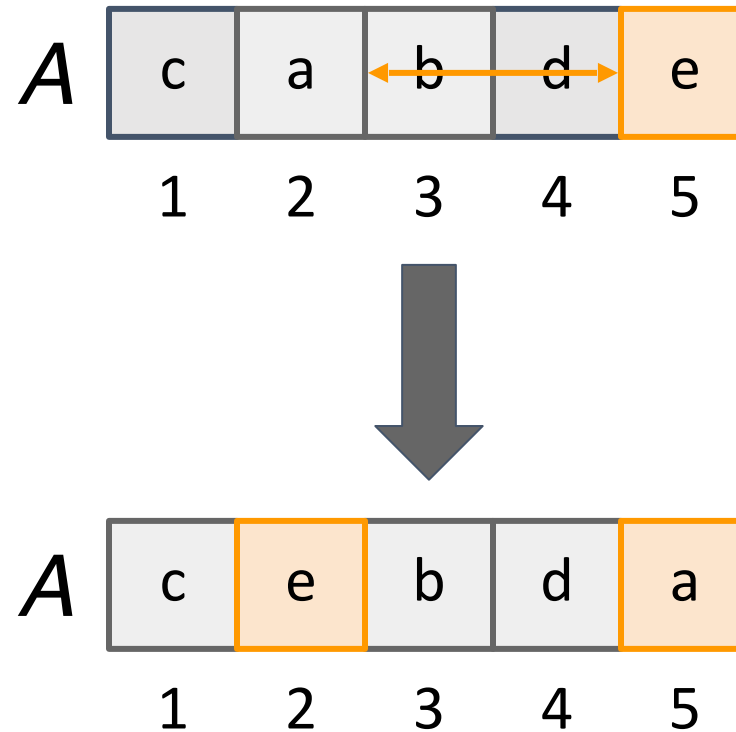
$i=3$ ,  $\text{random}(1,3)$  returned 1, we swap index  
3 with 1

# Behaviour trace



$i=4$ ,  $\text{random}(1,4)$  returned 4, we swap index  
4 with 4

# Behaviour trace



$i=5$ ,  $\text{random}(1,5)$  returned 2, we swap index 5 with 2  
We are done



# Guiding question

Can you prove the correctness of Knuth-shuffle and show that it will generate all permutations with equal probability?

*Hint:* Think about invariances

# Guiding question

What is the invariance for Knuth Shuffle?

# Guiding question

What is the invariance for Knuth Shuffle?

**Answer:** At the  $i^{\text{th}}$  iteration, we have a uniformly random permutation of the prefix with length  $i$ .

Realize that thinking of invariances here naturally lead you to the proof?

# Inductive proof outline

1. When  $i = 1$ , we get a permutation of the first 1 item(s) in the array (trivial)
2. Inductive Hypothesis : Assume for any  $k > 1$ ,  $i = k - 1$ , we get a uniformly random permutation of the first  $k - 1$  items in the array
3. When  $i = k$ , the  $k^{\text{th}}$  item in the array gets  $1 / k$  probability of being swapped anywhere into a uniformly random permutation of the first  $k - 1$  items (obtained from previous step)

Since [1] ( $i = 1$ ) is true and [2] ( $i = k - 1$ ) true implies [3] ( $i = k$ ) is also true, then  $i = 1, i = 2, \dots, i = n$  are all true and generate uniformly distributed prefix permutations of respective lengths  $i$ .

Therefore at the end of the routine, we would have grown the prefix to the entire array and obtained a uniformly random permutation of length  $n$ .