CS2040S

Recitation 3
AY20/21S2

Problem 1

Description

In the “DNA Sequence” problem from the previous recitation, we discussed how to modify MergeSort to sort a string with only 2 types of characters by using only substring reversals.

Problem 1.a.

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 2.a. solution with minimal modifications?

Guiding question

Which step in QuickSort can we implement using our 2.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 2.a. solution

Guiding question

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

Answer: $\mathrm{T}(n) = 2\mathrm{T}(n/2) + \mathrm{O}(n \log n)$

Complexity analysis: loose but intuitive

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

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

Complexity analysis: rigourous

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

Continued next page..

Complexity analysis: rigourous (cont'd)

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

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

$$= O(n[\log^2 n - (\log n)(\log n + 1)/2]) \qquad \text{Open up summations}$$

$$= O(n[\log^2 n - (\log^2 n + \log n)/2]) \qquad \text{Expand bracket}$$

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

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

$$= O(n[\log^2 n - \log n]) \qquad \text{Constant factors ignored by big O}$$

$$= O(n\log^2 n - n\log n]) \qquad \text{Opening up bracket}$$

$$= O(n\log^2 n) \qquad \text{Taking big O}$$

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: $\mathrm{O}(n)$ 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 :)

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 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 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 $1/n!$

Realise this objective entails a hard-constraint.

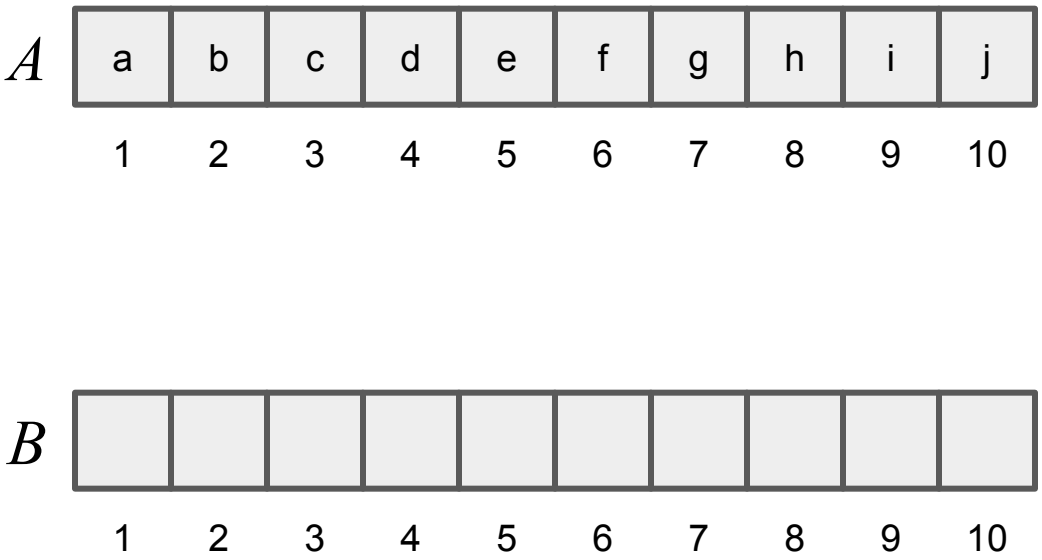
Problem 2.b.

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.

Discussion template



Naive Shuffle version 1

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

for (i from 1 to n) **do**
    do
        Choose j = 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 = random(1, $n - i + 1$)
    $B[i] = A[j]$
    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 $\mathrm{O}(n)$ so in total $\mathrm{O}(n^2)$ to pick all n items in A.

