Credits: Slides originally by Jin Zhe
With added notes by Eldric

CS2040S

Recitation 3
AY24/25S2

My DNA

Question 1

Problem

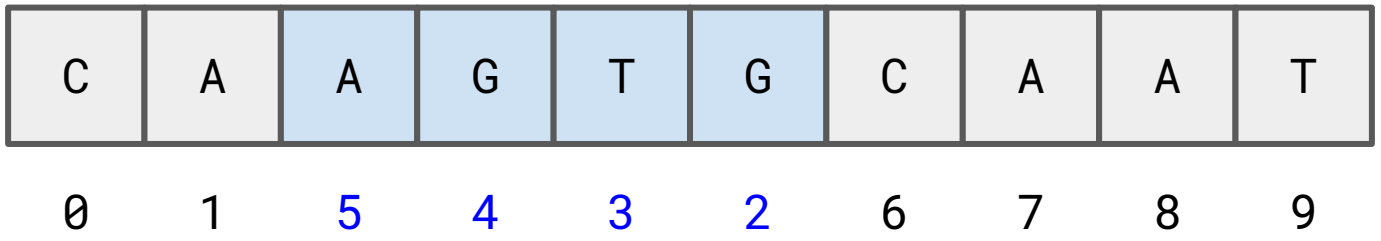
Input: String (sequence of characters) e.g. DNA sequence

Output: Ordered string

Constraint: Can only reverse a subsequence. Algorithm has to be in-place.

Example

After reversing [2,5]



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 data-manipulation operations are reversals. Everything has to be done in-place. I.e. you cannot simply count the ‘A’s and ‘T’s and rebuild the string!

Inspecting/examining the string is legal and free.

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

Suppose we wish to use QuickSort, which step do we need to modify and how? What would be its time complexity?

Guiding question

Suppose we wish to use QuickSort, which step do we need to modify and how? What would be its time complexity?

Answer: The partitioning step. Because we have have to facilitate mutual item swaps. Suppose we wish to $\text{Swap}(i, j)$, we can achieve that by doing $\text{Reverse}(i, j)$ followed by $\text{Reverse}(i+1, j-1)$.

This would mean partitioning step is now $O(n^2)$ which is bad..

Guiding question

Suppose we wish to use MergeSort, which step do we need to modify?

Guiding question

Suppose we wish to use MergeSort, which step do we need to modify?

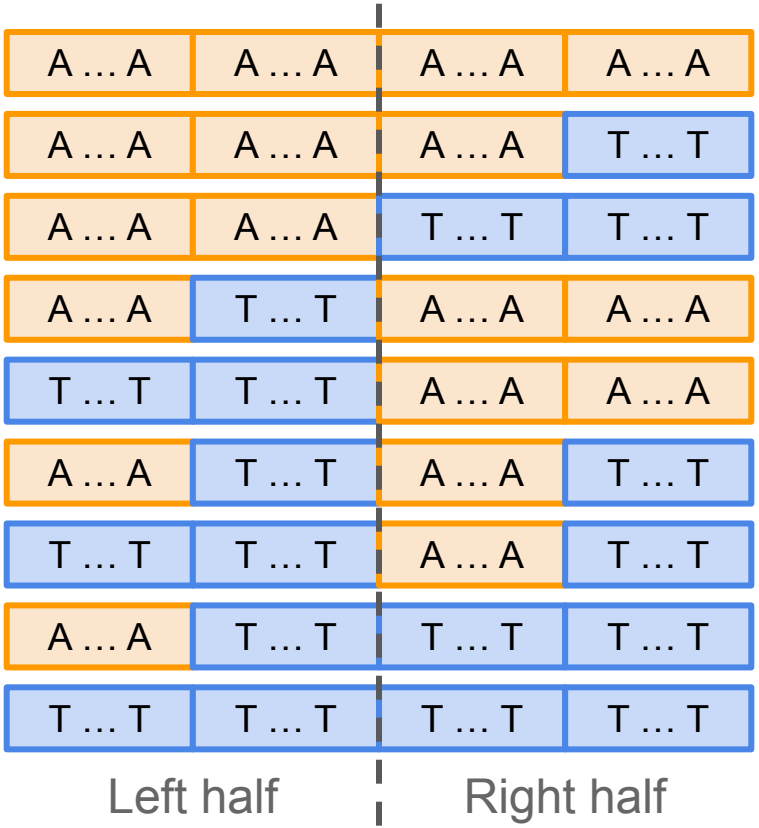
Answer: The Merge step. We need to somehow merge the left and right sorted halves in-place and only using subsequence reversals.

