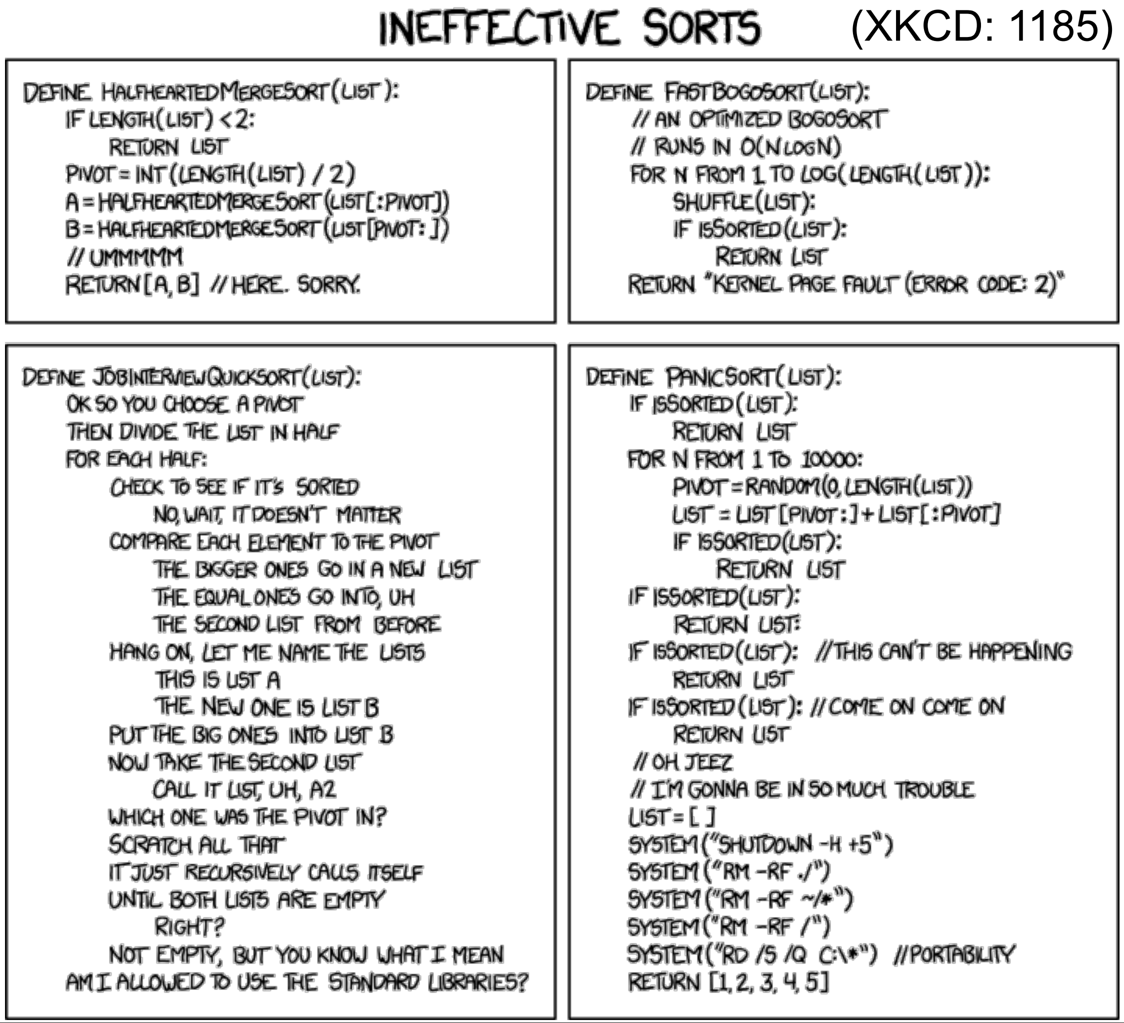
CS2040S
Data Structures and Algorithms

Welcome!



(XKCD: 1185)

Sorting, Part I

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

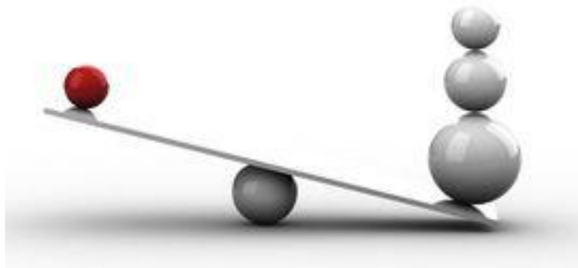
- Running time
- Space usage
- Stability

Sorting, Part II

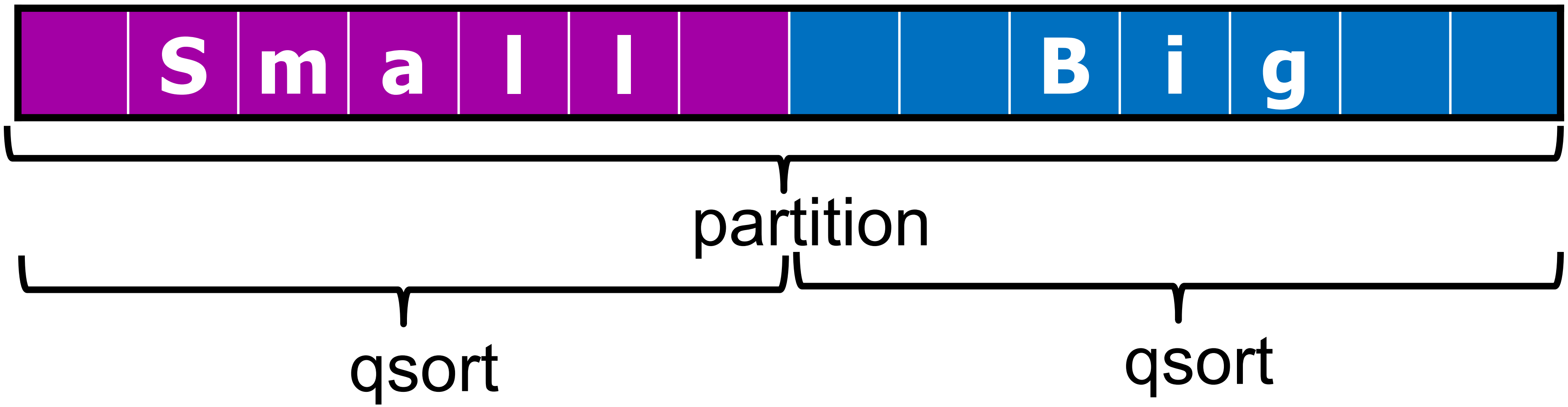
QuickSort

- Divide-and-Conquer
- Partitioning
- Duplicates
- Choosing a pivot

QuickSort



```
QuickSort(A[1..n], n)
    if (n==1) then return;
    else
        p = partition(A[1..n], n)
        x = QuickSort(A[1..p-1], p-1)
        y = QuickSort(A[p+1..n], n-p)
```



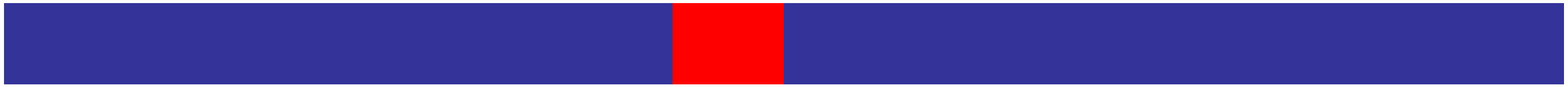
QuickSort Choices

How to choose a pivot?

1. Choose the first element of the array.

2. Choose the last element of the array.

3. Choose the middle element in the array.



4. Choose the median element in the array.

5. Choose a random element in the array.

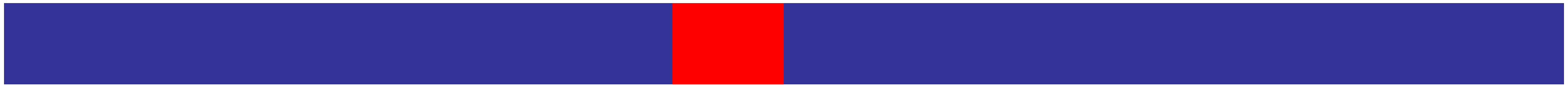
QuickSort Choices

How to choose a pivot?

1. Choose the first element of the array.

2. Choose the last element of the array.

3. Choose the middle element in the array.



Worst-case time: $\Theta(n^2)$

QuickSort Choices

How to choose a pivot?

Worst-case (expected) time: Θ(n log n)

4. Choose the median element in the array.

5. Choose a random element in the array.

QuickSort Choices

How to choose a pivot?

Worst-case (expected) time: $\Theta(n \log n)$

Simplest option: choose randomly!

4. Choose the median element in the array.

5. Choose a random element in the array.

QuickSort Choices:

How to partition?

1. Copy elements to new array.
2. In-place partitioning.

What about duplicate keys?

1. Ignore. They don't exist.
2. Two-pass partitioning.
3. One-pass partitioning.

QuickSort Choices:

Base case?

1. Recurse all the way to single-element arrays.
2. Switch to InsertionSort for small arrays.
3. Halt recursion early, leaving small arrays unsorted. Then perform InsertionSort on entire array.

QuickSort Stability

QuickSort is stable if partioning is stable.

1. In-place partitioning is not *stable*.
2. Extra-memory allows QuickSort to be stable.

QuickSort Analysis:

How to show good performance?

1. If pivot is median: simple recurrence analysis.
2. If pivot is random: *most* of the time, we get a good split.

Use coin flipping analysis and Paranoid variant to show that performance is good.

Summary

QuickSort:

- Algorithm basics: divide-and-conquer
- How to partition an array in O(n) time.
- How to choose a good pivot.
- Paranoid QuickSort.
- Randomized analysis.