Naive Shuffle version 3

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

```
for (i from 1 to n) do:
        Choose j = 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 are some issues of NaiveShuffleV3?

Guiding question

What are some issues of NaiveShuffleV3?

Answer: 1. This does not generate random permutation uniformly!

2. This take O(n^2) running time at worst cases.

Proof of error:

- Suppose in the first iteration item k was picked
- In the next iteration, the probability of picking the k+1 th item is 2/n because 1/n chance of randomly landing on it plus 1/n chance of landing on the already picked kth item
- For a uniformly random permutation, the probabilty of picking any second item should be 1/(n-1)
- In this buggy algorithm, the probability of items just after already picked items have higher probability of getting picked!

Guiding question

What is the underlying strategy of all the Naive Shuffles we've seen so far?

Guiding question

What is the underlying strategy of all the Naive Shuffles we've seen so far?

Answer: Build the permutation prefix from 1 to n. This is akin to selection sort, except that we randomly select an item each time to grow the "sorted" region.

Discuss

Using the same prefix-building strategy in the Naive Shuffles we've discussed so far, can you come up with a version that runs in $O(n)$ time?

Naive Shuffle++

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

for (i from 1 to n) **do**
    Choose j = random(1, $n-i+1$)
    $B[i] = A[j]$
    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: $\mathrm{O}(n)$ time.

Kudos to the students who proposed this during class!

Guiding question

How many sets of permutations does NaiveShuffle++ produce per run?

Guiding question

How many sets of permutations does NaiveShuffle++ produce per run?

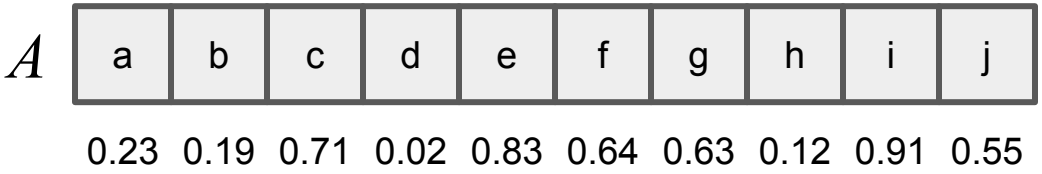
Answer: 2 sets! Realize A and B would be the reverse of each other at the end of the shuffle routine.

Challenge yourself!

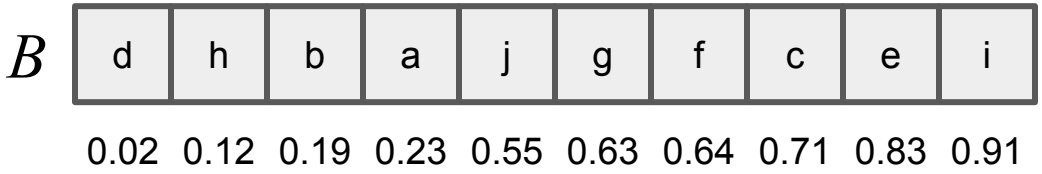
Can you turn Naive Shuffle++ into an in-place algorithm?

Sorting shuffle

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



Step 2: Sort based on assigned random numbers



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 occurences 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!

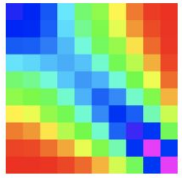
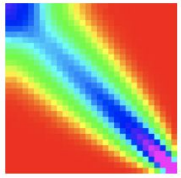
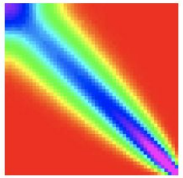
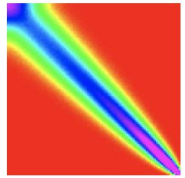
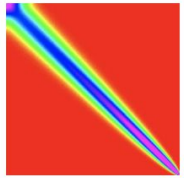
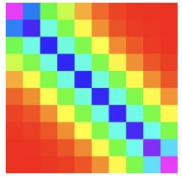
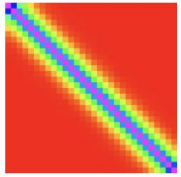
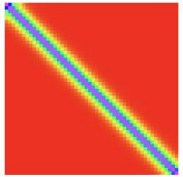
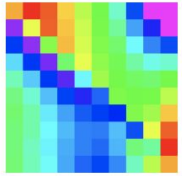
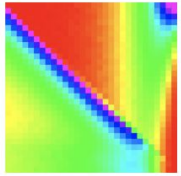
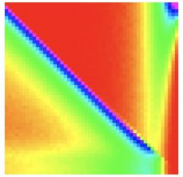
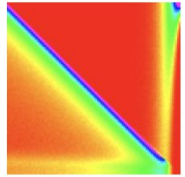
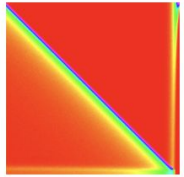
Did you know, this was an actual Javascript bug found in Windows 7! See here for more information.

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 ≥ ¼ instead of $1/n$.

Just for fun!

n	10	30	50	100	300
Bubble					
Insertion					
Selection					

Someone actually analyzed random comparators on many sorting algorithms and reported the findings here.

Problem 2.c.

Does the following algorithm work?

