

# CS2040S

Recitation 3  
AY20/21S2

# Problem 1

## 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 1.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 solution meets them?

## 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  $1/n!$

Realise this objective entails a hard-constraint.

## Problem 1.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?

# 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



# Naive Shuffle version 1

**NaiveShuffleV1( $A[1..n]$ )**

```
for ( $i$  from 1 to  $n$ ) do  
  do  
    Choose  $j = \text{random}(1, n)$   
    while  $A[j]$  is picked  
       $B[i] = A[j]$   
      mark  $A[j]$  as picked  
end
```

## Guiding question

What is one issue with NaiveShuffleV1 proposed?

## Guiding question

What is one issue with NaiveShuffleV1 proposed?

**Answer:** The probability of randomly selecting a previously picked item increase as we progress, leading to more and more time spent on repicking the random index.

As  $n$  gets very large, the probability of having to keep repicking a random index for the last slot approaches 100%!

## Naive Shuffle version 2

**NaiveShuffleV2( $A[1..n]$ )**

**for** ( $i$  from 1 to  $n$ ) **do**

    Choose  $j = \text{random}(1, n - i)$

$B[i] = A[j]$

    delete  $A[j]$

**end**

## Guiding question

What is the time complexity of NaiveShuffleV2?

## Guiding question

What is the time complexity of NaiveShuffleV2?

**Answer:** Each delete item operation costs  $O(n)$  so in total  $O(n^2)$  to pick all  $n$  items in  $A$ .

## Naive Shuffle version 3

**NaiveShuffleV3( $A[1..n]$ )**

```
for ( $i$  from 1 to  $n$ ) do:  
    Choose  $j = \text{random}(1, n)$   
    while  $A[j]$  is picked do           // linear probing  
         $j = j + 1$   
        if  $j > n$  do                   // wrap-around  
             $j = 1$   
        end  
    end  
     $B[i] = A[j]$   
    mark  $A[j]$  as picked  
end
```

## Guiding question

What is the time complexity of NaiveShuffleV3?



## Guiding question

What is the time complexity of NaiveShuffleV3?

**Answer:** Since in the worst case, each linear probe for finding an unpicked item will take  $O(n)$  time, the total time for this algorithm is also  $O(n^2)$ .

## Naive Shuffle++

**NaiveShuffle++( $A[1..n]$ )**

**for** ( $i$  from 1 to  $n$ ) **do**

    Choose  $j = \text{random}(1, n - i)$

$B[i] = A[j]$

    Swap( $A, j, n - i + 1$ )                      // throw picked ones to the rear

**end**

## Guiding question

What is the time complexity of NaiveShuffle++?

## Guiding question

What is the time complexity of NaiveShuffle++?

**Answer:**  $O(n)$  time.

Kudos to the students who proposed this during class!

# Sorting shuffle

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

$A$	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

$B$	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

## Sorting shuffle: Duplicates values

What if there are equivalent numbers in the random numbers assigned? (Remember, computers have finite precision!)

2 options:

1. Re-run the Sorting Shuffle again until this doesn't happen
2. Amongst equivalent numbers, break ties by choosing new random numbers for them

But wait.. how do you even detect equivalent numbers efficiently?

# Sorting shuffle: duplicate assignments

Does duplicate assignments actually matter?

- If sort is stable, then perhaps we can just treat the earlier occurrences of the number to be lesser than the ones that appear after them
- If sort is not stable, then the sort will decide on a relative ordering based on its sorting procedure (note this is *not* a random ordering!)

However we should probably still break ties because

- It *guarantees* randomness (agnostic to sorting algorithm used)
- For the same random range being sampled, when array length increases, we expect duplicate numbers to occur more frequently. This means the solution might not scale well for large  $n$  as randomness of permutations degrade

# Clever Shuffle?

Idea: Modify compareTo to return a random value and sort.

```
public int compareTo(Object other) {  
    double r = Math.random();  
    if (r < 0.5) return -1;  
    if (r > 0.5) return 1;  
    return 0;  
}
```

“Hijacking” sorting procedure to generate permutations. Seems clever right? Does this work?



# Buggy shuffle

2 problems with this approach:

1. In order for sorting to have meaning, `compareTo` should always return the same answer and must be transitive
2. Turns out this does not actually yield a random permutation!