Today: Sorting, Part III

Selection and Order Statistics

- QuickSelect

Random Shuffles

- Sorting Shuffle
- Fisher-Yates-Knuth Shuffle

Order Statistics

Find k^{th} smallest element in an *unsorted* array:

$\mathbf{x_{10}}$	$\mathbf{x_2}$	$\mathbf{x_4}$	$\mathbf{x_1}$	$\mathbf{x_5}$	$\mathbf{x_3}$	$\mathbf{x_7}$	$\mathbf{x_8}$	$\mathbf{x_9}$	$\mathbf{x_6}$

E.g.: Find the median ($k = n/2$)

Find the 7th element ($k = 7$)

Order Statistics

Find k^{th} smallest element in an *unsorted* array:

x_{10}	x_2	x_4	x_1	x_5	x_3	x_7	x_8	x_9	x_6

Option 1:

- Sort the array.
- Count to element number k.

Running time: O(n log n)

Order Statistics

Find k^{th} smallest element in an *unsorted* array:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}

Option 1:

- Sort the array.
- Count to element number k.

Running time: O(n log n)

Order Statistics

Find k^{th} smallest element in an *unsorted* array:

x_{10}	x_2	x_4	x_1	x_5	x_3	x_7	x_8	x_9	x_6

Option 2:

– Only do the minimum amount of sorting necessary

Order Statistics

Key Idea: partition the array

x_2	x_4	x_1	x_3	x_5	x_7	x_8	x_9	x_6	x_{10}

Now continue searching in the correct half.

E.g.: Partition around x_5 and recursively search for x_3 in left half.

Order Statistics

Example: search for 5th element

9	**22**	**13**	**17**	**5**	**3**	**100**	**6**	**19**	**8**

Order Statistics

Example: search for 5th element

9	**22**	**13**	**17**	**5**	**3**	**100**	**6**	**19**	**8**

Partition around random pivot: 17

9	**8**	**13**	**5**	**3**	**6**	**17**	**100**	**19**	**22**
1	2	3	4	5	6	**7**	**8**	**9**	**10**

Order Statistics

Example: search for 5th element

9	**8**	**13**	**5**	**3**	**6**	**17**	**100**	**19**	**22**
1	2	3	4	5	6	**7**	**8**	**9**	**10**

Search for 5th element in left half.

9	**8**	**13**	**5**	**3**	**6**				
1	2	3	4	5	6				

Order Statistics

Example: search for 5th element

9	**8**	**13**	**5**	**3**	**6**				

Partition around random pivot: 8

6	**3**	**5**	**8**	**13**	**9**				
1	2	3	**4**	5	6				

Order Statistics

Example: search for 5[th] element

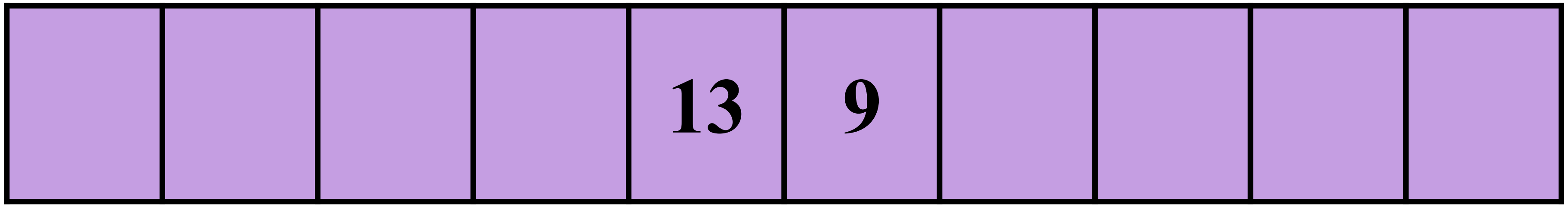
9	8	13	5	3	6				

Search for: 5 – 4 = 1 in right half

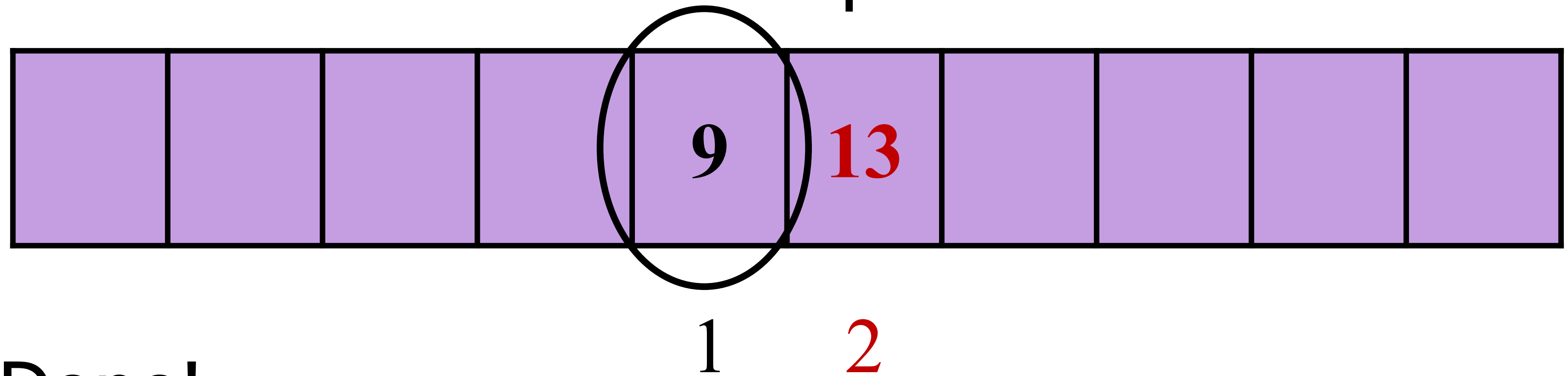
6	3	5	8	13	9				
1	2	3	4	5	6				

Order Statistics

Search for: 5 – 4 = 1 in right half



Partition around random pivot: 13



Done!

Finding the k^{th} smallest element

```
Select(A[1..n], n, k)
    if (n == 1) then return A[1];
    else   Choose random pivot index pIndex.
           p = partition(A[1..n], n, pIndex)
           if (k == p) then return A[p];
           else if (k < p) then
                  return Select(A[1..p–1], k)
           else if (k > p) then
                  return Select(A[p+1], k – p)
```

Order Statistics

Recursing right and left are not exactly the same.

Example: search for 5^{th} element

9	22	13	17	5	3	100	6	19	8

Partition around random pivot: 17

9	8	13	5	3	6	17	100	19	22
1	2	3	4	5	6	**7**	**8**	**9**	**10**

Search for 5^{th} element on the left.

Order Statistics

Recursing right and left are not exactly the same.

Example: search for 8th element

9	**22**	**13**	**17**	**5**	**3**	**100**	**6**	**19**	**8**

Partition around random pivot: 8

5	**6**	**3**	**8**	**17**	**13**	**100**	**22**	**19**	**9**
1	2	3	4	5	6	7	**8**	**9**	**10**

Search for 4th element on the right.

Order Statistics

Recursing right and left are not exactly the same.

Example: search for 4th element

9	**22**	**13**	**17**	**5**	**3**	**100**	**6**	**19**	**8**

Partition around random pivot: 8

5	**6**	**3**	**8**	**17**	**13**	**100**	**22**	**19**	**9**
1	2	3	4	5	6	**7**	**8**	**9**	**10**

Return 8.

Finding the k^{th} smallest element

```
Select(A[1..n], n, k)
    if (n == 1) then return A[1];
    else  Choose random pivot index pIndex.
          p = partition(A[1..n], n, pIndex)
          if (k == p) then return A[p];
          else if (k < p) then
                return Select(A[1..p–1], k)
          else if (k > p) then
                return Select(A[p+1], k – p)
```

Finding the k^{th} smallest element

Key point:

- Only recurse *once*!

- Why not recurse twice?
  - Does not help---the correct element is on one side.
  - You do not need to sort both sides!
  - Makes it run a lot faster.

Analysis

Paranoid-Select:

Repeatedly partition until at least $n/10$ in each half of the partition.

```
repeat
        p = partition(A[1..n], n, pIndex)
until (p > n/10) and (p < 9n/10)
```

Analysis

Paranoid-Select:

Repeatedly partition until at least $n/10$ in each half of the partition.

Recurrence:

$$\mathbf{E}[T(n)] \leq \mathbf{E}[T(9n/10)] + \mathbf{E}[\#\text{ partitions}](n)$$

cost of partitioning

Analysis

Paranoid-Select:

Repeatedly partition until at least $n/10$ in each half of the partition.

Recurrence:

$$\mathbf{E}[\mathrm{T}(\mathrm{n})] \le \mathbf{E}[\mathrm{T}(9\mathrm{n}/10)] + \mathbf{E}[\#\ \mathrm{partitions}](n)$$

$$\le \mathbf{E}[\mathrm{T}(9\mathrm{n}/10)] + 2n$$

Analysis

Paranoid-Select:

Repeatedly partition until at least $n/10$ in each half of the partition.

Recurrence:

$$\mathbf{E}[\mathrm{T(n)}] \leq \mathbf{E}[\#\ \mathrm{partitions}](n) + \mathbf{E}[\mathrm{T(9n/10)}]$$

$$\leq 2n+\mathbf{E}[\mathrm{T(9n/10)}]$$

$$\leq 2n+2n\ (9/10)+(9/10)\ \mathbf{E}[\mathrm{T(9n/10)}]$$

$$\leq 2n+2n\ (9/10)+2n\ (9/10)^2+\ldots\ldots$$

Analysis

Paranoid-Select:

Repeatedly partition until at least $n/10$ in each half of the partition.

Recurrence:

$$\mathbf{E}[\mathrm{T(n)}] \leq \mathbf{E}[\mathrm{T(9n/10)}] + \mathbf{E}[\#\ \mathrm{partitions}](n)$$
$$\leq \mathbf{E}[\mathrm{T(9n/10)}] + 2n$$
$$\leq \mathrm{O}(n)$$

Recurrence: $T(n) = T(n/2) + O(n)$

Other sorting related problems

Uniqueness testing

- Input:
  - Array A
- Output:
  - Does array A contain any duplicate items? YES/NO?