```
for (i from 1 to n) do
    Choose j = 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 #outcomes / #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 = 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 = random(1, n)
    Swap(A, i, j)
end
```

Answer: No it will not! This has n^n outcomes because each of the n iterations get n possible swaps. 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 2.d.

Consider the Fisher-Yates / Knuth Shuffle algorithm:

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

for (i from 2 to n) **do**
    Choose r = 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

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?

Compare and contrast

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

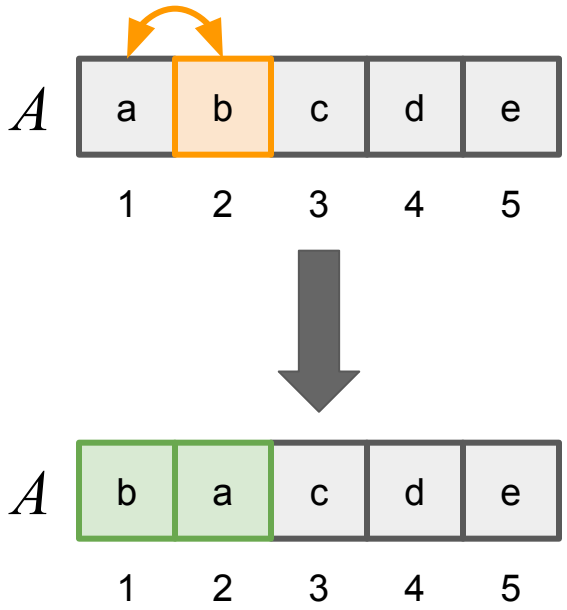
for (i from 2 to n) **do**
    Choose r = random(1, i)
    Swap(A, i, r)
end

Problem 2.c.

for (i from 1 to n) **do**
    Choose j = random(1, n)
    Swap(A, i, j)
end

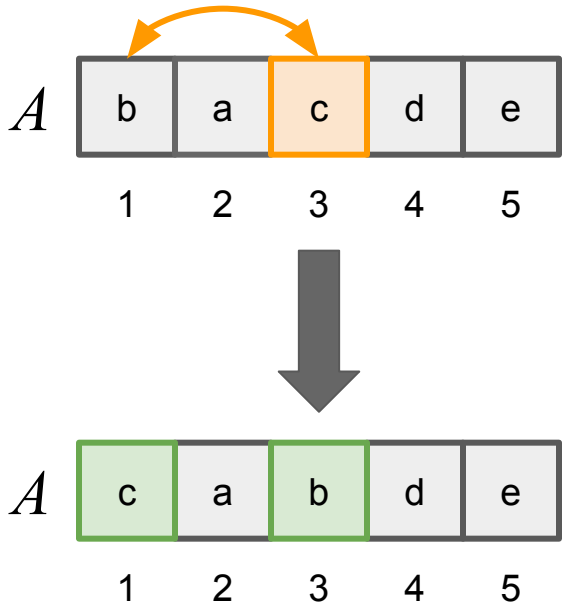
Note: we can skip i from 1 since it's a redundant step where the first item swaps with itself.

Behaviour trace



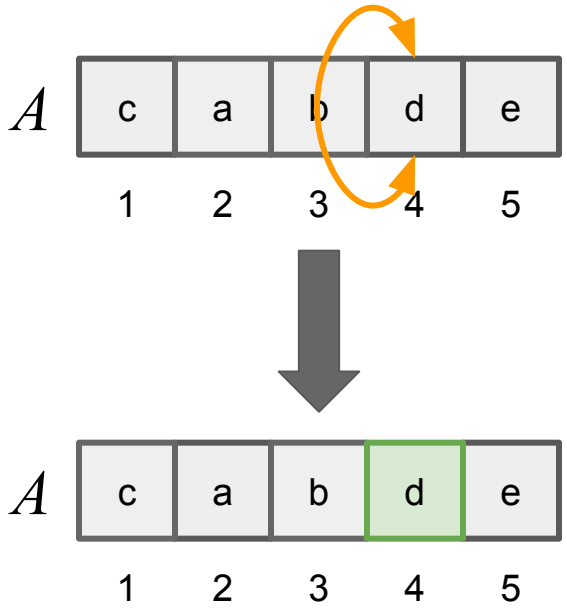
$i = 2$, random(1,2) returned 1, we swap index 2 with 1

Behaviour trace



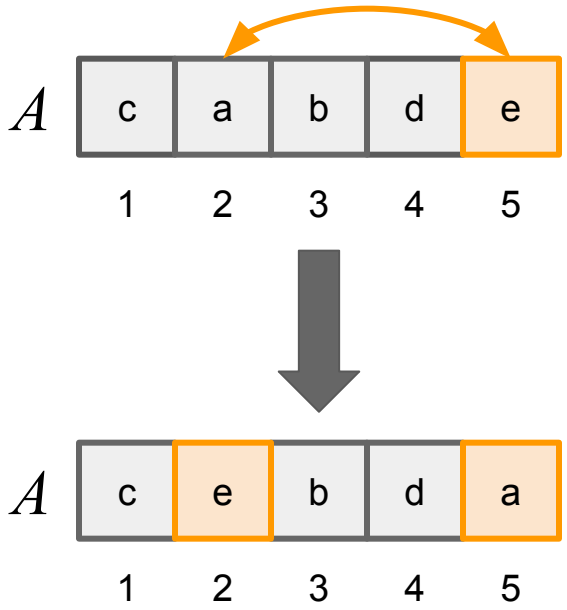
$i = 3$, random(1,3) returned 1, we swap index 3 with 1

Behaviour trace



$i = 4$, random(1,4) returned 4, we swap index 4 with 4

Behaviour trace



$i = 5$, 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?

