COMP2823

Lecture 12: Randomized Algorithms
[GT 19.1]

Dr. André van Renssen
School of Computer Science

Some content is taken from material provided by the textbook publisher Wiley.

THE UNIVERSITY OF SYDNEY

Randomized algorithms

Randomized algorithms are algorithms where the behaviour doesn't depend solely on the input. It also depends (in part) on random choices or the values of a number of random bits.

Reasons for using randomization:

- Sampling data from a large population or dataset
- Avoid pathological worst-case examples
- Avoid predictable outcomes
- Allow for simpler algorithms

Randomized algorithms

Randomized algorithms are algorithms where the behaviour doesn't depend solely on the input. It also depends (in part) on random choices or the values of a number of random bits.

- Generating random permutations
- Treaps
- Skip lists (discussed earlier)

Generating random permutations

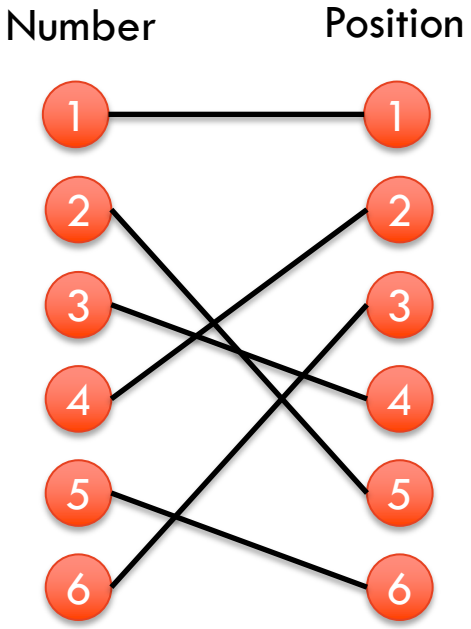
Input: An integer n.

Output: A permutation of {1, …, n} chosen uniformly at random, i.e., every permutation has the same probability of being generated.

Example:

n = 6

<1,4,6,3,2,5>



Generating random permutations

What are random permutations used for?

- Many algorithms whose input is an array perform better in practice after randomly permuting the input (for example, QuickSort).
- Can be used to sample k elements without knowing k in advance by picking the next element in the permuted order when needed.
- Can be used to assign scarce resources.
- Can be a building block for more complex randomized algorithms.

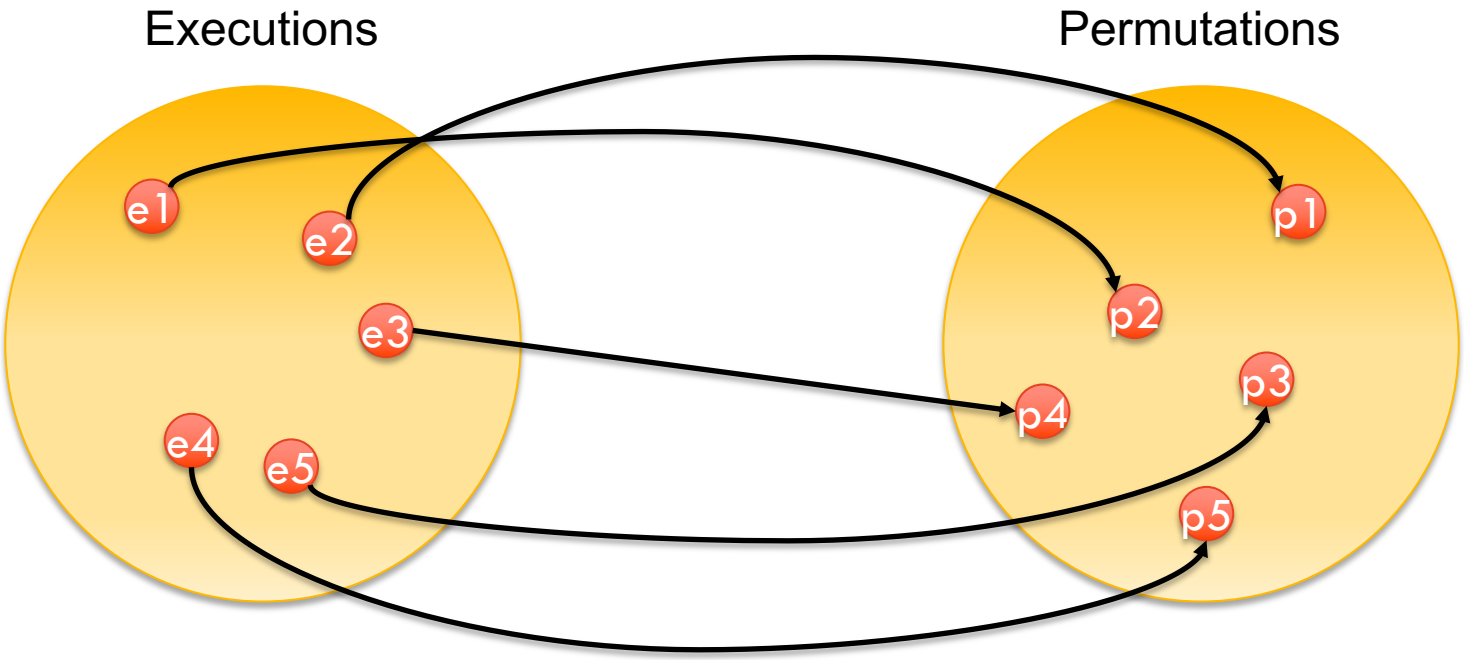
First (incorrect) attempt

```
def permute(A):

  # permute A in place
  n ← length of array A

  for i in [0:n] do
    # swap A[i] with random position
    j ← pick_uniformly_at_random([0:n])
    A[i], A[j] ← A[j], A[i]

  return A
```