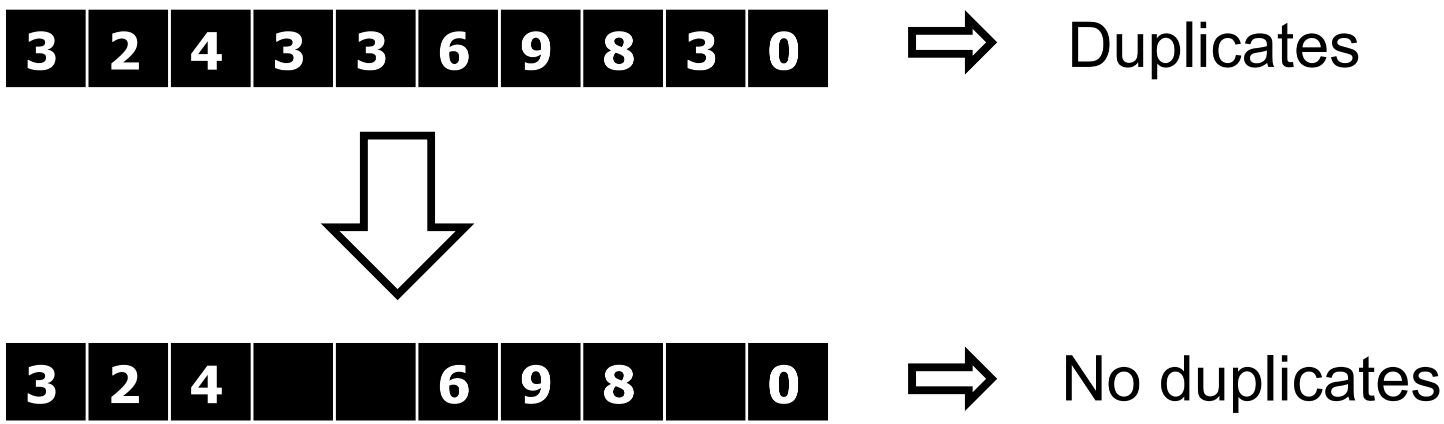
7	2	4	3	5	6	9	8	1	0	⇨	No duplicates

3	2	4	3	3	6	9	8	3	0	⇨	Duplicates

Other sorting related problems

Deleting duplicates

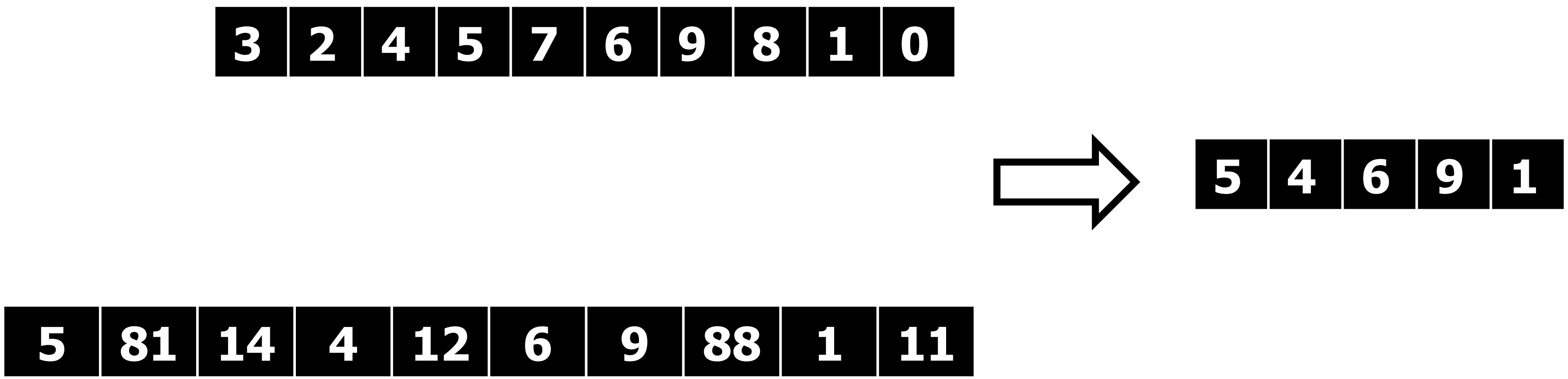
- Input:
  - Array A
- Output:
  - Array A with all the duplicates removed, same order.



Other sorting related problems

Set intersection:

- Input:
  - Array A, B
- Output:
  - Array C containing all items in both A and B.



Other sorting related problems

Target pair:

- Input:
  - Array A, target
- Output:
  - Two elements (x,y) in A where (x+y) = target.

5	81	14	4	12	6	9	88	1	11

⇨ (4, 9)

target = 13?

Summary

QuickSort: O(n log n)

- Partitioning an array
- Deterministic QuickSort
- Paranoid Quicksort

Order Statistics: O(n)

- Finding the k^{th} smallest element in an array.
- Key idea: partition
- Paranoid Select

Today: Sorting, Part III

Selection and Order Statistics

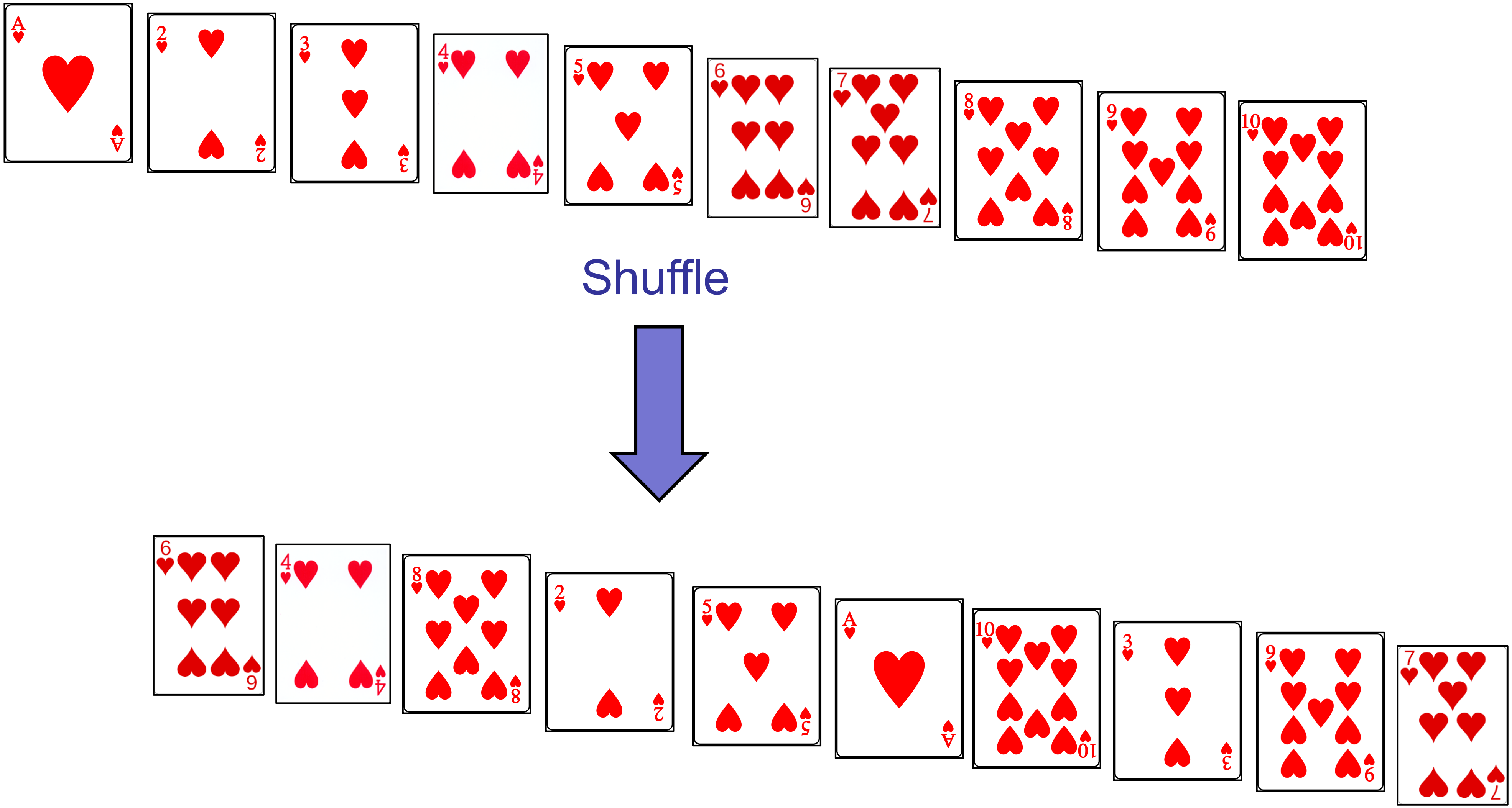
- QuickSelect

Random permutations

- Sorting Shuffle
- Knuth Shuffle

Generating a random permutation

or: How to shuffle a deck of cards.