Guiding question

At the merge step, how many possible configurations (in terms of contiguous subsequences) do we have?

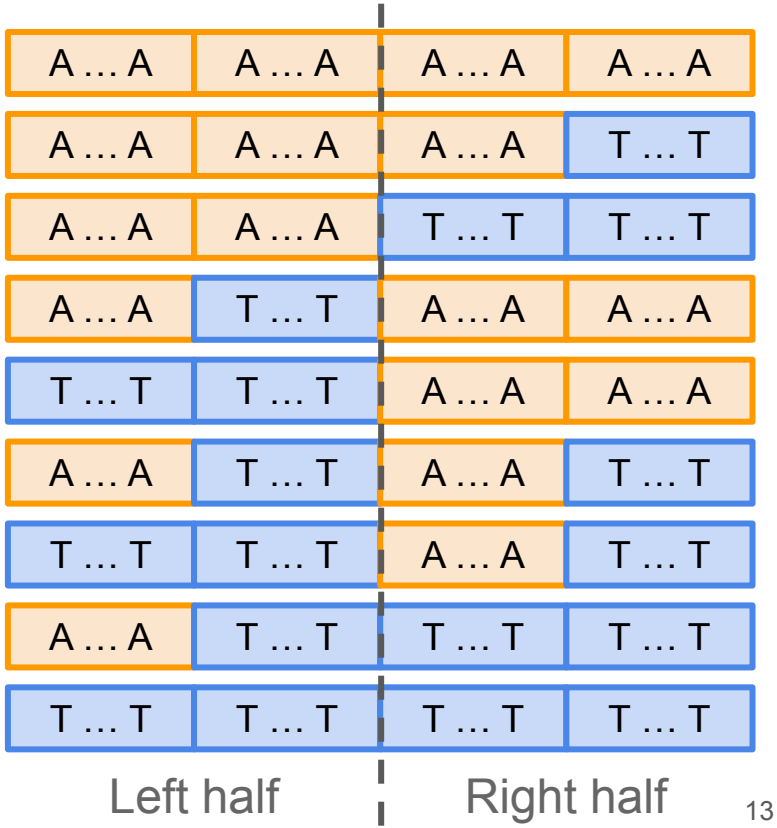
Answer: Realize LHS and RHS are themselves sorted. Each sorted side alone has 3 possible configurations: (1) A…AT…T, (2) A…A, (3) T…T. Therefore there are a total of $3 \times 3 = 9$ possible configurations at the merge step.

