Assignment 4

Jon Nordby

In [2]:
```
%matplotlib inline
import random
import math

import pandas
import matplotlib.pyplot as plt
```

Exercise 16

In the Select-IDE algorithm (lecture 6), what is the probability that you

1. Select only a single IDE?

The algorithm initial selects the first candidate. And then only switches away from the current IDE if the candidate has higher score.

To select only one IDE, the first candidate would need to have the highest score. The other n-1 can be in any position, giving probability $P(\text{select-one}) = \frac{(n-1)!}{(n)!} = \frac{1}{n}$

2. Select n different IDEs?

Selecting N different IDEs in a sequence of N candidates only happens if the scores are in in strictly increasing order.

There is only one permutation of the sequence with this order, giving probability $P(\text{select-all}) = \frac{1}{n!}$

3. Select exactly two IDEs?

Exercise 17

CLRS 5.2-5: Let A[1..n] be an array of n distinct numbers. If i < j and A[i] > A[j], then the pair (i, j) is called an inversion. Suppose that all the elements of A form a uniform random permutation (that any ordering is equally likely) of . Use indicator random variables to compute the expected number of inversions.

To compute the number of we need to compare each i with each j, which is all the combinations of two numbers.

The probability of an (i, j) pair being an inversion is the probability of A[i] > [j]. Since the values are uniformly distributed with no duplicate values, it has the same chance of being true as false:

$$P(\text{ij-is-inversion}) = X_{inv} = \frac{1}{2}$$

Using k-combination $\frac{n!}{k!(n-k)!}$ with $k = 2$, there are $\frac{n!}{2!(n-2)!} = \frac{n(n-1)(n-2)!}{2!(n-2)!} = \frac{n(n-1)}{4}$ combinations.

The expected number of inversions is the

$$E = I * X_{inv} = \frac{n(n-1)}{4} * \frac{1}{2} = \frac{n(n-1)}{8}$$

Exercise 18

CLRS 5.3-2: The following procedure is supposed to generate any permutation of A with equal probability, except the identity permutation (leaving A unchanged). Does this do so?

```
Permute-Without-Identity(A)
  n = A.length
  for i = 1 to n-1
    swap A[i] with A[Random(i+1, n)]
```

Hint: Begin by experimenting with small arrays A. Assume that all elements of A are unique.

This algorithm *always* swaps each item away, and there is no way for the item be swapped back to the same position. Thus all permutations where one or more of the items stay in the original location cannot happen.

In [10]:
```
def permutation_without_identity(arr):
    n = len(arr)
    for i in range(0, n-1):
        swapidx = random.randint(i+1, n-1)
        arr[i], arr[swapidx] = arr[swapidx], arr[i]
    return arr

A = list(range(0, 3))
permutations = [permutation_without_identity(A.copy()) for _ in range(0, 15)]
print('original:', A)
print('\n'.join(str(a) for a in permutations))
```

```
original: [0, 1, 2]
[2, 0, 1]
[2, 0, 1]
[2, 0, 1]
[2, 0, 1]
[1, 2, 0]
[1, 2, 0]
[1, 2, 0]
[1, 2, 0]
[1, 2, 0]
[1, 2, 0]
[2, 0, 1]
[1, 2, 0]
[2, 0, 1]
[2, 0, 1]
[2, 0, 1]
```

Experiment confirms that permutations where 0 stays at index 0, or 1 at index 1, or 2 at index 2 does not occur.

Exercise 19

CLRS 6.2-5: The Max-Heapify algorithm is recursive. There is only a single recursive function call, and it occurs at the very end of the algorithm. This is an example of tail recursion. Since function calls can be quite expensive, one usually tries to avoid tail recursion by replacing recursion by a loop construct.

Write a Max-Heapify algorithm that uses looping instead of recursion (pseudocode)!

Following conventions in CLRS

```
Max-Heapify-Looping(A, i)

  while True
    l = left(i)
    r = right(i)
    if l < A.heap-size and A[l] > A[i]
      largest = l
    if r < A.heap-size and A[r] > A[largest]
      largest = r
    if largest != i
      exchange A[i] with A[largest]
      i = largest
    else
      break
```

Extra credit

Implement recursive and iterative Max-Heapify and Build-Max-Heap in Python and compare their performance on random input arrays.