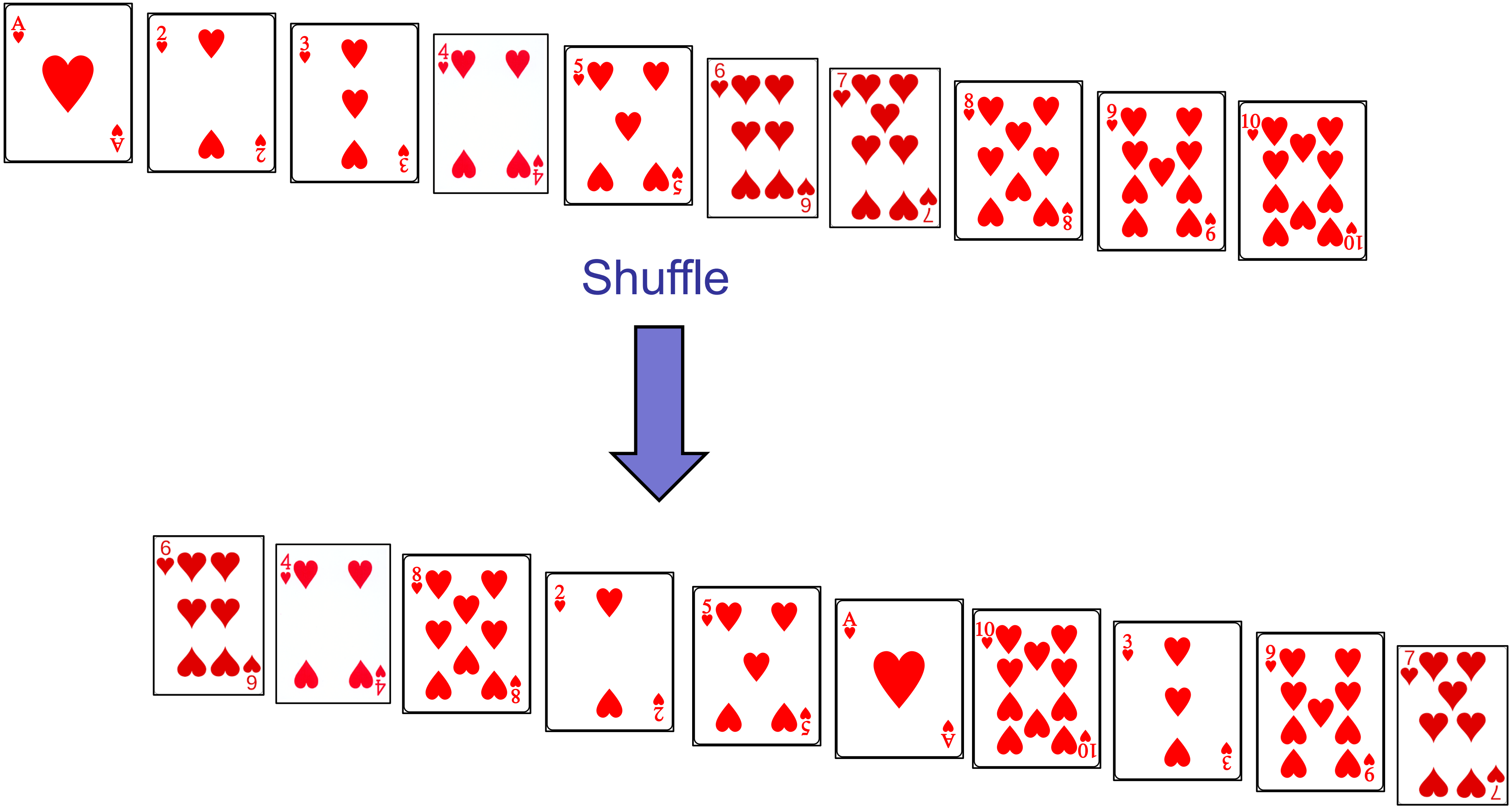
Random permutations

Why?

- Randomize data.
- Anonymize data.
- Generate fair schedules.
- Test average case performance.
- Quicksort:
    1. Permute array at random.
    2. Run QuickSort, choosing pivot to be first element.

Generating a random permutation

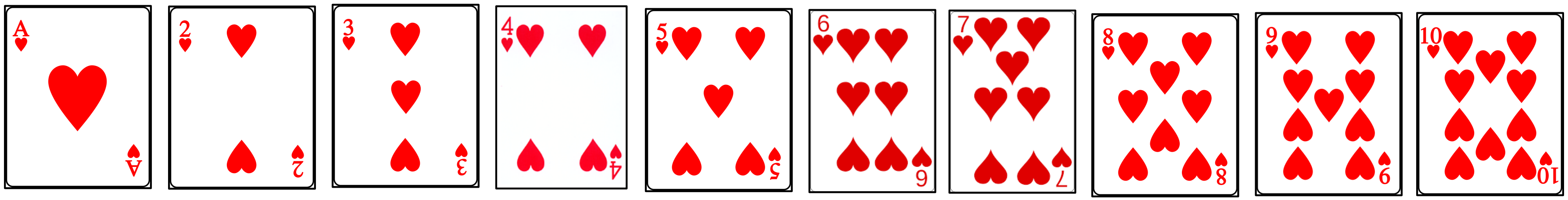
Goal: each permutation is equally likely.



Generating a random permutation

Algorithm: Sorting Shuffle

Step 1: Choose a random real number between [0,1] for each.

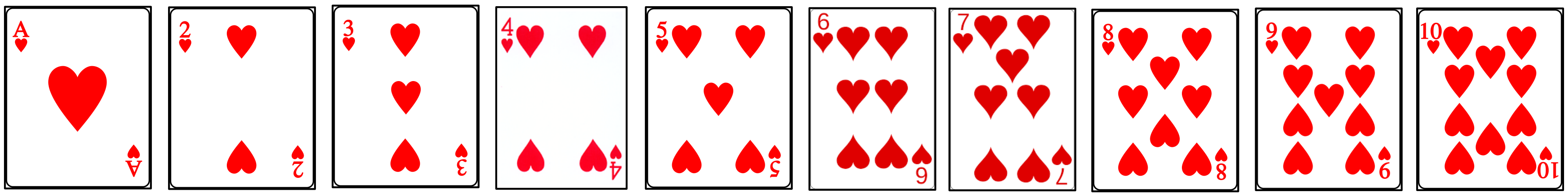


0.23 0.19 0.71 0.02 0.83 0.64 0.63 0.12 0.91 0.55

Generating a random permutation

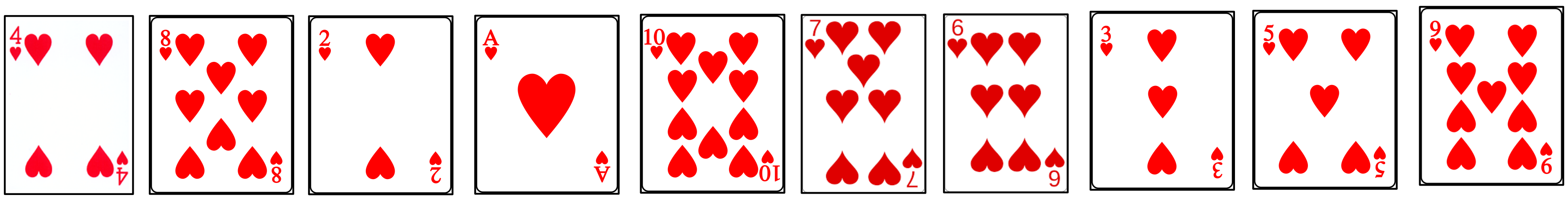
Algorithm: Sorting Shuffle

Step 1: Choose a random real number between [0,1] for each.



0.23 0.19 0.71 0.02 0.83 0.64 0.63 0.12 0.91 0.55

Step 2: Sort.



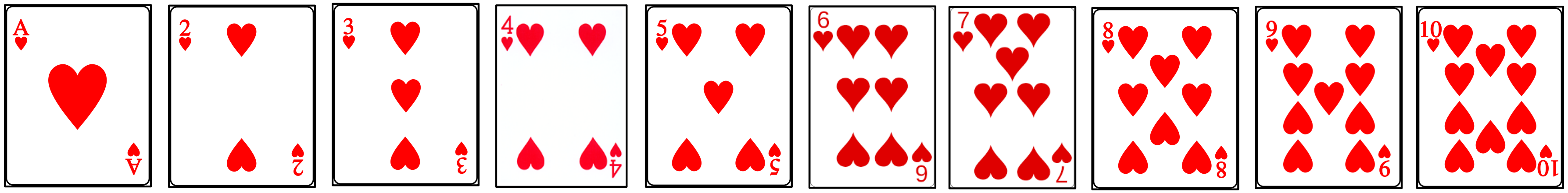
0.02 0.12 0.19 0.23 0.55 0.63 0.64 0.71 0.83 0.91

Generating a random permutation

Algorithm: Sorting Shuffle

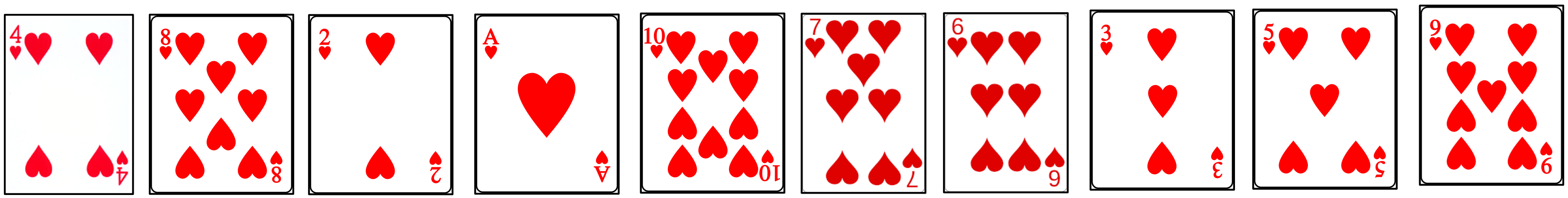
O(n log n)

Step 1: Choose a random real number between [0,1] for each.



0.23 0.19 0.71 0.02 0.83 0.64 0.63 0.12 0.91 0.55

Step 2: Sort.

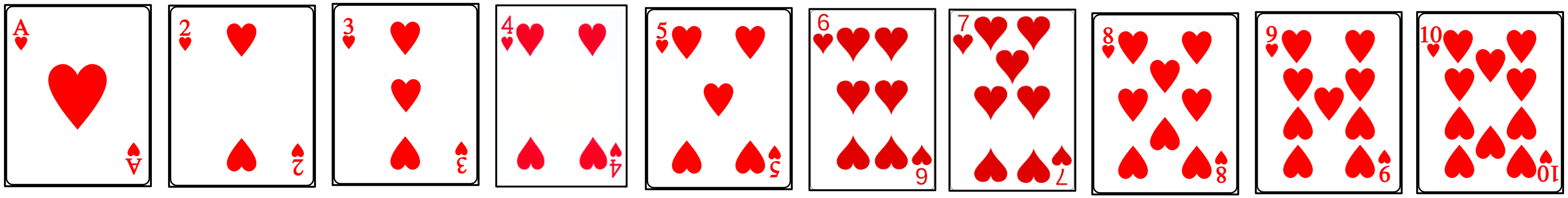


0.02 0.12 0.19 0.23 0.55 0.63 0.64 0.71 0.83 0.91

Generating a random permutation

Algorithm: Sorting Shuffle

Step 1: Choose a random real number between [0,1] for each.