9 possible configurations



Guiding question

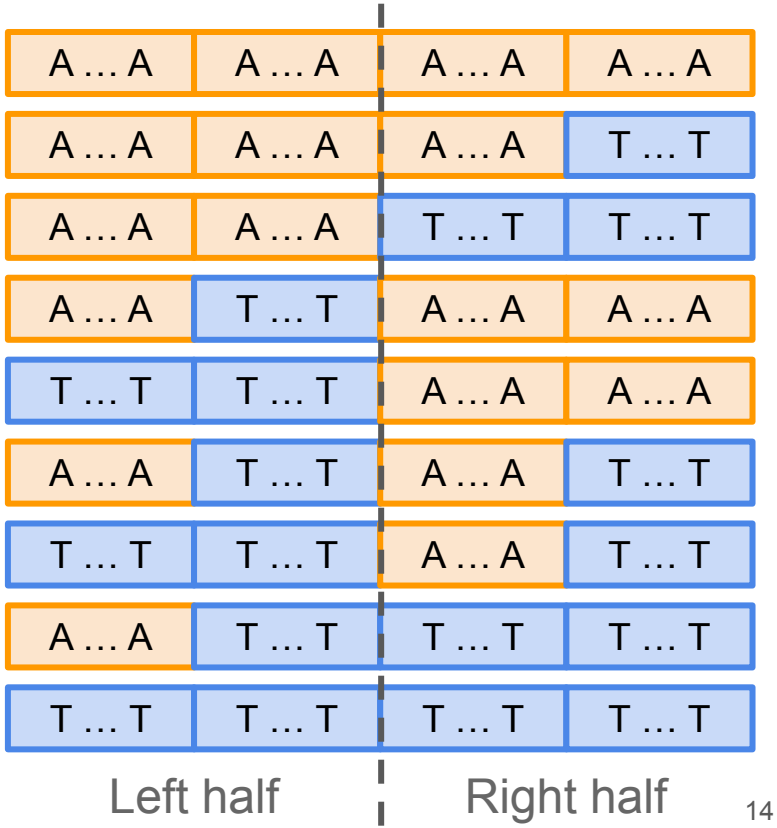
Given these configurations, how do we implement the modified merge step? Can you come up with a generalisable algorithm to do so? What is its complexity?



Guiding question

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

Answer: Reverse [first T in left half, last A in right half]. $O(n)$ time.



Guiding question

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

Guiding question

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

Answer: Since we only modified the merge step and its complexity is still $\mathrm{O}(n)$. The overall complexity of our modified MergeSort is still $\mathrm{O}(n \log n)$.

Guiding question

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

Guiding question

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

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

Problem 1.b.

Now, assume your string consists of arbitrary characters. Devise a QuickSort-like algorithm for sorting.

Hint: use the binary algorithm above to help you implement partition

Assume for now that each element in the string is unique, i.e., there are no duplicates.

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? Is this better than performing swaps using reversals?

Guiding question

Which step in QuickSort can we implement using our 1.a. solution with minimal modifications? Is this better than performing swaps using reversals?

Answer: Step 2 (partitioning step). We can transform it into a binary partitioning problem by treating each item greater than pivot as `A` and lesser than pivot as `T`. Thereafter, we can solve as per 1.a. Solution. This would achieve partitioning in $O(n \log n)$ which is faster than the $O(n^2)$ if we perform swaps using reversals.

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 QuickSort'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 elements in the string?

Can you still use the exact routine from the previous part? If not, what would you now do differently?

Hint: think the most extreme case for duplicate elements.

Guiding question

How might we achieve 3-way partitioning using a binary partitioning algorithm?

Guiding question

How might we achieve 3-way partitioning using a binary partitioning algorithm?

Answer: Partition twice!

First bipart to partition $< \text{pivot}$ and $\geq \text{pivot}$ out of all items,

Second bipart to partition $= \text{pivot}$ and $> \text{pivot}$ out of $\geq \text{pivot}$ items.

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 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 $1/n!$

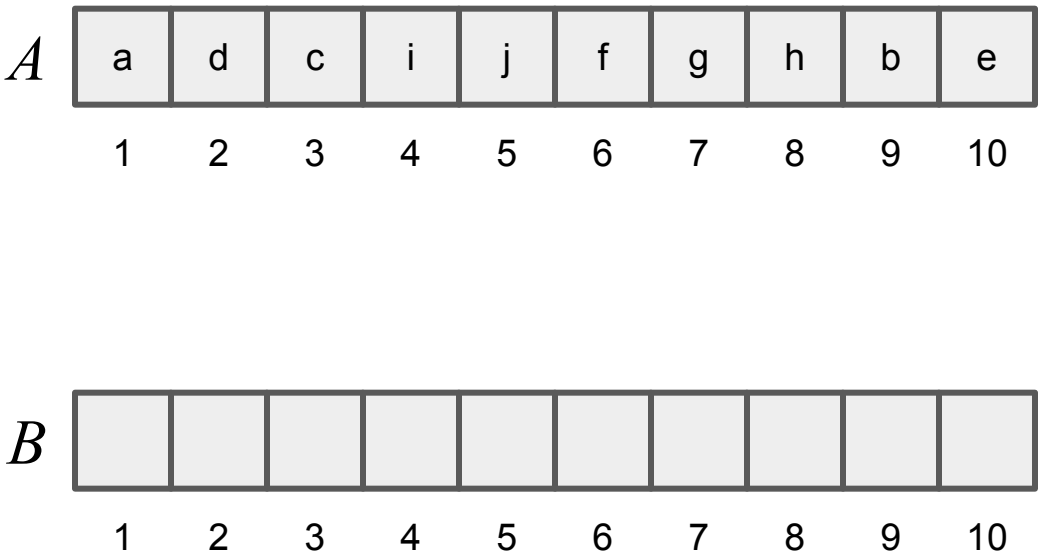
Realise this objective entails a hard-constraint.