Note that since j is picked at random, different executions lead to different outcomes

So, why is this incorrect?

For all permutations to be equally likely, we want that every permutation is generated by the same number of possible executions.



First (incorrect) attempt: Analysis

```
def permute(A):

  # permute A in place
  n ← length of array A

  for i in [0:n] do
    # swap A[i] with random position
    j ← pick_uniformly_at_random([0:n])
    A[i], A[j] ← A[j], A[i]

  return A
```

Number of executions:

$$\underbrace{n * n * n * \ldots * n}_{\text{n times}} = n^n$$

Number of permutations:

$1 * 2 * 3 * \ldots * n = n!$

n^n isn't divisible by $n!$

Example:

n = 3

$n^n = 27$

$n! = 6$

27 isn't a multiple of 6, so some permutations are more likely than others.

Second attempt

```
def FisherYates(A):

  # permute A in place
  n ← length of array A

  for i in [0:n] do
    # swap A[i] with random position
    j ← pick_uniformly_at_random([i:n])
    A[i], A[j] ← A[j], A[i]

  return A
```

Note that since j is picked at random, different executions lead to different outcomes

Second attempt: Analysis

Number of executions:

$1 * 2 * 3 * \ldots * n = n!$

Number of permutations:

$1 * 2 * 3 * \ldots * n = n!$

Observation: Every execution leads to a different permutation.

```
def FisherYates(A):

  # permute A in place
  n ← length of array A

  for i in [0:n] do
    # swap A[i] with random position
    j ← pick_uniformly_at_random([i:n])
    A[i], A[j] ← A[j], A[i]

  return A
```

Example: To generate <3,2,4,1> starting from <1,2,3,4>

<1,2,3,4> → <3,2,1,4>, i=0 and j=2

<3,2,1,4> → <3,2,1,4>, i=1 and j=1

<3,2,1,4> → <3,2,4,1>, i=2 and j=3

<3,2,4,1> → <3,2,4,1>, i=3 and j=3

Second attempt: Analysis

Theorem:

The Fisher-Yates algorithm generates a permutation uniformly at random.

Proof:

- Every execution of the algorithm happens with probability $1/n!$.
- Each execution generates a different permutation.
- Hence, the probability that a specific permutation is generated is $1/n!$, for all possible permutations of $<1, 2, \ldots, n>$.

Alternatives to Balanced Binary Search Trees

Alternative to AVL trees:

- Treaps
- Skiplists (discussed earlier)

So, why are we looking at another different way of doing this?