0.23 0.19 0.71 0.02 0.71 0.71 0.63 0.12 0.91 0.55

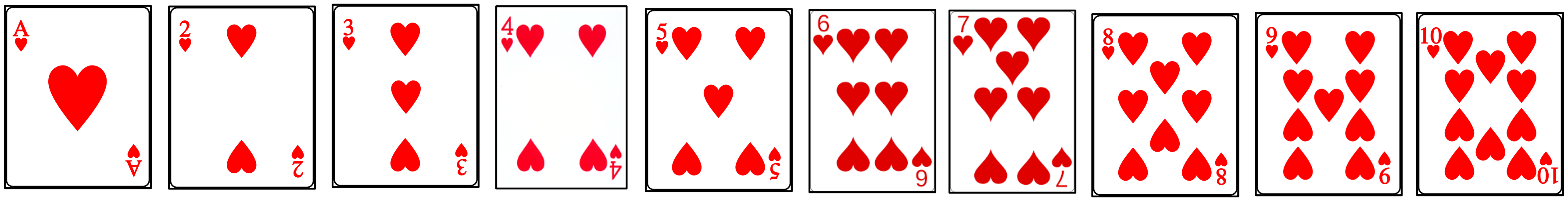
What if there are duplicate values chosen?

(Remember, computers have finite precision!)

Generating a random permutation

Algorithm: Sorting Shuffle

Step 1: Choose a random real number between [0,1] for each.



0.23 0.19 0.71 0.02 0.71 0.71 0.63 0.12 0.91 0.55

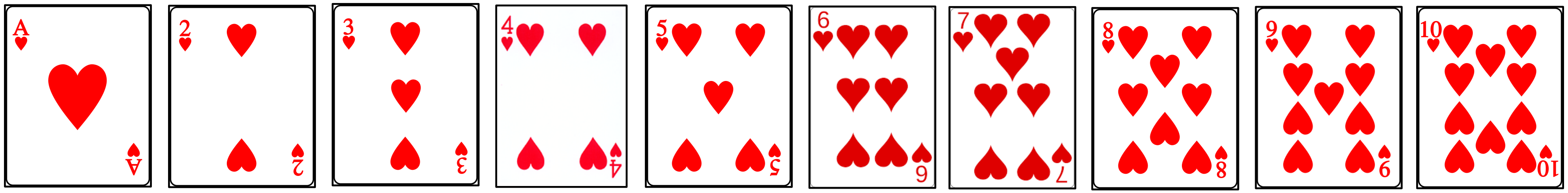
What if there are duplicate values chosen?

1. Re-run the Sorting Shuffle again.
2. Choose new random real numbers to break ties.

Generating a random permutation

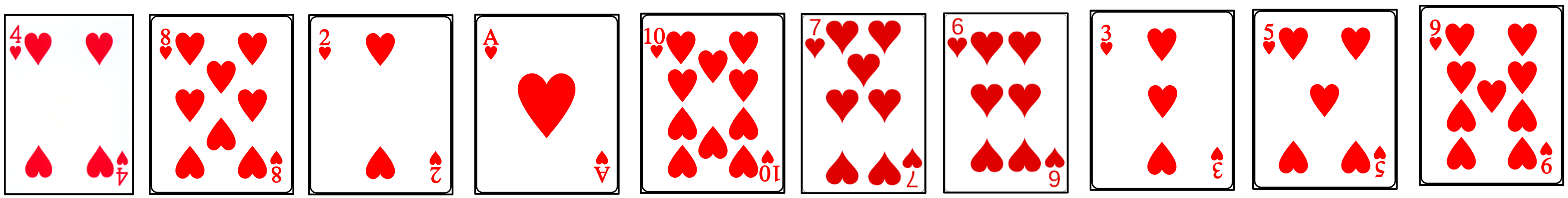
Algorithm: Sorting Shuffle

Step 1: Choose a random real number between [0,1] for each.



0.23 0.19 0.71 0.02 0.83 0.64 0.63 0.12 0.91 0.55

Step 2: Sort.



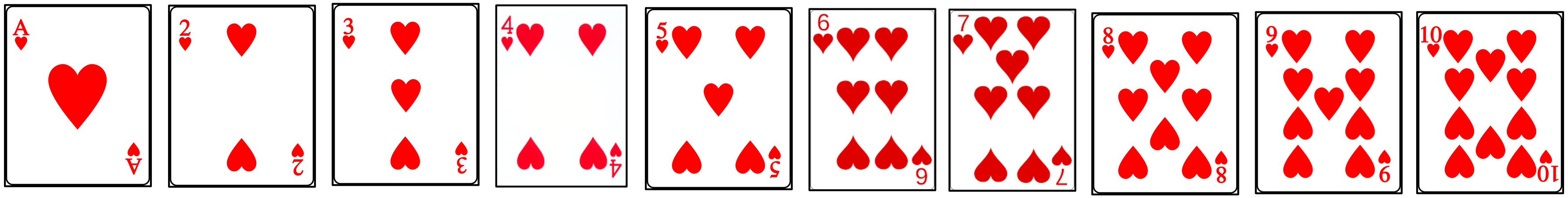
0.02 0.12 0.19 0.23 0.55 0.63 0.64 0.71 0.83 0.91

Step 3: If any duplicate values, repeat.

Generating a random permutation

Shortcut: Java 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;
}
```

Seems clever, right?

Generating a random permutation

Buggy 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;
}
```

Problem 1: compareTo must always return the same answer and must be transitive.

Generating a random permutation

Buggy 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;
}
```

Problem 2: Does not yield a random permutation!

Generating a random permutation

Claim:

For InsertionSort, the probability that A[1] = 1 is ≥ 1/4.

(If it were a correct shuffle, Pr(A[1] = 1) = 1/n.)

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

Challenge: Prove the above claim!

Generating a random permutation

Buggy 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;
}
```

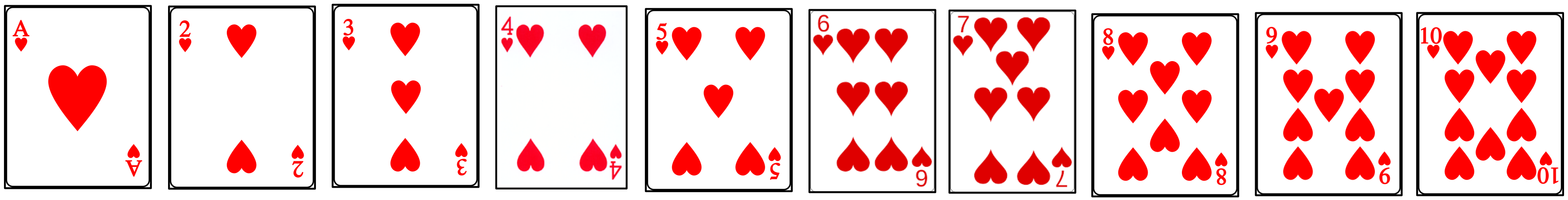
Real bug! This was a bug found in Windows 7.

See: http://www.robweir.com/blog/2010/02/microsoft-random-browser-ballot.html

Generating a random permutation

Algorithm: Knuth Shuffle [Fisher/Yates]

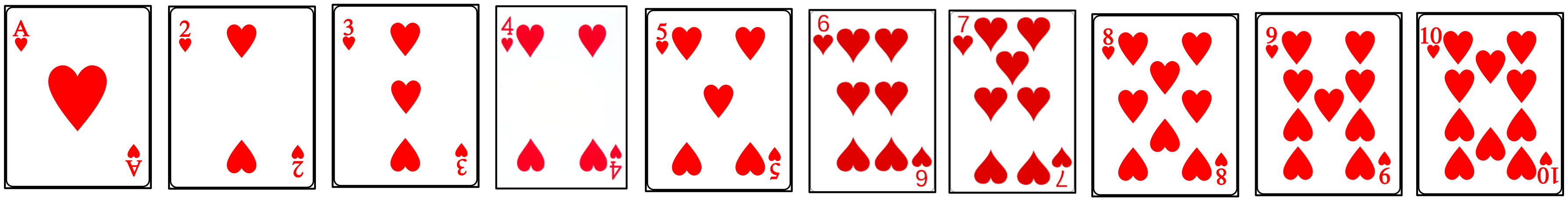
Idea: Iterate through array, creating a random prefix.



Generating a random permutation

Algorithm: Knuth Shuffle [Fisher/Yates]

Idea: Iterate through array, creating a random prefix.