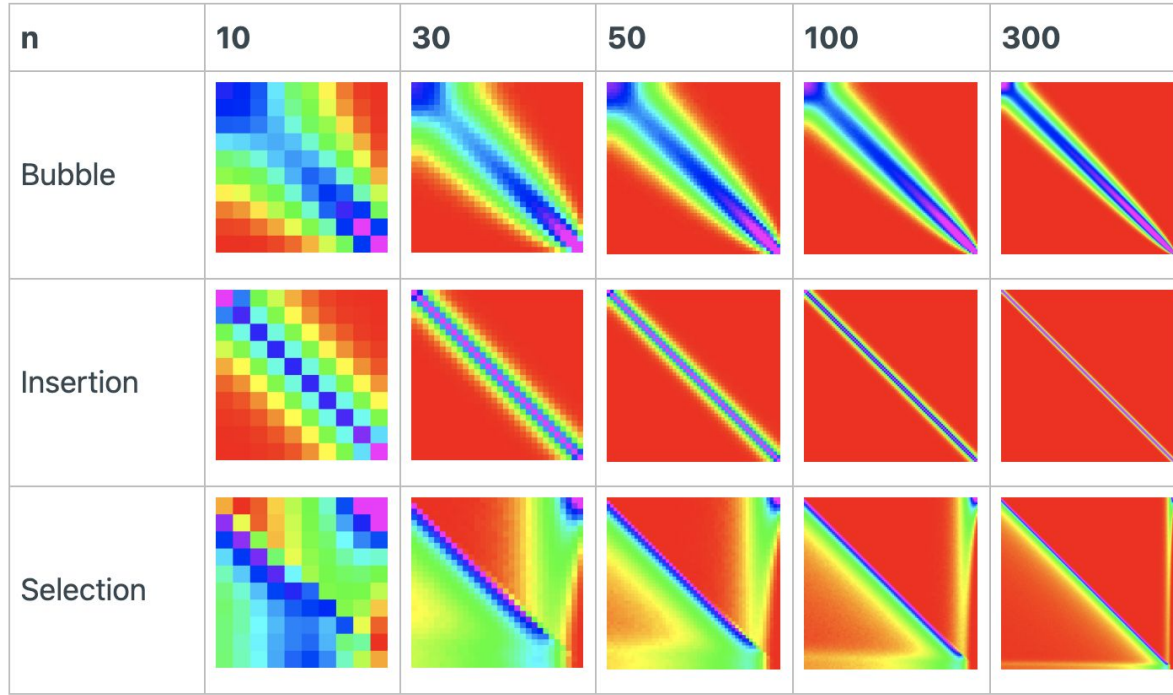
Did you know, this was an actual Javascript bug found in Windows 7! See [here](#) for more information.

# Challenge yourself!

Can you prove the following claim?

For Insertion Sort, the probability that the first item remains in the same spot after buggy shuffle is  $\geq \frac{1}{4}$  instead of  $1/n$ .

# Just for fun!



Someone actually analyzed random comparators on many sorting algorithms and reported the findings [here](#).

## Problem 1.c.

Does the following algorithm work?

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

## 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!



## Problem 1.d.

Consider the following algorithm

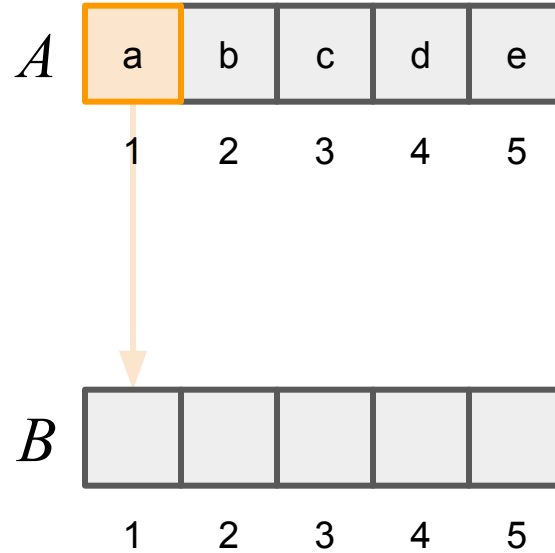
Source array  $A$

Destination array  $B$

```
for ( $i$  from 1 to  $n$ ) do                                // Build permutation prefix from  $i$  to  $n$   
    Choose  $j = \text{random}(1, i-1)$  // Choose random position in current permutation prefix  
     $B[i] = B[j]$  // Copy random element to end  
     $B[j] = A[i]$  // Insert next item from  $A$  into the random slot  
end
```

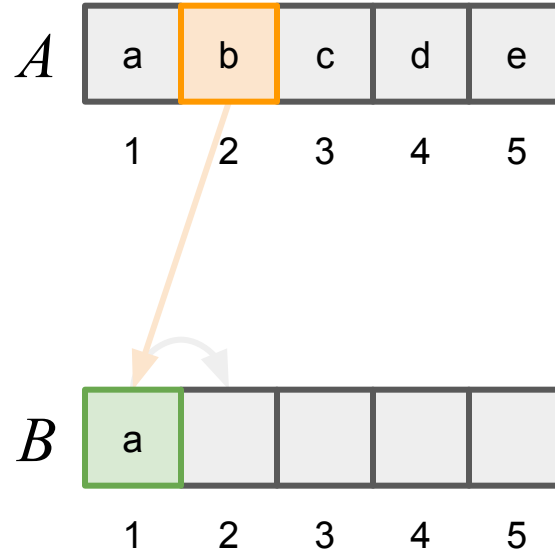
What is the idea behind this? Will this produce good permutations? If so why? If not, how do you fix it?