- More simple data structures built in a randomized way
- No need for rebalancing like in AVL trees
- put and get in O(log n) worst-case time with high probability

Applications:

- Various database systems use it
- Concurrent/parallel computing environments

The Map ADT (recap)

- get(k): if the map M has an entry with key k, return its associated value
- put(k, v): if key k is not in M, then insert (k, v) into the map M; else, replace the existing value associated to k with v
- remove(k): if the map M has an entry with key k, remove it
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterable collection of the values in M

Treap

Given a collection $\{(v_0, p_0), \ldots, (v_{n-1}, p_{n-1})\}$ a Treap is a binary tree T holding these items such that:

1. If we look at the v-values T is binary search tree

   v(descendent of u.left) < v(u) < v(descendent of u.right)

2. If we look at the p-values T is heap

   p(child of u) > p(u)

Theorem:

Given values $\{v_0, \ldots, v_{n-1}\}$ if we pick p_i UAR from [0,1] then a Treap for $\{(v_0, p_0), \ldots, (v_{n-1}, p_{n-1})\}$ has height O(log n) with high probability

Building a Treap

Theorem:

Given a collection $\{(v_0, p_0), \ldots, (v_{n-1}, p_{n-1})\}$ sorted by increasing v-value we build a Treap using a recursive approach in O(n height) time

```
def build_treap(items):

  n ← len(items)
  i ← index in [0:n] minimizing items[i][1]
  left ← build_treap(items[0:i])
  right ← build_treap(items[i+1:n])

  return Node(items[i], left, right)
```

We could reduce the space complexity to O(n) by representing the subproblems implicitly with a pair of indices.

Building a Map

Theorem:

Given values $\{v_0, \ldots, v_{n-1}\}$ we can build in $O(n \log n)$ time a binary search tree with height $O(\log n)$ (with high probability) using the build_treap function by picking priorities at random.

```
def build_map(values):
  sort(values)
  priorities ← list of n random reals in [0, 1]
  items ← pairs of (values, priorities)

  return build_treap(items)
```

But we know how to do this already by putting the median element at the root!

Building a Treap incrementally

Theorem:

Given a treap T for $\{(v_0, p_0), \ldots, (v_{n-1}, p_{n-1})\}$ we insert a new item in O(height) time

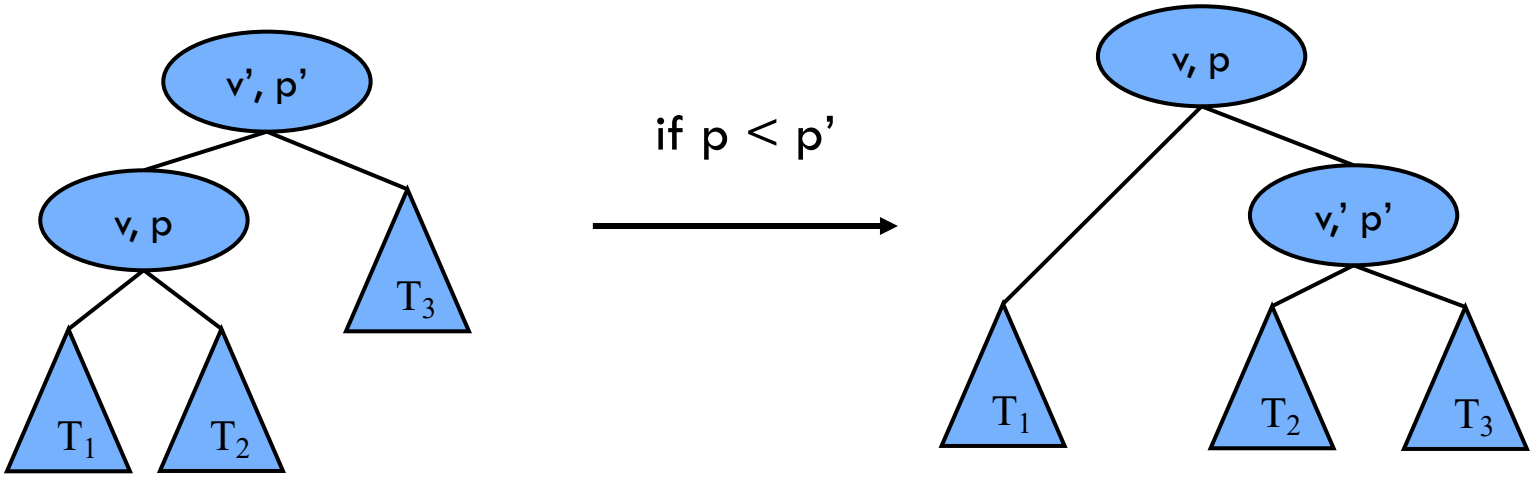
```
def insert_map(T, value):

  priority ← random real in [0, 1]
  T.insert_treap( (value, priority) )
```

Theorem:

We can incrementally build a binary search tree in O(log n) expected time per insertion using the insert_item function by picking priorities at random.