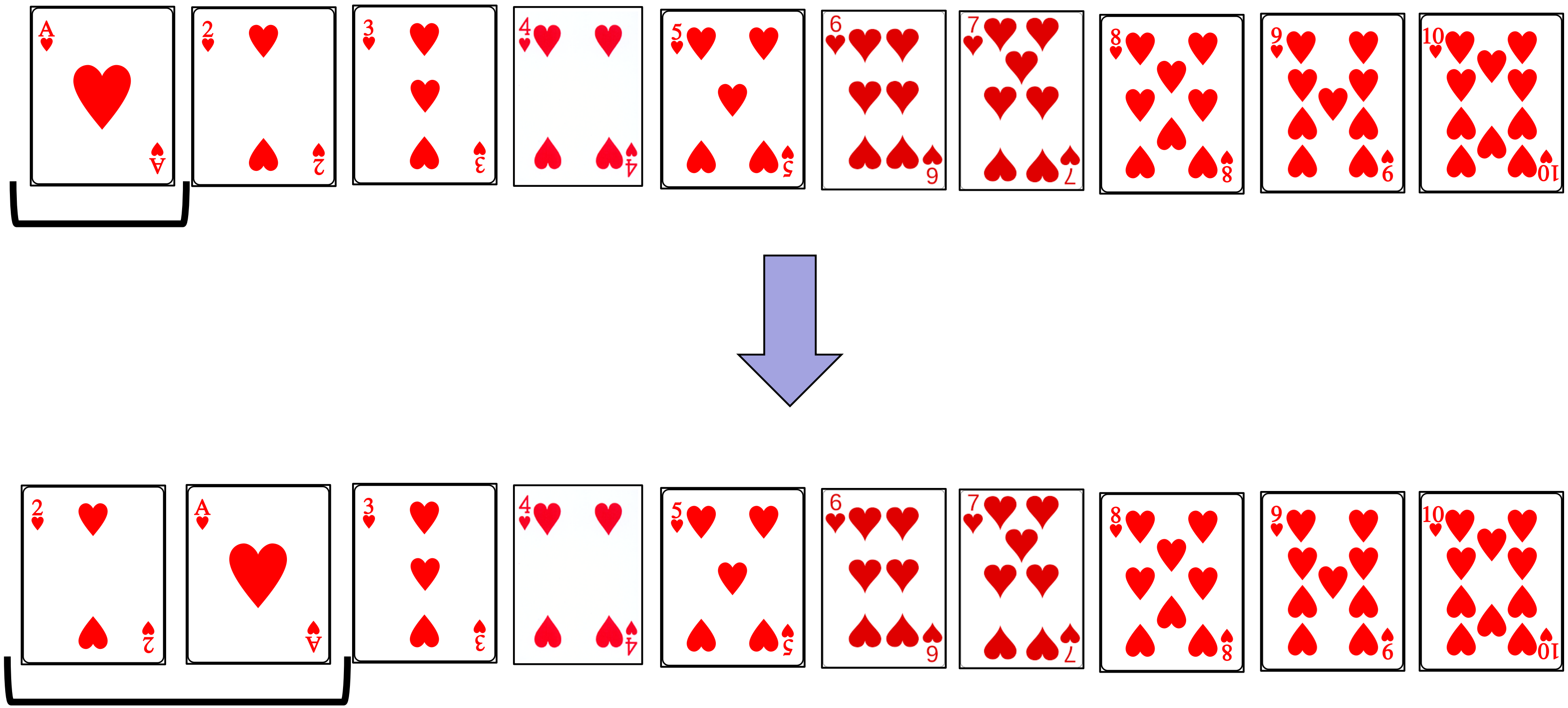


```
KnuthShuffle(A[1..n])
    for i = 2 to n do
        r = random(1, i)
        swap(A, i, r)
```

Generating a random permutation

Algorithm: Knuth Shuffle [Fisher/Yates]

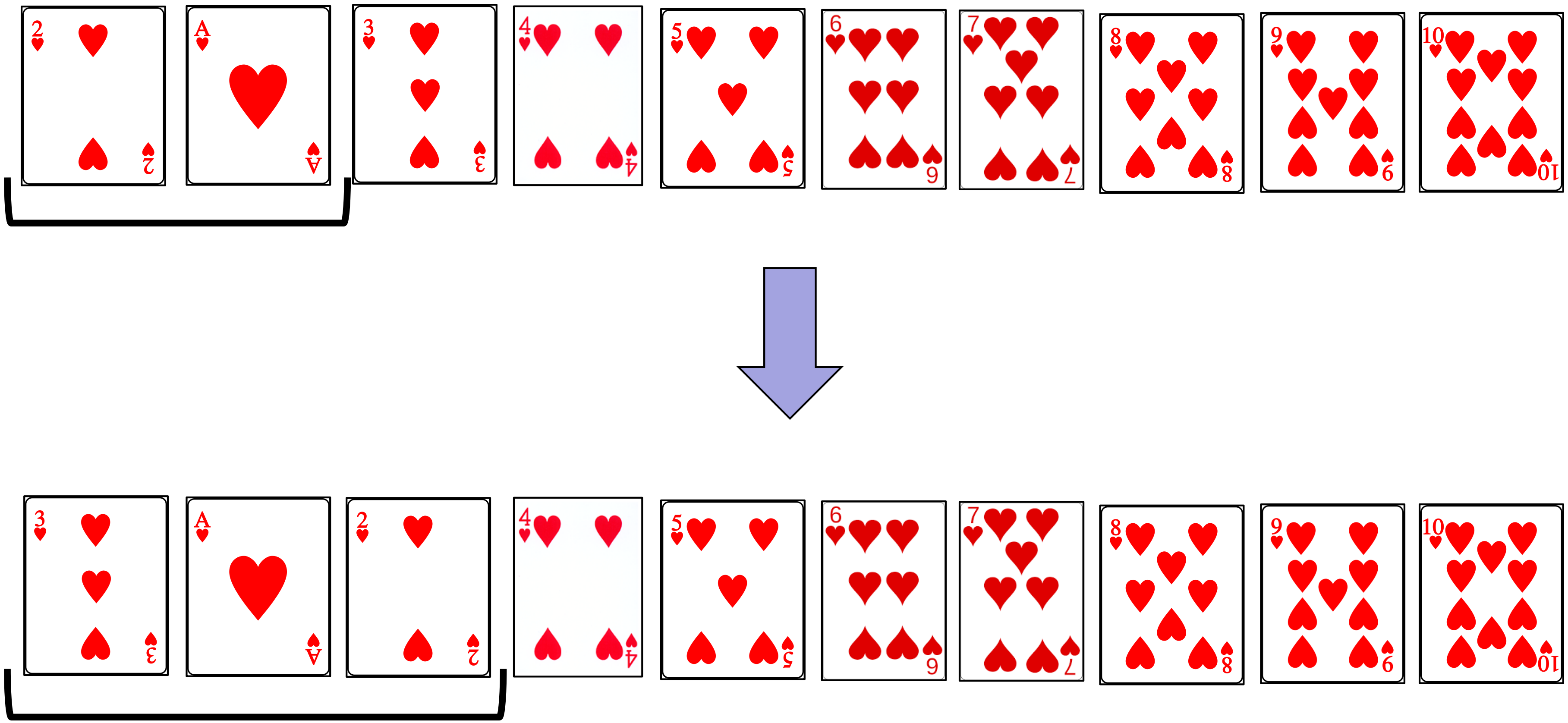
Example: i = 2, r =1



Generating a random permutation

Algorithm: Knuth Shuffle [Fisher/Yates]

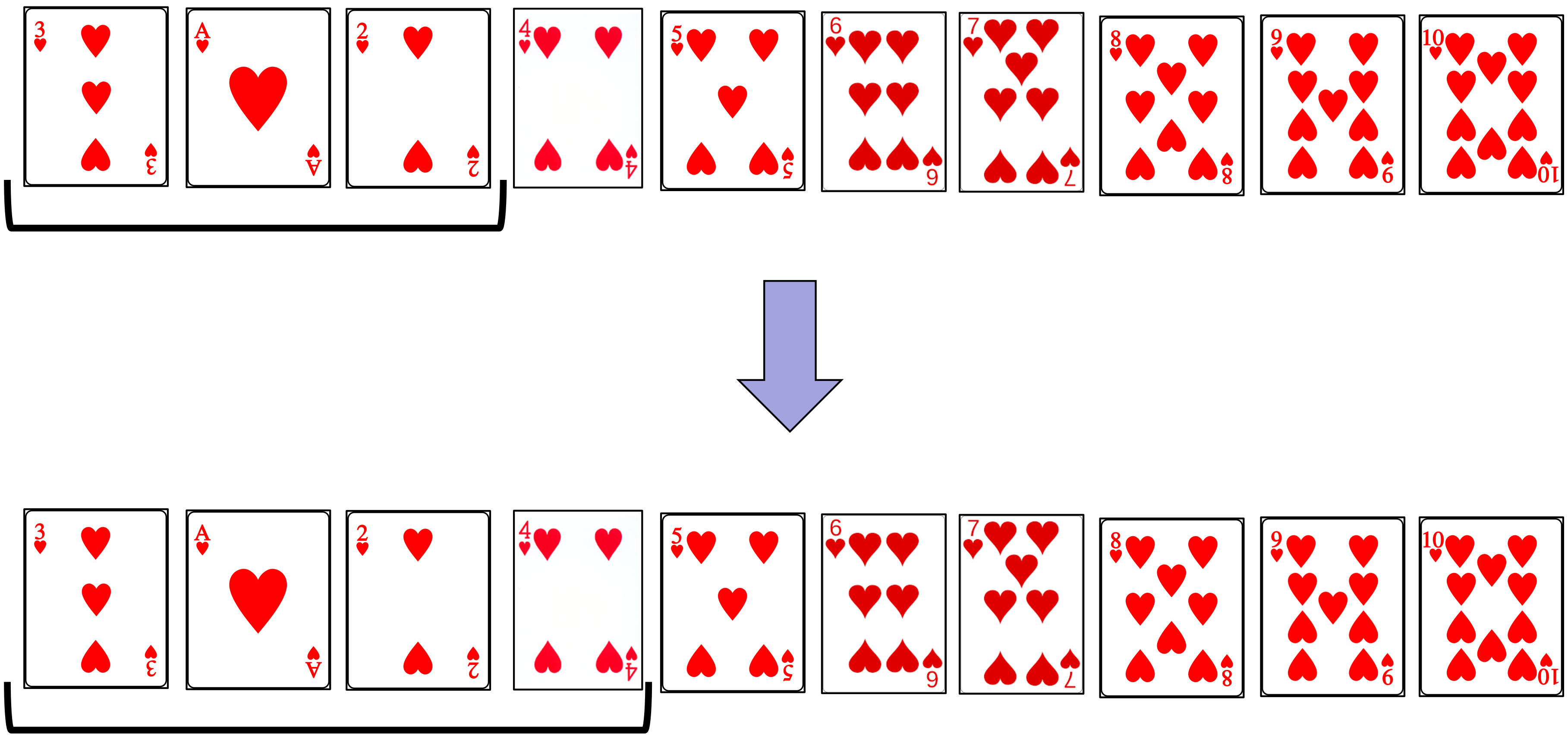
Example: i = 3, r =1



Generating a random permutation

Algorithm: Knuth Shuffle [Fisher/Yates]