# Behaviour trace



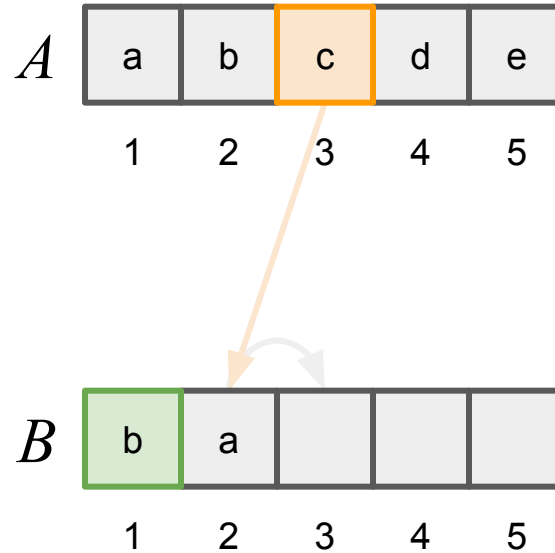
Initial step: We shall just copy first element over

# Behaviour trace



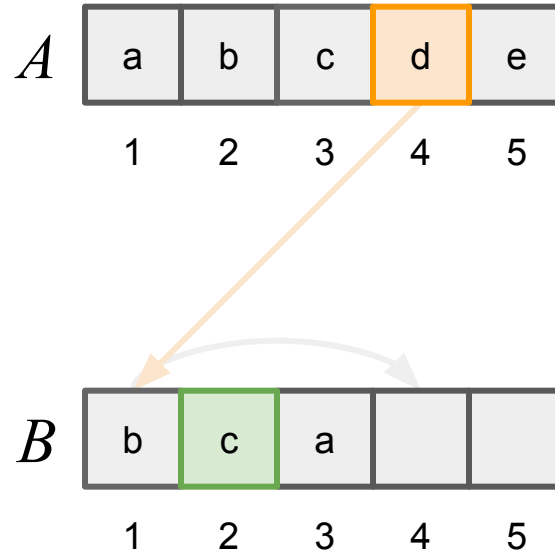
$i=2$ ,  $\text{random}(1,1)$  returns 1 so  $B[1]$  goes to  $B[2]$  and  $A[2]$  goes to  $B[1]$ .

# Behaviour trace



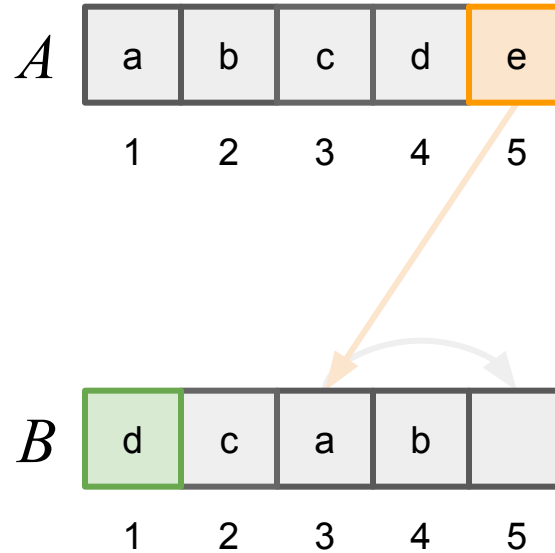
$i=3$ ,  $\text{random}(1,2)$  returns 2 so  $B[2]$  goes to  $B[3]$  and  $A[3]$  goes to  $B[2]$ .

# Behaviour trace



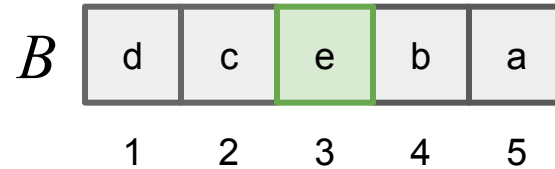
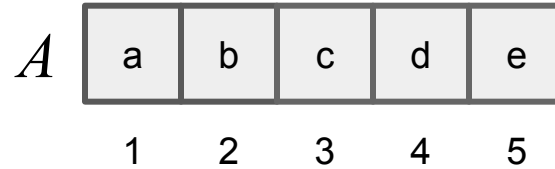
$i=4$ ,  $\text{random}(1,3)$  returns 1 so  $B[1]$  goes to  $B[4]$  and  $A[4]$  goes to  $B[1]$ .

# Behaviour trace



$i=5$ ,  $\text{random}(1,4)$  returns 3 so  $B[3]$  goes to  $B[5]$  and  $A[5]$  goes to  $B[3]$ .

# Behaviour trace



We are done!

## Guiding question

What is the idea behind this algorithm?



## Guiding question

What is the idea behind this algorithm?

**Answer:** Instead of going through each slot in  $B$  and picking a random item in  $A$ , this goes through each item in  $A$  and selects a random slot for it in  $B$ .

## Guiding question

Will this produce good permutations?

*Hint:* Is there a type of permutation it can never produce?

## Guiding question

Will this produce good permutations?

*Hint:* Is there a type of permutation it can never produce?

**Answer:** It can never produce permutations at *fixed points* because every element will be shuffled *out* of its original position.

## Problem 1.e.

How can we turn the previous algorithm into an in-place one?

## Problem 1.e.

How can we turn the previous algorithm into an in-place one?

**Answer:** Fisher-Yates/Knuth-shuffle

If we just want an in-place algorithm, then even a naive one (from 1.b.) would suffice.