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^{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] When $i = k - 1$, we get a uniformly random permutation of the first $k - 1$ items in the array
[3] When $i = k$, the k^{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 $i = 1$, $i = 2$, … , $i = n$ 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.

Description

Now let's consider a problem that can be approached by permutation-generation. There are currently 650 students enrolled in CS2040S. To handle grading such a large 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. Given a roster A of the class, generate a random permutation B of the students
2. Assign student $A[i]$ to grade the homework of student $B[i]$

Problem 2.e.

If you use solutions from 2.b or 2.d, what is the expected number of students that'll have to grade their own homework in one random permutation?

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 $n \times 1/n = 1$

Problem 2.f.

How might we modify and adapt Knuth Shuffle to this problem?

Guiding question

Can we use Knuth Shuffle for this problem?

Guiding question

Can we use Knuth Shuffle for this problem?

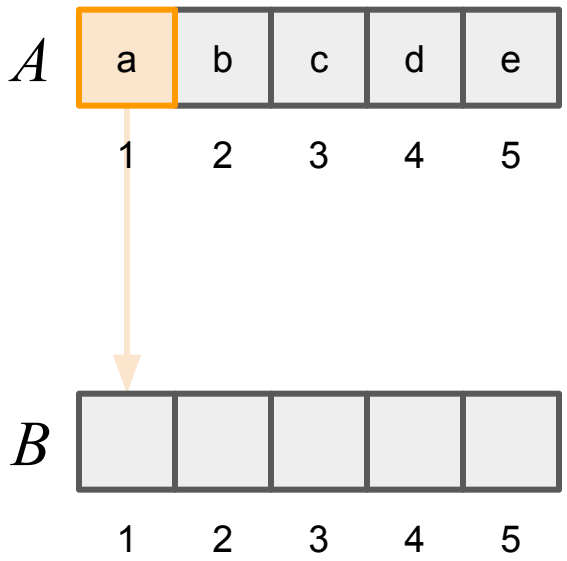
Answer: Not without modifications! 2 issues to address:

1. We don’t want permutations at *fixed points* but want to keep all other permutations equally likely.
2. Original Knuth Shuffle is an in-place algorithm. We shouldn’t shuffle the original roster A.

Solution