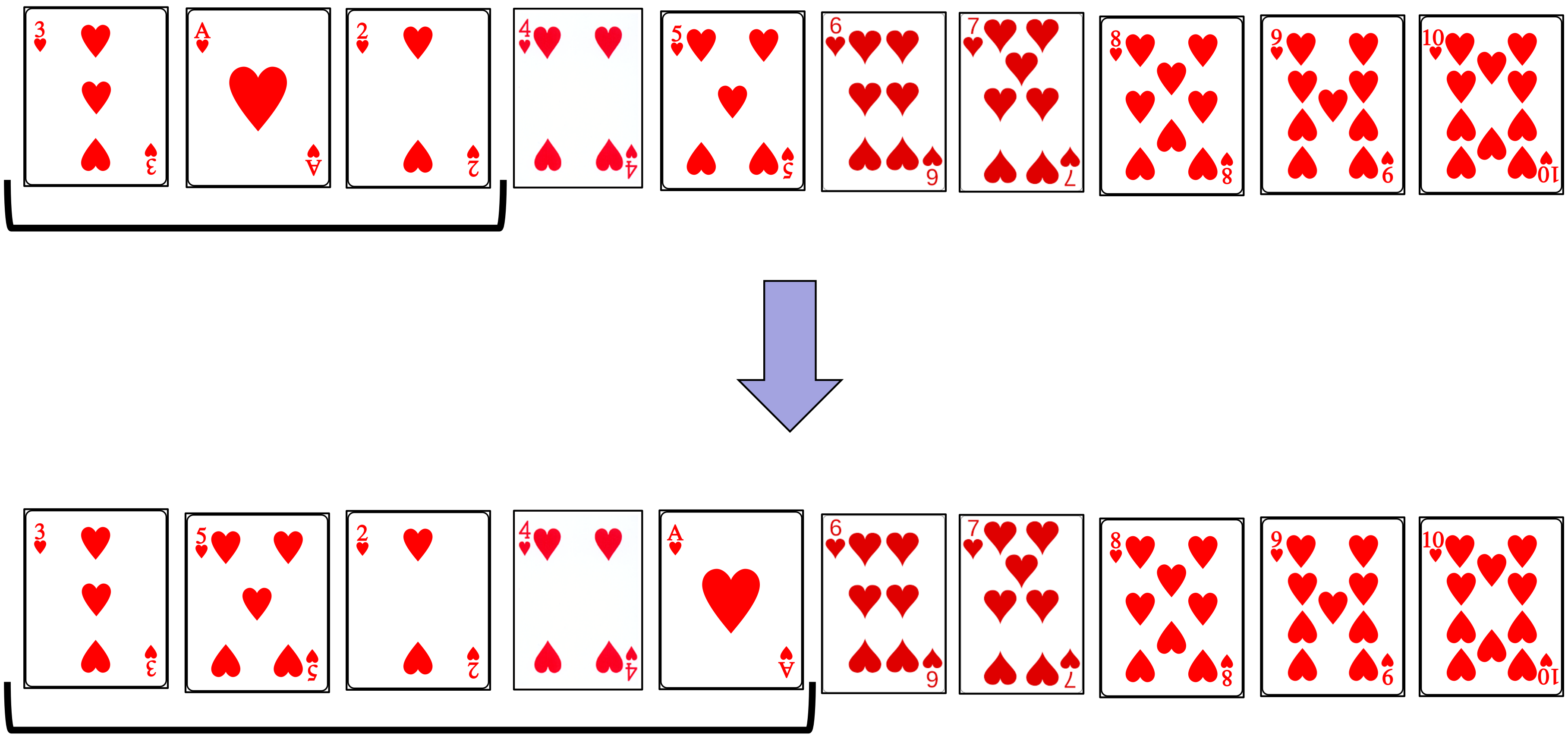
Example: i = 4, r =4



Generating a random permutation

Algorithm: Knuth Shuffle [Fisher/Yates]

Example: i = 5, r =2



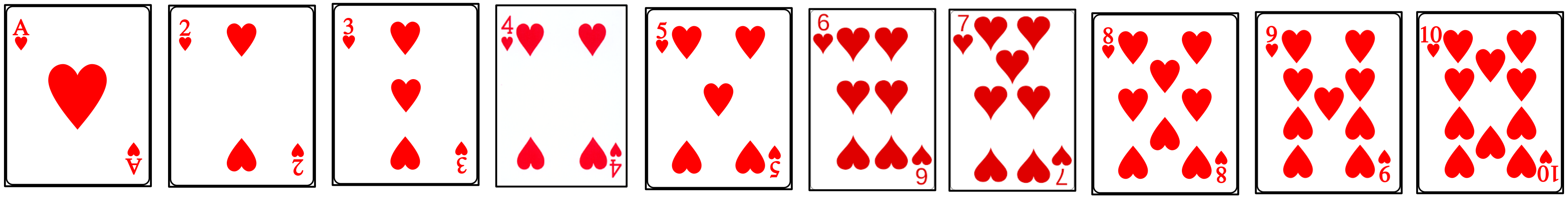
What is the running time of the Knuth Shuffle?

A. $O(\log n)$
B. $O(n)$
C. $O(n \log n)$
D. $O(n^2)$
E. $O(2^n)$

Generating a random permutation

Algorithm: Knuth Shuffle [Fisher/Yates]

Idea: Iterate through array, creating a random prefix.



```
KnuthShuffle(A[1..n])
    for i = 2 to n do
        r = random(1, i)
        swap(A, i, r)
```

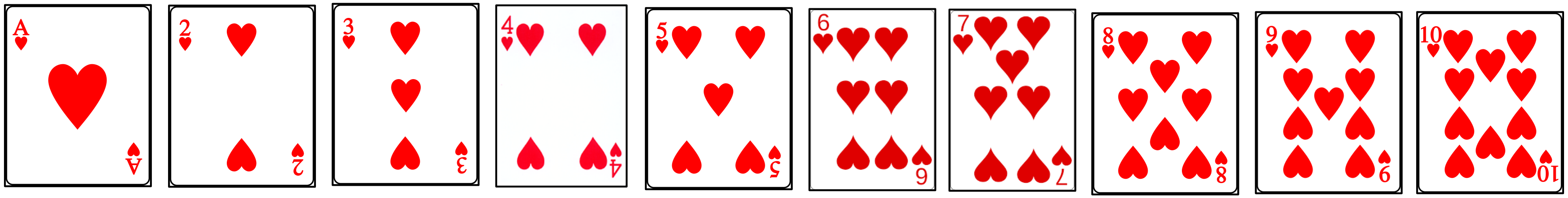
O(n log n)

Generating a random permutation is *faster* than sorting!

Generating a random permutation

Algorithm: Not Knuth Shuffle

What is wrong?



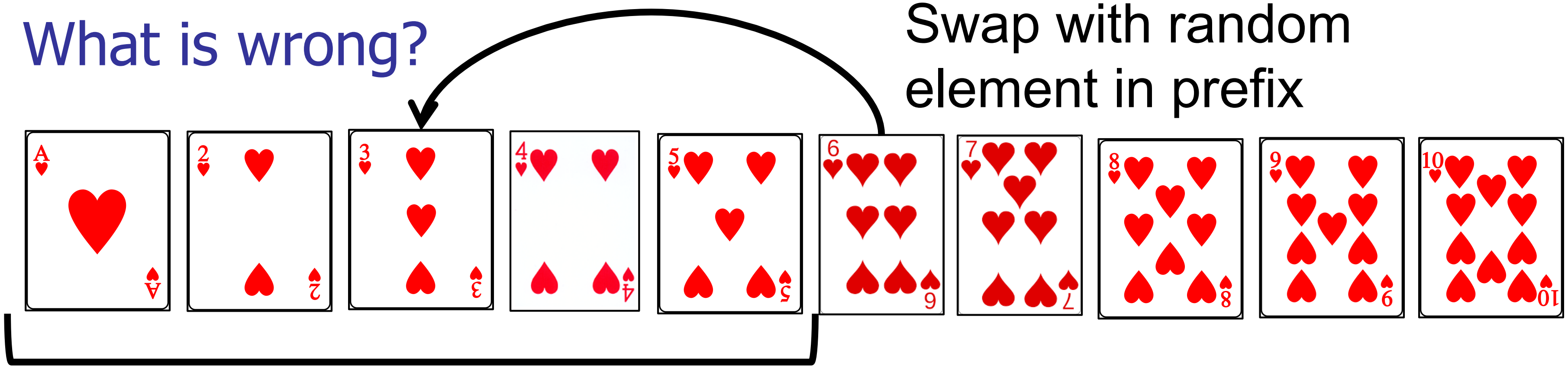
```
NotKnuthShuffle(A[1..n])
    for i = 2 to n do
        r = random(1, i – 1)
        swap(A, i, r)
```

Generating a random permutation

Algorithm: Not Knuth Shuffle

What is wrong?

Swap with random element in prefix



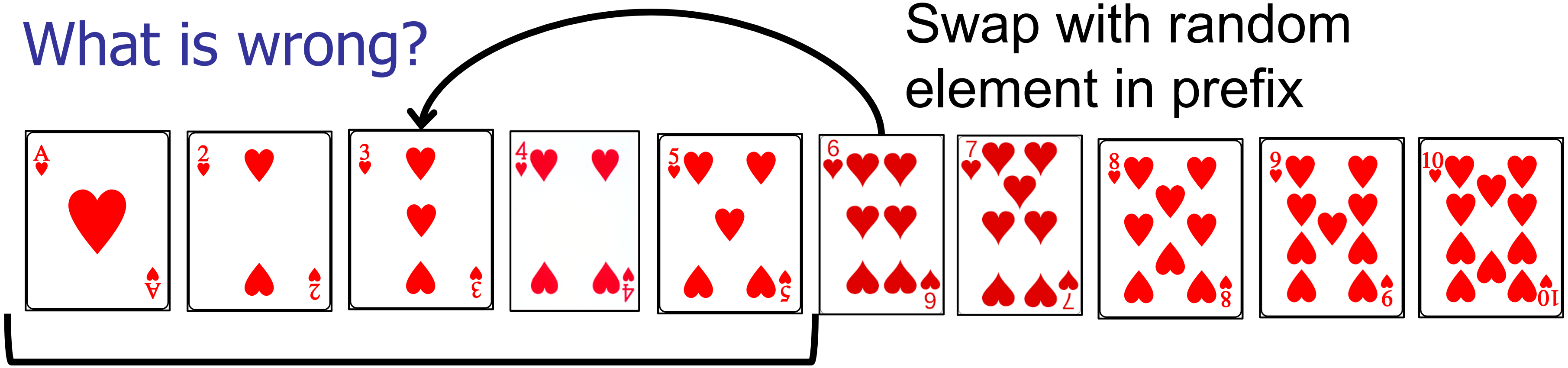
```
NotKnuthShuffle(A[1..n])
    for i = 2 to n do
        r = random(1, i – 1)
        swap(A, i, r)
```

Generating a random permutation

Algorithm: Not Knuth Shuffle

What is wrong?