# Fisher-Yates/Knuth-shuffle

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

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

“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( $A[1..n]$ )**

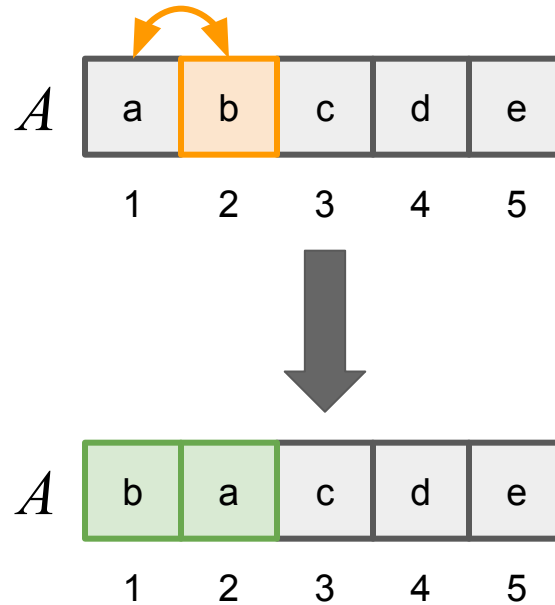
```
for ( $i$  from 2 to  $n$ ) do  
    Choose  $r = \text{random}(1, i)$   
    swap( $A, i, r$ )  
end
```

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

**Problem 1.c.**

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

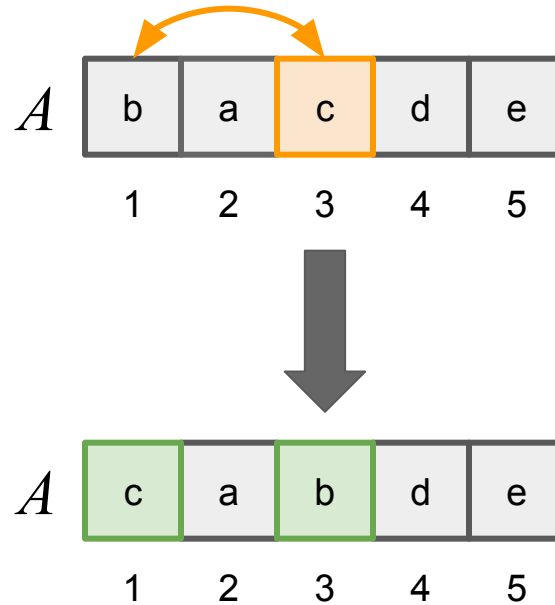
# Behaviour trace



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

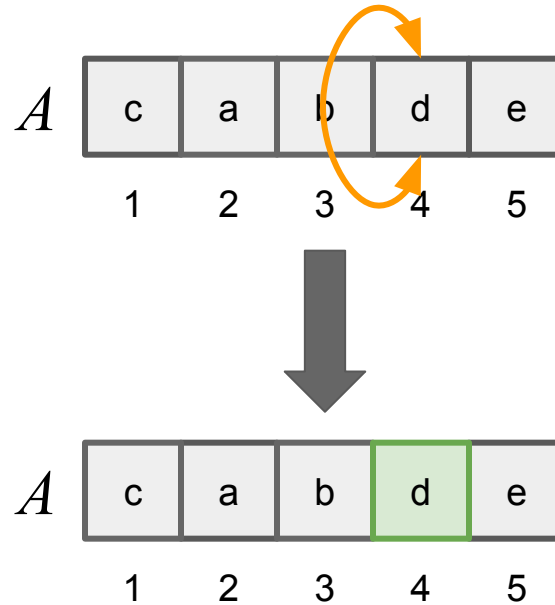


# Behaviour trace



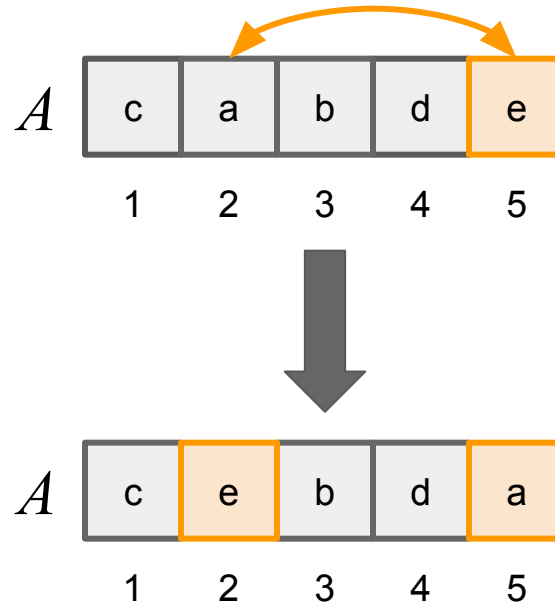
$i=3$ ,  $\text{random}(1,3)$  returned 1, we swap index 3 with 1

# 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

Challenge yourself!

Can you write out the recurrence relation for Knuth-shuffle and implement it as a recursion?