Recursive implementation

In [57]:
```
def max_heapify_recursive(H, p):
    l = 2*p + 1
    r = 2*p + 2
    largest = p
    length = len(H)
    if l < length and H[l] > H[largest]:
        largest = l
    if r < length and H[r] > H[largest]:
        largest = r
    if largest != p:
        H[largest], H[p] = H[p], H[largest]
        max_heapify_recursive(H, largest)

def build_max_heap_recursive(A):
    H = A.copy()
    for i in range(len(H)//2, -1, -1):
        max_heapify_recursive(H, i)
    return H

def get(A, i, default=None):
    try:
        return A[i]
    except IndexError:
        return default

def test_max_heap(build_max_heap):
    for _ in range(0, 10):
        A = [ random.randint(-99, 99) for _ in range(7) ]
        H = build_max_heap(A)

        # Check that max-heap property is kept: parent is always larger than child
        for i in range(len(H)//2, -1, -1):
            p = H[i]
            l = get(H, i*2 + 1, -math.inf)
            r = get(H, i*2 + 2, -math.inf)
            assert p >= l
            assert p >= r


test_max_heap(build_max_heap_recursive)
'tests passed'
```

Out[57]: 'tests passed'

Iterative implementation

In [58]:
```
def max_heapify_iterative(H, top):
    p = top
    length = len(H)
    while True:
        l = 2*p + 1
        r = 2*p + 2
        largest = p
        if l < length and H[l] > H[largest]:
            largest = l
        if r < length and H[r] > H[largest]:
            largest = r
        if largest != p:
            H[largest], H[p] = H[p], H[largest]
            p = largest
        else:
            break

def build_max_heap_iterative(A):
    H = A.copy()
    for i in range(len(H)//2, -1, -1):
        max_heapify_iterative(H, i)
    return H

test_max_heap(build_max_heap_iterative)
'tests passed'
```

Out[58]: 'tests passed'

Recursive versus iterative heap building

In [43]:
```
def random_integers(l):
    return [ random.randint(-9999, 9999) for _ in range(l+1) ]
heap_inputs = random_integers(10000)
```

In [44]:
```
%timeit build_max_heap_recursive(heap_inputs)
```

```
10 loops, best of 3: 22.4 ms per loop
```

In [45]:
```
%timeit build_max_heap_iterative(heap_inputs)
```

```
10 loops, best of 3: 19.5 ms per loop
```

The iterative version is approximately 10% faster than the recursive one

Exercise 20

CLRS 6.5-7

- Show how to implement a FIFO queue with a priority queue.
- Show how to implement a LIFO stack with a priority queue.

Using Python as executable pseudocode.

Note: No explicit handling of overrun/underrun. We also assume that integers are arbitrary length, which in Python they can be, but in other languages would be finite and thus subject to overflow.

In [30]:
```
from queue import PriorityQueue

class Stack():
    def __init__(self, size):
        self._queue = PriorityQueue(size) # minimum-priority-queue
        self._priority = 0

    def pop(self):
        priority, val = self._queue.get()
        return val

    def push(self, val):
        # ensure this value will be first returned (LIFO),
        # by assigning priorities monotonically decreasing
        self._priority -= 1
        self._queue.put((self._priority, val))


def check_lifo(values):
    s = Stack(len(values))
    for val in values:
        # check that push/pop gives same element back
        s.push(val)
        popped = s.pop()
        assert popped == val
        # put the element so can check results when filled
        s.push(val)

    out = [ s.pop() for _ in enumerate(values) ]

    rev = list(reversed(out))
    assert values == rev

check_lifo([random.randint(-999, 999) for _ in range(5)])
print('tests passed')
```

```
tests passed
```

In [35]:
```
class Queue():
    def __init__(self, size):
        self._queue = PriorityQueue(size) # minimum-priority-queue
        self._priority = 0

    def dequeue(self):
        priority, val = self._queue.get()
        return val

    def enqueue(self, val):
        # ensure that the value will be retrieved last (FIFO)
        # by assigning priorities monotonically increasing
        self._priority += 1
        self._queue.put((self._priority, val))

def check_fifo(values):
    # check that order is preserved, first in should be first out
    q = Queue(len(values))
    for val in values:
        q.enqueue(val)

    out = [ q.dequeue() for _ in enumerate(values) ]
    assert out == values

check_fifo([random.randint(-999, 999) for _ in range(5)])
print('tests passed')
```

```
tests passed
```

If the underlying priority queue had been a max-priority-queue instead of min-priority-queue, then priority order would have to be inverted: LIFO would assign increasing priority values, and FIFO decreasing priority values.