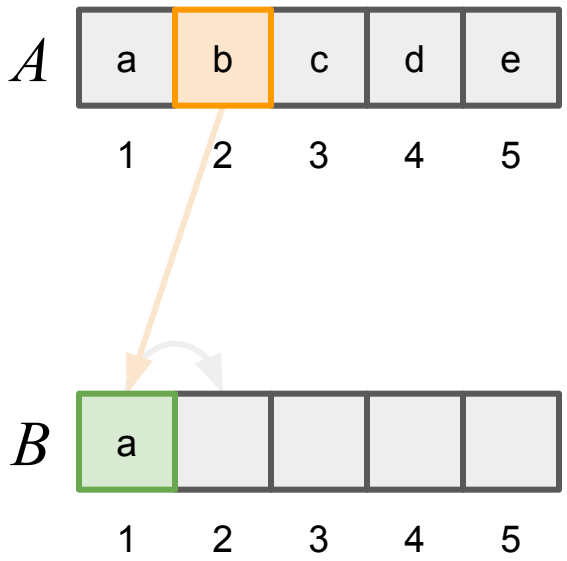
```
Source array A
Destination array B
for (i from 1 to n) do                  // Build permutation prefix from 1 to n
        Choose j = random(1, i − 1)     // Choose random position in current prefix size - 1
        B[i] = B[j]                         // Copy random element to end of prefix
        B[j] = A[i]                         // Insert next item from A into the random slot
end
```

Behaviour trace



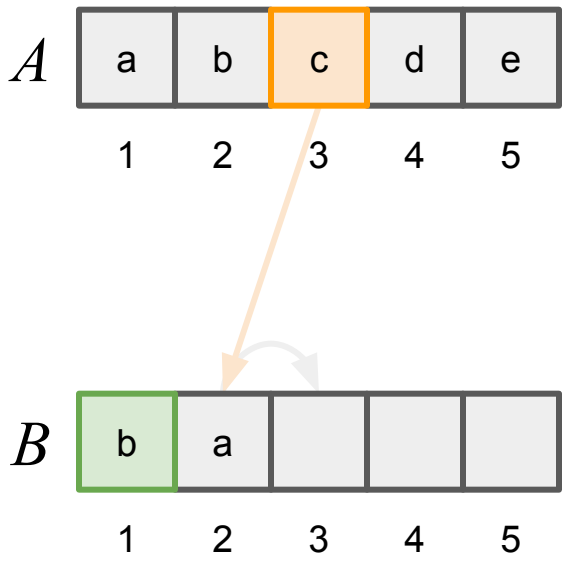
Initial step: We shall just copy first element over

Behaviour trace



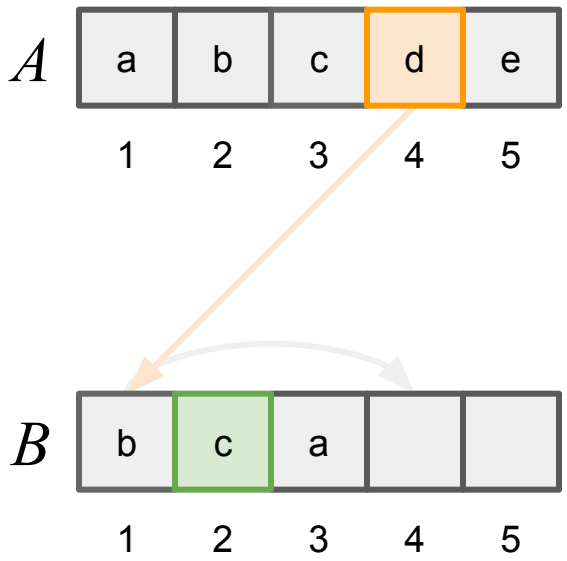
$i = 2$, random(1,1) returns 1 so $B[1]$ goes to $B[2]$ and $A[2]$ goes to $B[1]$..

Behaviour trace



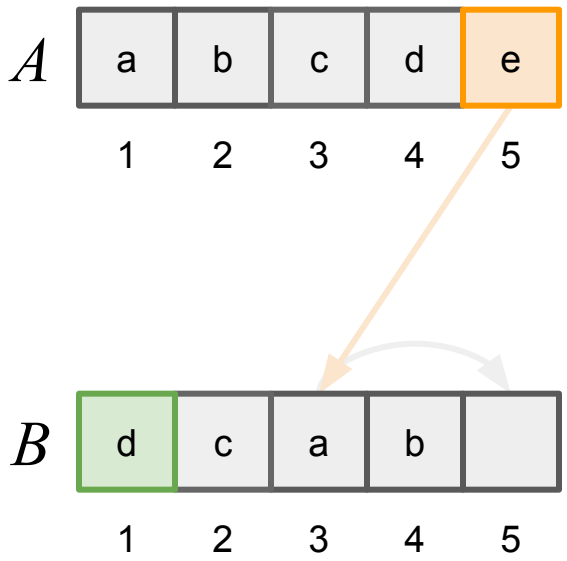
$i = 3$, random(1,2) returns 2 so $B[2]$ goes to $B[3]$ and $A[3]$ goes to $B[2]$..

Behaviour trace



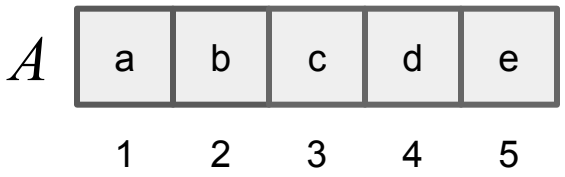
$i = 4$, random(1,3) returns 1 so $B[1]$ goes to $B[4]$ and $A[4]$ goes to $B[1]$..

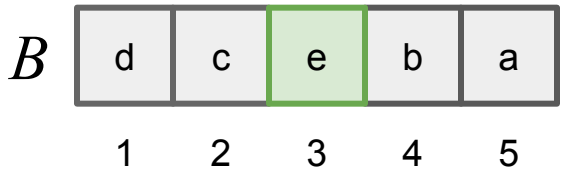
Behaviour trace



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

Behaviour trace





We are done!

Problem 2.f.

So which is the better permutation generating algorithm.

What's the moral of the story here?

Guiding question

Which is the better permutation generating algorithm?

Guiding question

Which is the better permutation generating algorithm?

Answer: The better algorithm here is the one which explicitly avoids fixed point permutations so as to ensure the expected number of students grading their own assignment is zero.