## Challenge yourself!

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

*Hint:* You just need to prove (inductively?) that the probability of an item ending up at a specific slot is always  $1/n$ .

# Description

Now let's consider another possible application of permutations. To handle grading a 500 person class without exhausting the tutors, suppose we decided to have each student grade another student's work.

So, for PS5, we will do as follows:

1. Generate a random permutation  $B$  of the students
2. Assign student  $A[i]$  to grade the homework of student  $B[i]$

## Problem 1.f.

If you use solutions from 1.b or 1.e, what is the expected number of students that'll have to grade their own homework in one random permutation?

What about the algorithm proposed in 1.d. Which is the better algorithm?

What's the moral of the story here?

## Guiding question

For a uniformly random permutation, what is the expectation that an element remains in its spot after shuffling?



## Guiding question

For a uniformly random permutation, what is the expectation that an element remains in its spot after shuffling?

**Answer:** Probability of a specific element remaining in its spot is  $(n-1)!/n! = 1/n$

Thus the expected number of elements which experience this is  $n \times 1/n = 1$

# Guiding question

Out of all the shuffling algorithms so far, which is the better algorithm here?

What's the moral of the story?

## Guiding question

Out of all the shuffling algorithms so far, which is the better algorithm here?

What's the moral of the story?

**Answer:** The algorithm from 1.d. is better because it explicitly avoids fixed point permutations and therefore the expected number of students grading their own assignment is zero.

Metrics of success is now different from the previous parts. Once again, it is the metrics we choose that drives the solution. An algorithm itself is neither good nor bad if its without the problem context it serves.

# Problem 2

# Description

- Seth is giving a Zoom lecture
- Questions that are streaming in
- Seth will just randomly pick one question to answer in the end
- Fairness must be ensured with every question having an equal chance of being answered
- However Seth only intends to bear one question in mind at any one moment

## Problem 2.a.

How will Seth accomplish this?

Realize that if he didn't pick a question the moment it was posed then the opportunity will pass because he cannot come back to pick it later.

How can Seth know which question to pick (as they are streaming in) without knowing the total number of questions ahead of time?

## Guiding question

For a stream with  $n$  total questions, what should be the probability of picking a question under a fair scheme?

## Guiding question

For a stream with  $n$  total questions, what should be the probability of picking a question under a fair scheme?

**Answer:**  $1/n$



# Reservoir sampling

- Maintain only one item in the pocket
- Initially the first item is picked
- Replace the item in pocket with  $i^{\text{th}}$  item in the stream with probability  $1/i$

## Guiding question

What is the scenario for which we pick item  $i$  in the end?

## Guiding question

What is the scenario for which we pick item  $i$  in the end?

**Answer:** That's the scenario in which we replaced our item in pocket with item  $i$  and thereafter did not ever replace it again.

# Analysis

$$P(i) = \frac{1}{i} \times \left(1 - \frac{1}{i+1}\right) \times \left(1 - \frac{1}{i+2}\right) \times \cdots \times \left(1 - \frac{1}{n-1}\right) \times \left(1 - \frac{1}{n}\right) \quad (1)$$

$$= \frac{1}{i} \times \frac{i}{i+1} \times \frac{i+1}{i+2} \times \cdots \times \frac{n-2}{n-1} \times \frac{n-1}{n} \quad (2)$$

$$= 1/n \quad (3)$$

Test yourself!

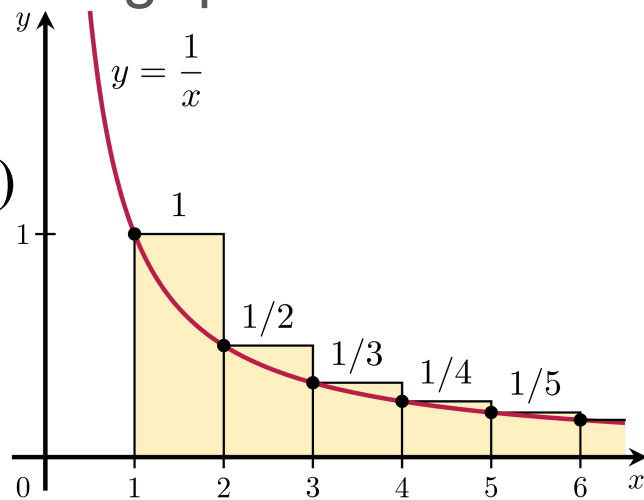
How often do you *expect* to swap an existing question for a new one?

# Test yourself!

How often do you *expect* to swap an existing question for a new one?

**Answer:**  $1/1 + 1/2 + 1/i + \dots + 1/n \approx O(\log n)$

This is the none other than the harmonic series! It's a *divergent* series, but can be treated as  $O(\log n)$ .



$$\sum_{n=1}^k \frac{1}{n} > \int_1^{k+1} \frac{1}{x} dx = \ln(k+1).$$

Source: [Wikipedia](#)

# Cool idea

In practice,

This means that if you can skip parts of the stream by fast-forwarding, you might only need to look at  $O(\log n)$  elements in the stream in expectation.

## Problem 2.b.

What if he wishes to answer  $k$  random questions instead?