Problem 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?

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, $n - i$)
    $B[i] = A[j]$
    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 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 = 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

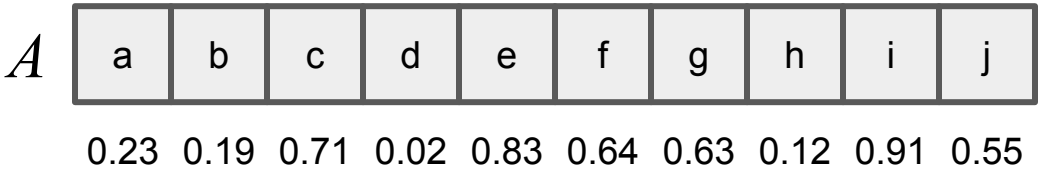
What is the time complexity of NaiveShuffle++?

Answer: $\mathrm{O}(n)$ time.

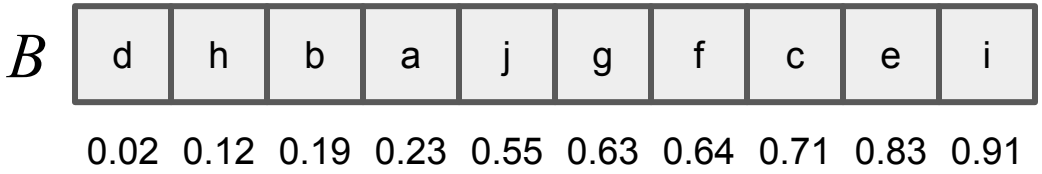
Kudos to the students who proposed this during class!

Sorting shuffle

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



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!

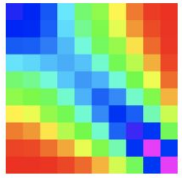
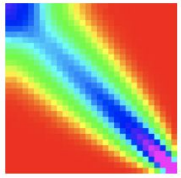
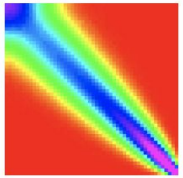
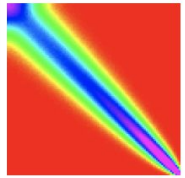
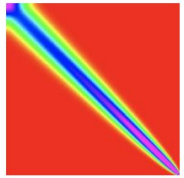
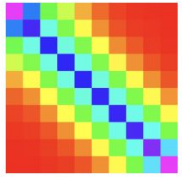
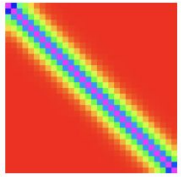
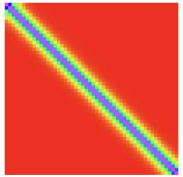
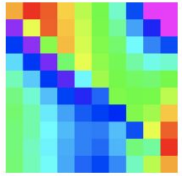
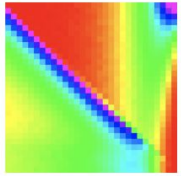
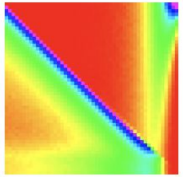
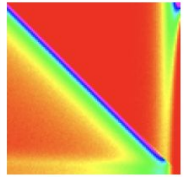
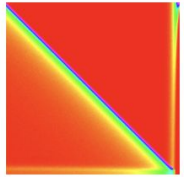
Did you know, this was an actual Javascript bug found in Windows 7! See here for more information.