This isn't the algorithm that produces all permutations with equal probability.

Moral of the story

Realize the metrics of success is now different from the previous parts.

Before: producing fixed point permutations is necessary to ensure true uniform randomness.

Now: we explicitly don't want fixed point permutations but desires uniform randomness otherwise.

Once again, the moral of the story here is that it is the metrics we choose that drives the solution. An algorithm itself is neither good nor bad without the problem context it serves.

Problem 3

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 3.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^{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

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

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

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

Test yourself!

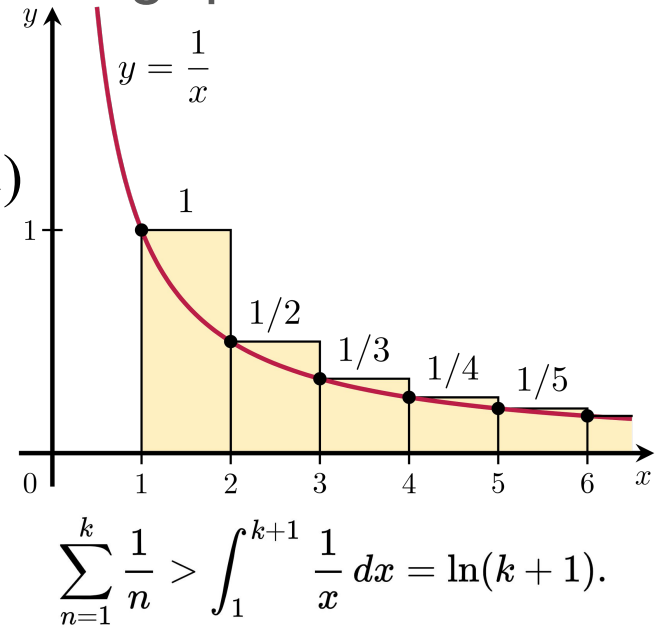
How often do you *expect* to swap an existing question for a new one?

Test yourself!

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

Answer: $1/1 + 1/2 + 1/i + \ \ldots\ + 1/n \approx \mathrm{O}(\log n)$

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



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 $\mathrm{O}(\log n)$ elements in the stream in expectation.

Problem 3.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^{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 :)