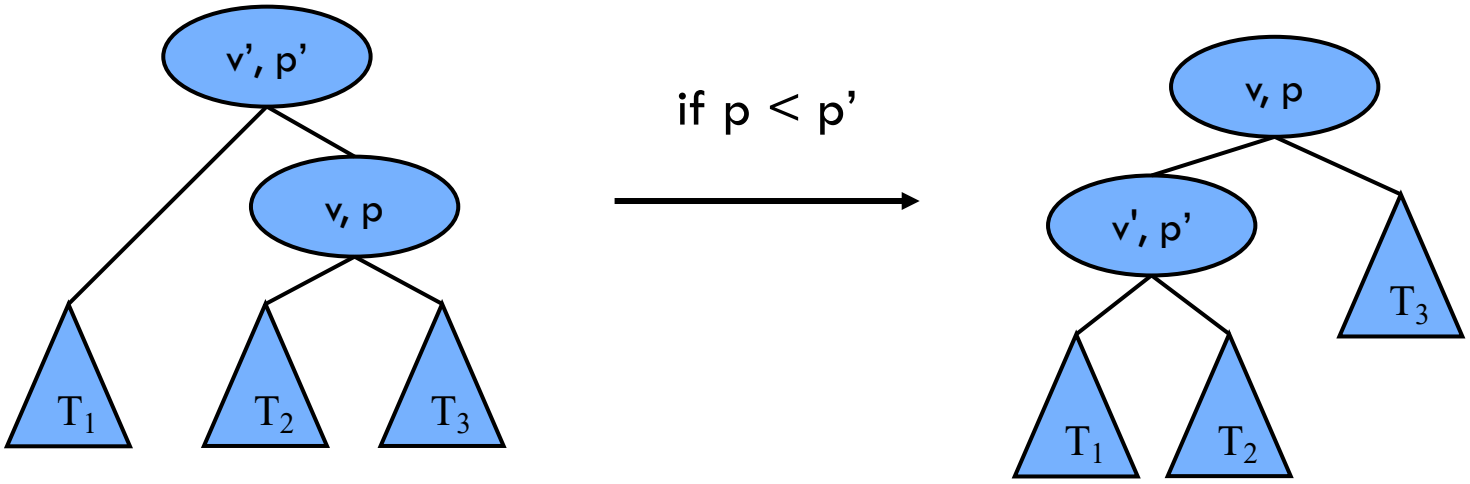
High level idea behind incremental Treap

1. insert (v, p) using the standard BST insert
2. use modified bubble-up routine to fix heap property



High level idea behind incremental Treap

1. insert (v, p) using the standard BST insert
2. use modified bubble-up routine to fix heap property



Expected height

Theorem:

We can incrementally build a binary search tree in O(log n) expected time per insertion using the insert_item function by picking priorities at random.

$$E[depth\ x_k] = \sum_{i=1}^{n} \mathrm{E}[\#x_i\ above\ x_k] = \sum_{i=1}^{n} \Pr[x_i\ above\ x_k]$$

x_i occurs above x_k in the treap exactly when x_i has the smallest priority of all items in $\{x_i, \ldots, x_k\}$ or $\{x_k, \ldots, x_i\}$

$$\Pr[x_i\ above\ x_k] = \begin{cases} \dfrac{1}{k-i+1}, & if\ i < k \\ 0, & if\ i = k \\ \dfrac{1}{i-k+1}, & if\ i > k \end{cases}$$

Expected height

Theorem:

We can incrementally build a binary search tree in O(log n) expected time per insertion using the insert_item function by picking priorities at random.

$$E[depth\ x_k] = \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1}$$

These are harmonic numbers!

$$E[depth\ x_k] = \sum_{i=2}^{k} \frac{1}{i} + \sum_{i=2}^{n-k+1} \frac{1}{i} = H(k) + H(n-k+1)$$

Harmonic numbers are O(log n), so expected height is O(log n)