## Guiding question

For a stream with  $n$  total questions, now what should be the probability of picking a question under a fair scheme?

## Guiding question

For a stream with  $n$  total questions, now what should be the probability of picking a question under a fair scheme?

**Answer:**  $k/n$

# Generalised reservoir sampling

- Maintain  $k$  slots in the pocket
- Initially the first  $k$  items are picked
- Pick  $i^{\text{th}}$  item in the stream with probability  $k/i$
- If item is picked, *randomly choose* one of the  $k$  slots in pocket for replacement

## Guiding question

What is the probability now for picking item  $i$  *and* swapping it with a specific slot? What is therefore the probability of *not* having a specific slot replaced during a trial?

## Guiding question

What is the probability now for picking item  $i$  *and* swapping it with a specific slot? What is therefore the probability of *not* having a specific slot replaced during a trial?

**Answer:** The probability now for picking item  $i$  *and* swapping it with a specific slot is  $k/i \times 1/k = 1/i$ .

Therefore the probability of not having a specific slot replaced during a trial is  $1 - 1/i$ . This is the same as before :)

# Problem 3

## Description

We saw how to generate a random permutation from an array.

Now how would you generate a random binary tree?

But wait, what is a random tree?

## Problem 3.a.

Recall from your problem set that a perfectly balanced binary tree is defined to be a binary tree in which for every node, the sizes of its left and right subtrees differ by at most one.

Consider now a *full binary tree* in which every node other than the leaves has two children. For such a tree with  $n$  nodes, what can you say about  $n$ ?

Given an array  $A$  of  $n$  items (assuming all items are unique), how do you randomly generate a full binary tree from the items?



## Important note

In this question, we aren't interested in the actual values in the array because we are not building *binary search* trees.

The values in the array merely serve to *uniquely identify* a node. I.e. think of them as node IDs

## Guiding question

For a full-binary tree of size  $n$ , what can you say about  $n$ ?

## Guiding question

For a full-binary tree of size  $n$ , what can you say about  $n$ ?

**Answer:**  $n = 2^k - 1$  where  $k$  is some positive integer.

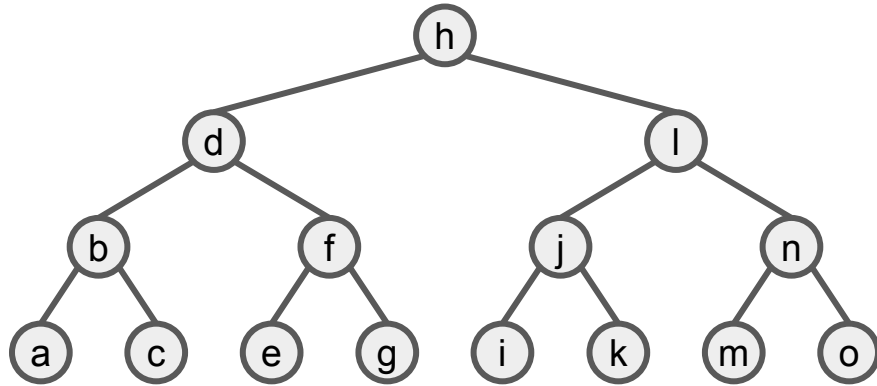
## Method 1: Center parting

- Take the middle element  $m$  in the current subarray
- Assign  $m$  as the root and use it to partition the subarray into two halves
- Recurse into left partition [subarray start,  $m - 1$ ]
- Recurse into right partition [ $m + 1$ , subarray end]

## Method 1: Example

Suppose we have the following array where  $n=15=2^4-1$

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

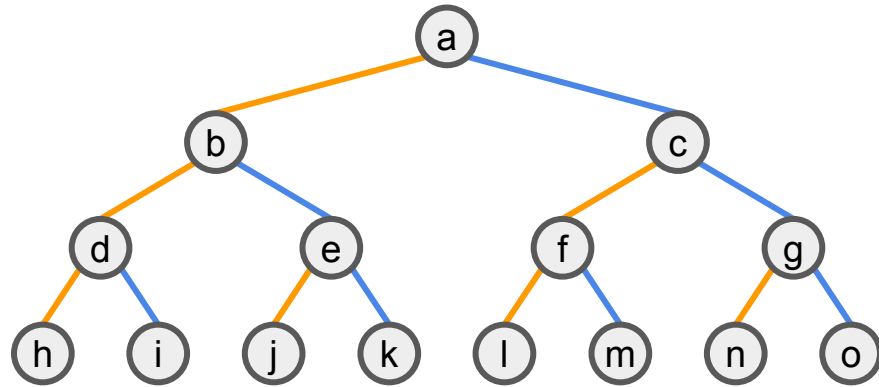
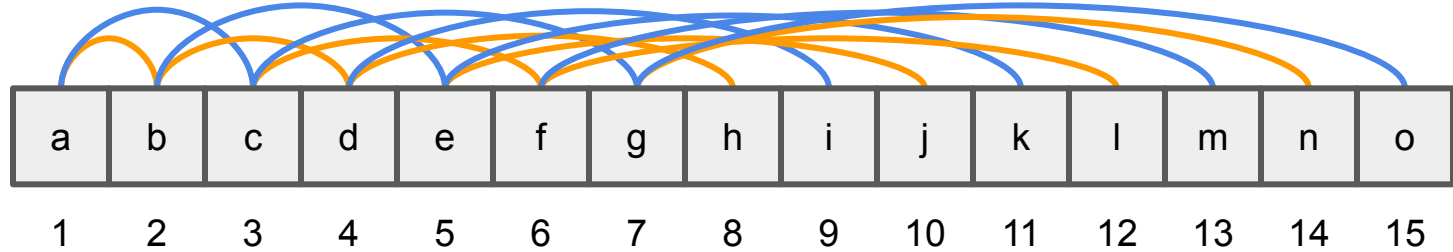