Swap with random element in prefix



```
NotKnuthShuffle(A[1..n])
    for i = 2 to n do
        r = random(1, i – 1)
        swap(A, i, r)
```

BUG:
Never leaves an element unchanged.

Challenge

How bad is the NotKnuthShuffle?

In a real shuffle, every permutation appears with probability 1/n!

In the NotKnuthShuffle, what is the probability of a given permutation appearing?

Puzzle: Grading Homework

(The Lazy Instructor Problem)

I do not want to grade.

Instead, each of you has to grade the homework of one other student in the class.

I assign each of you a random student's assignment to grade using the KnuthShuffle to permute the pile of homework.

Puzzle: Grading Homework

(The Lazy Instructor Problem)

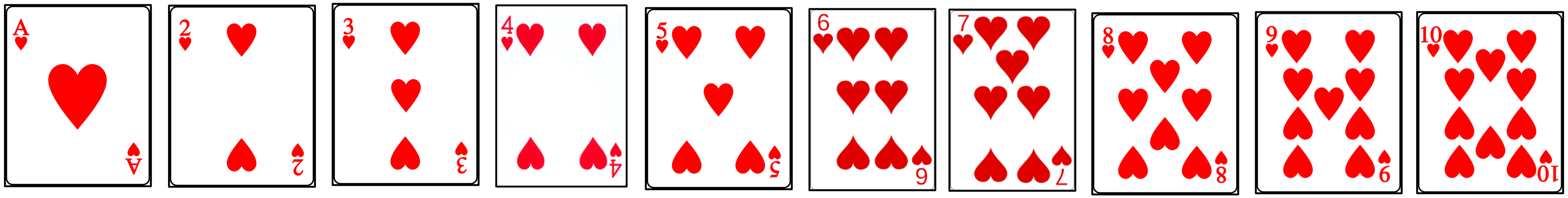
I assign each of you a random student's assignment to grade using the Knuth Shuffle to permute the pile of homework.

What is the expected number of students that grade their own homework?

Generating a random permutation

Algorithm: Alternate Knuth Shuffle

Idea: Pick random element from unsorted suffix.



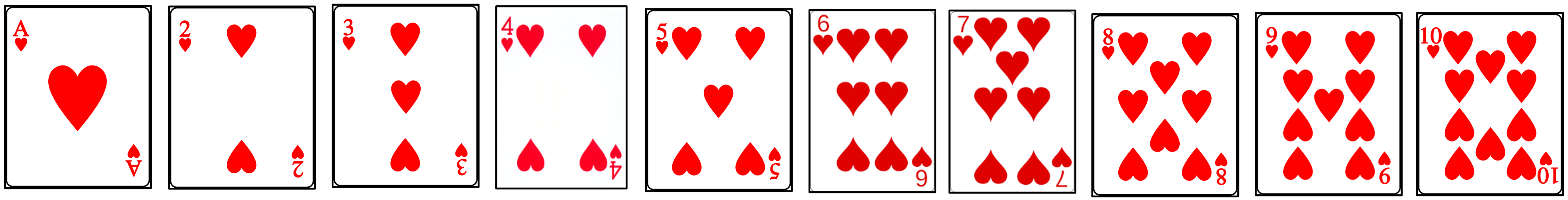
```
AlternateKnuthShuffle(A[1..n])
    for i = 1 to n – 1 do
        r = random(i, n)
        swap(A, i, r)
```

Note: remember to include i in the range.

Generating a random permutation

Is this a valid shuffle?

Idea: Pick random element for each spot in the array.



```
IsItAShuffle?(A[1..n])
    for i = 1 to n do
        r = random(1, n)
        swap(A, i, r)
```

Is it a good shuffle?

A. Yes

B. No

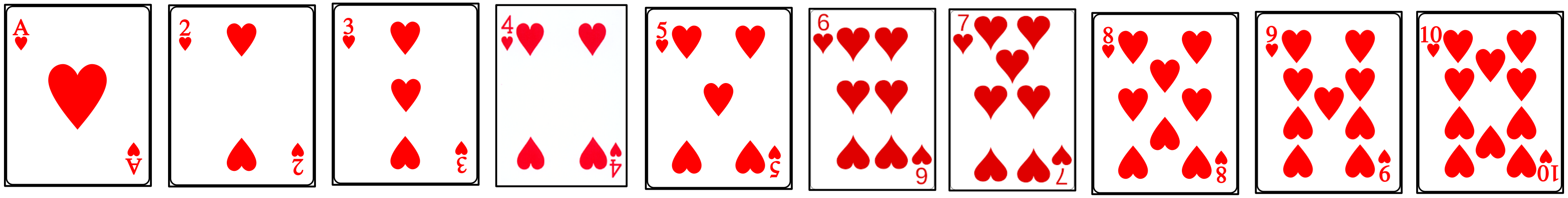
C. Only if n is prime.

```
IsItAShuffle?(A[1..n])
    for i = 1 to n do
        r = random(1, n)
        swap(A, i, r)
```

Generating a random permutation

Not a valid shuffle.

Idea: Pick random element for each spot.



```
IsItAShuffle?(A[1..n])
    for i = 1 to n do
        r = random(1, n)
        swap(A, i, r)
```

Does not generate uniform probability!

Simulate it and see why!

Generating a random permutation

Buggy card shuffle:

```
BadShuffle(card)
    randomize() // Use system clock to seed RNG
    for i = 1 to 52 do
        r = random(51)+1
        swap = card[r]
        card[r] = card[i]
        card[i] = swap
```

Real bug! This was a bug found in PlanetPoker's card shuffling.

See: http://www.developer.com/tech/article.php/616221/How-We-Learned-to-Cheat-at-Online-Poker-A-Study-in-Software-Security.htm

Generating a random permutation

Buggy card shuffle:

```
BadShuffle(card)
    randomize() // Use system clock to seed RNG
    for i = 1 to 52 do
        r = random(51)+1 // between 1 and 51
        swap = card[r]
        card[r] = card[i]
        card[i] = swap
```

Problem 1: Not uniform: chooses random number from [1..n-1].

Card 52 cannot end up in slot 52!
Fix: choose from [1..52]

Generating a random permutation

Buggy card shuffle:

```
BadShuffle(card)
    randomize() // Use system clock to seed RNG
    for i = 1 to 52 do
        r = random(52)+1 // between 1 and 52
        swap = card[r]
        card[r] = card[i]
        card[i] = swap
```

Problem 2: Not uniform: chooses random number from [1..n].

Fix: choose random number from [i, 52] or from [1,i].

Generating a random permutation

Buggy card shuffle:

```
BadShuffle(card)
    randomize() // Use system clock to seed RNG
    for i = 1 to 52 do
        r = random(52)+1 // between 1 and 52
        swap = card[r]
        card[r] = card[i]
        card[i] = swap
```

Problem 3: random uses a 32-bit seed. Only 2^{32} possible permutations generated by this routine.

Generating a random permutation

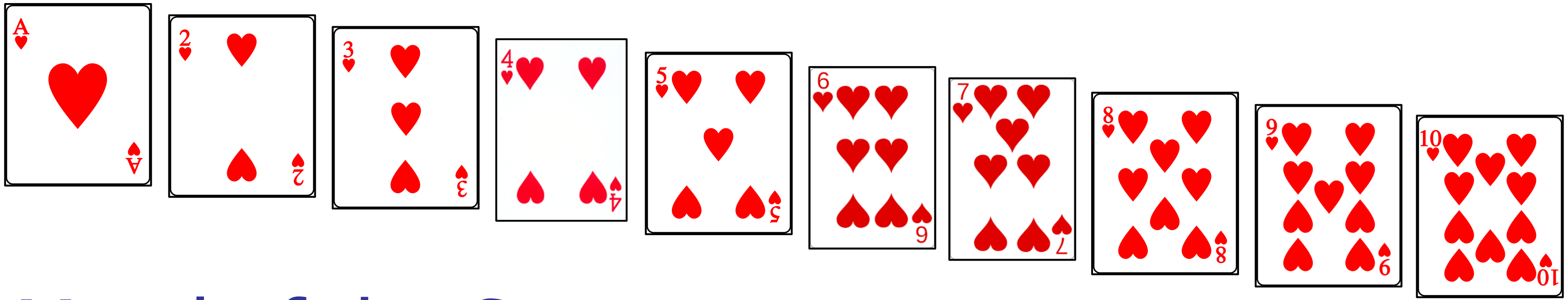
Buggy card shuffle:

```
BadShuffle(card)
    randomize() // Use system clock to seed RNG
    for i = 1 to 52 do
        r = random(52)+1 // between 1 and 52
        swap = card[r]
        card[r] = card[i]
        card[i] = swap
```

Problem 4: System clock is used as seed, i.e., milliseconds since midnight. Only 86.4 million seeds, hence only 86.4 million permutations.

Generating a random permutation

or: How to shuffle a deck of cards.



Moral of the Story

- Shuffling is hard.
- Bugs are subtle.
- If it really matters:
  - Use hardware random number generator.
  - Monitor / measure statistical properties.

Today: Sorting, Part III

Selection and Order Statistics

- QuickSelect

Random permutations

- Sorting Shuffle
- Knuth Shuffle