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 $\geq ¼$ 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” or “litmus test” for any proposed permutation-generating algorithms?

Guiding question

What would be a simple “sanity check” or “litmus test” for any proposed permutation-generating algorithms?

Answer: We can check if it is even *possible* for the number of *algorithmic outcomes* to distribute over each permutation *uniformly*. I.e. each permutation having equal representation in the outcome space. Here we define an algorithmic outcome as a unique sequence of swaps undertaken by the algorithm.

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. 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.

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

Problem 2.d.

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 2.b.) would suffice.

Fisher-Yates/Knuth-shuffle

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

Compare and contrast

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

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

Invariants

Observe how the initial set of algorithms maintain an invariant like:

- First k items are a random permutation of k random items in xs

This is done by selecting a random item and extending the permuted region.
Effectively selection sort for permutations.

Invariants

Observe how the initial set of algorithms maintain an invariant like:

- First k items are a random permutation of **k random items in xs**

This is done by selecting a random item and extending the permuted region.
Effectively selection sort for permutations.

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

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

Spend a few moments to think about what this invariant could be

Invariants

Observe how the initial set of algorithms maintain an invariant like:

- First k items are a random permutation of **k random items in xs**

This is done by selecting a random item and extending the permuted region.
Effectively selection sort for permutations.

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

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

Spend a few moments to think about what this invariant could be

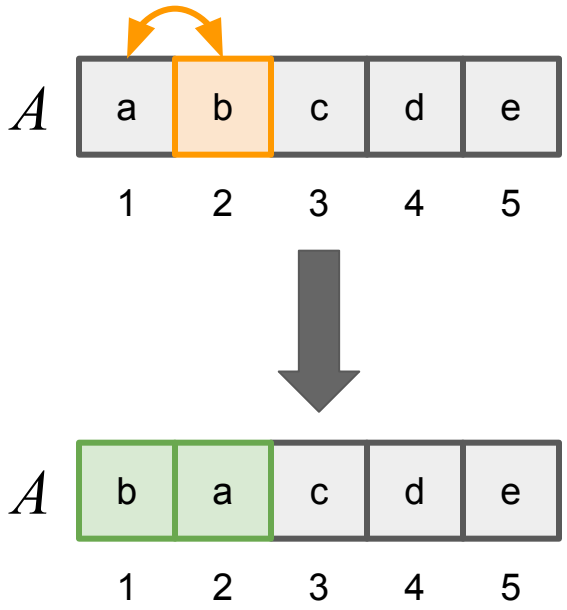
- First k items are a random permutation of **the first k items in xs**

Similar to insertion sort

Behaviour trace

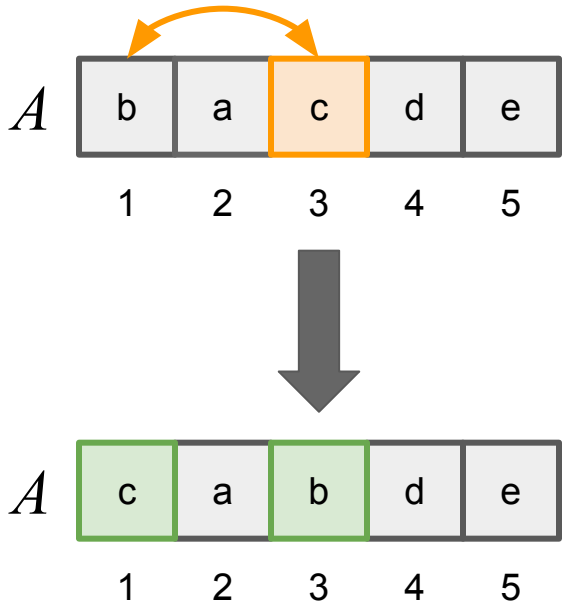
Treat [a] as the permuted region.