## Method 2: Complete binary tree

- We construct the tree by iterating elements from  $i$  to  $n$
- For each node  $i$ 
  - Left child is element  $2i$
  - Right child is element  $2i+1$

*Note:* A full binary tree is also a complete binary tree. We will revisit this idea when we look at heaps.

## Method 2: Example



## Problem 3.b.

Now what if you want to explore all possible binary trees (i.e., doesn't need to be balanced).

How will the algorithm look like and can you write down the time complexity recurrence for it?



# Approach

Iterate each item  $i$  in array:

- Select  $i$  as the root
- $i$  splits the array into 2 partitions
- Recursively build each partition as a subtree

What is the time complexity recurrence of this?

# Complexity

$$T(n) = \sum_{i=1}^n [T(i-1) + T(n-i) + 1]$$

Explore each  
node  $i$  as the root  
for current subtree

Time taken to build  
left subtree based  
on chosen root  $i$

Time taken to build  
right subtree based  
on chosen root  $i$

Constant time for  
current step  
operations

### Problem 3.c.

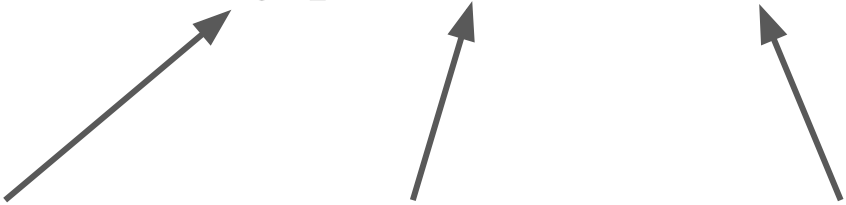
Suppose the number of different binary trees you can construct with  $n$  nodes is counted by  $c(n)$ , express it as a recurrence relation.

Try working out the answers for  $c(1)$ ,  $c(2)$ ,  $c(3)$ ,  $c(4)$ ,  $c(5)$ .  
(Bonus: Do you know the combinatorial formula for these numbers?)

Do you know what is the significance of these numbers?

# Recurrence relation

$$c(n) = \sum_{i=1}^n [c(i-1) \times c(n-i)]$$



Explore each  
node  $i$  as the root  
for current subtree

#variations for left  
subtree by choosing  
 $i$  as root

#variations for right  
subtree by choosing  
 $i$  as root

## Guiding question

$n$	$c(n)$
1	
2	
3	
4	
5	

Do you know the significance of these numbers?

## Guiding question

$n$	$c(n)$
1	1
2	2
3	5
4	14
5	42

Do you know the significance of these numbers?

**Answer:** These are [Catalan numbers](#)!

Those interested to see how it can be derived into the combinatorics formula can see [here](#) (it's tedious!).

## Problem 3.d.

Consider the following strategy in constructing a random binary tree:

1. Assign each node a random number which we will call its priority
2. Now build a tree that maintains priority hierarchy: the root of each subtree is the node in that subtree with the highest priority

That should should also work, right? How would you implement it?

# Approach

Select item  $m$  with the *maximum priority* in the current subarray:

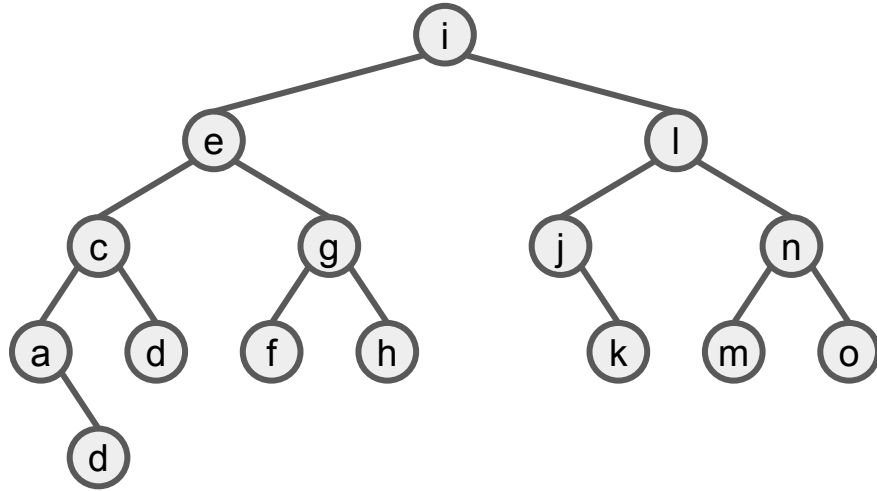
- Select  $m$  as the root
- $m$  splits the array into left and right partitions
- Recurse on each partition to construct them as subtrees



# Example

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Priorities: 23 19 71 2 83 63 64 12 91 47 10 55 7 12 5



### Problem 3.e.

What is the sum of node depths in the tree from the previous part? What is the expected depth of the tree?