Extend it to size 2.



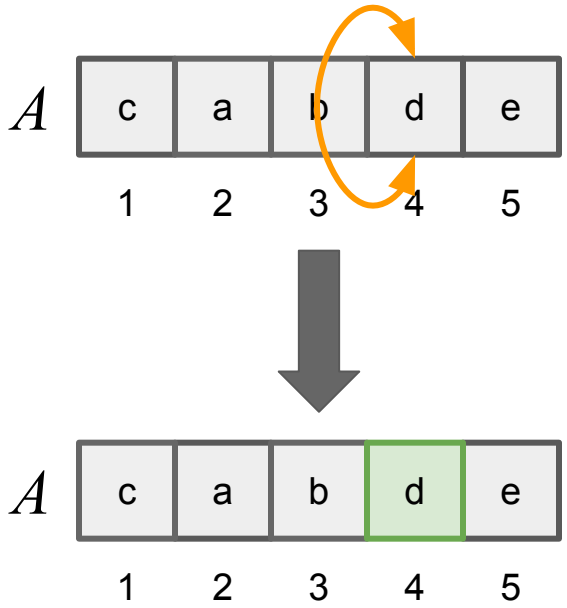
$i = 2$, random(1,2) returned 1, we swap index 2 with 1

Behaviour trace



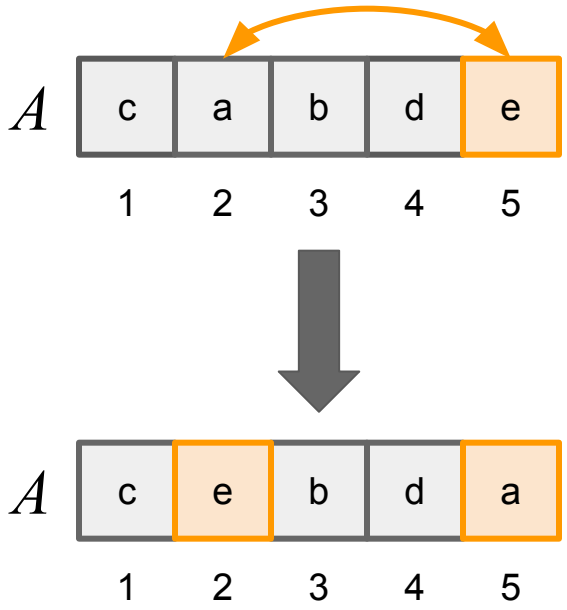
$i = 3$, random(1,3) returned 1, we swap index 3 with 1

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

Think inductively: Suppose the first 4 items were a random permutation, what to do with e?

e can be at any location (including the last) with probability 1/5

Conclude: Swap with random position.

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 760 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 $B[i]$

Optional Problems

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?

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 $n \times 1/n = 1$

Formally: Let Xi be the indicator random variable that the nth item stays in position
P(Xi = 1) = 1/n, E[Xi] = 1/n
E[X] = E[X1+...Xn] = E[X1] + … + E[Xn]
= n * 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.

Recall 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

How to modify?

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.

Recall 2.d.

Consider the Fisher-Yates / Knuth Shuffle algorithm:

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

Let B be a copy of A
for (i from 2 to n) **do**
    Choose r = random(1, i-1)
    Swap(B, i, r)
end

How to modify?

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.

Note: this algorithm not guaranteed to be fair, someone raised an objection but none of us can recall it

Method guaranteed to be fair.

If it fails, try again.

- Probability of success: ((n-1)/n) ^ n.
- Argue about the number of times required to succeed.
  - Expected tries: (n / (n-1)) ^ n
  - E.g. for n = 1000, requires expected 2.72 times.
    - Does this number look familiar?
    - lim n -> infinity: (n / (n-1)) ^ n ?
    - Can prove that this number is e = 2.718…
  - I.e. for large n, constant number of tries. Hence still O(n)

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

Define problem, then optimise to solve problem.

- Best algorithm is dependent on problem.