We shouldn't have to answer these two questions formally because something we have already seen should imply their answers. What is it?

## Guiding question

This tree serves as a graphical representation of which algorithm that we have seen before?

## Guiding question

This tree serves as a graphical representation of which algorithm that we have seen before?

**Answer:** This is Quick Sort! The random priorities represent the random pivots we choose in every partition. Now they are just “pre-selected”.

## Guiding question

Which operation is the main contributor of time complexity across all comparison-based sorting algorithms?

## Guiding question

Which operation is the main contributor of time complexity across all comparison-based sorting algorithms?

**Answer:** Comparisons! Realize this is why we can evaluate any comparison-based sorting algorithm purely based on the number of comparisons it require.

## Guiding question

What does the depth of a node  $i$  in the tree represent?

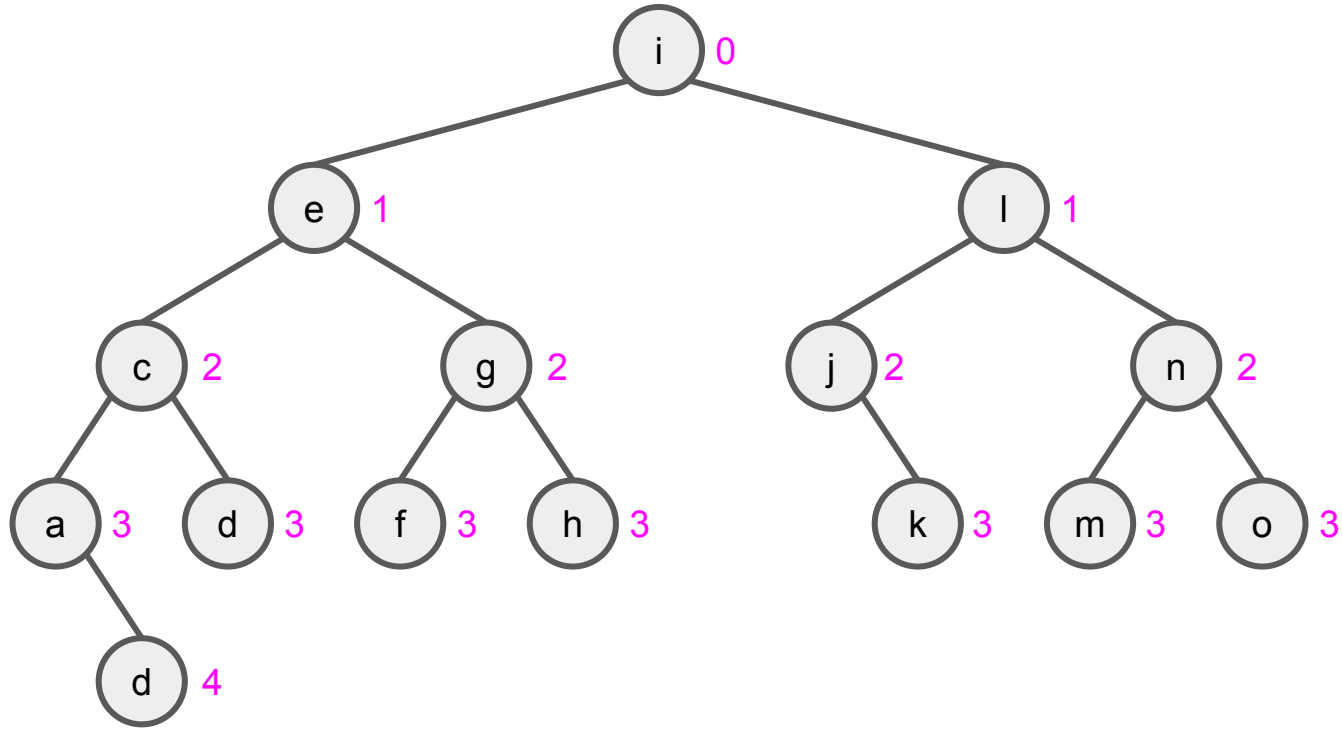
## Guiding question

What does the depth of a node  $i$  in the tree represent?

**Answer:** All the number of comparisons against *previous pivots* before  $i$  itself becomes a pivot. In other words, the number of comparisons until  $i$  is swapped to its rank.



# Node depths



## Node depths

Therefore the sum of node depths in this tree is nothing but the total number of comparisons needed to run Quick Sort, which is  $O(n \log n)$ .

Realize the average node depth is  $O(n \log n) / n = O(\log n)$ , which implies that we expect the tree to have a short height of just  $O(\log n)$ . This again concurs with our understanding of Quick Sort which has an expected  $O(\log n)$  levels of recursion.