Useful facts

Order Notation Summary:

$f(n) \in O(g(n))$ if $\exists\ c > 0$ and $n_0 \geq 0$ such that $|f(n)| \leq c\,|g(n)|\ \forall\ n \geq n_0$.
$f(n) \in \Omega(g(n))$ if $\exists\ c > 0$ and $n_0 \geq 0$ such that $|f(n)| \geq c\,|g(n)|\ \forall\ n \geq n_0$.
$f(n) \in \Theta(g(n))$ if $\exists\ c_1, c_2 > 0$ and $n_0 \geq 0$ such that $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|\ \forall\ n \geq n_0$.
$f(n) \in o(g(n))$ if $\forall\ c > 0\ \exists\ n_0 \geq 0$ such that $|f(n)| \leq c\,|g(n)|\ \forall\ n \geq n_0$.
$f(n) \in \omega(g(n))$ if $\forall\ c > 0\ \exists\ n_0 \geq 0$ such that $|f(n)| \geq c\,|g(n)|\ \forall\ n \geq n_0$.

Some useful sums:

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
- $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$
- $\sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma + o(1) \in \Theta(\log n)$
- $\sum_{i=0}^{\infty} \frac{1}{2^i} = 2$
- $\sum_{i=0}^{\infty} \frac{i}{2^i} = 2$
- $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$.
- $\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$

Some well-known sequences:

n	0	1	2	3	4	5	6	7	8	9
Power of 2, 2^n	1	2	4	8	16	32	64	128	256	512
Factorial $n!$	1	1	2	6	24	120	720	5040	40320	362 880
Fibonacci number $F(n)$	0	1	1	2	3	5	8	13	21	34
Catalan-number $C(n)$	1	1	2	5	14	42	132	429	1430	4862

Randomization, probability and moments:

- `random(int` n`)` returns an integer in $\{0, \ldots, n-1\}$, chosen uniformly.
- $E[aX] = aE[X]$, $E[X + Y] = E[X] + E[Y]$ (linearity of expectation)
- $V[X] = V[a + X]$
- Chebyshev's inequality: $P(|X - E[X]| \geq t) \leq \frac{V(X)}{t^2}$

Some recursions that we have seen:

Recursion	resolves to
$T(n) = T(n/2) + \Theta(1)$	$T(n) \in \Theta(\log n)$
$T(n) = 2T(n/2) + \Theta(n)$	$T(n) \in \Theta(n \log n)$
$T(n) = 2T(n/2) + \Theta(\log n)$	$T(n) \in \Theta(n)$
$T(n) = T(cn) + \Theta(n)$ for some $0 < c < 1$	$T(n) \in \Theta(n)$
$T(n) = \frac{1}{2}T(\frac{3}{4}n) + \frac{1}{2}T(n-1) + \Theta(1)$	$T(n) \in \Theta(\log n)$
$T(n) = \frac{1}{n}\sum_{i=0}^{n-1} \max\{T(i), T(n-i-1)\} + \Theta(n)$	$T(n) \in \Theta(n)$
$T(n) = \frac{2}{n}\sum_{i=2}^{n-1} T(i) + \Theta(n)$	$T(n) \in \Theta(n \log n)$
$T(n) = \frac{4}{n}\sum_{i=2}^{n-1} T(i)$	$T(n) \in \Theta(n^3)$
$T(n) = T(\sqrt{n}) + \Theta(\sqrt{n})$	$T(n) \in \Theta(\sqrt{n})$
$T(n) = T(\sqrt{n}) + \Theta(1)$	$T(n) \in \Theta(\log \log n)$

Pseudocode from slides

Efficient sorting with heaps

- Idea: *PQ-sort* with heaps.
- $O(1)$ auxiliary space: Use same input-array A for storing heap.

```
HeapSort(A, n)
    // heapify
    n ← A.size()
    for i ← parent(last()) downto 0 do
        fix-down(A, i, n)

    // repeatedly find maximum
    while n > 1
        // 'delete' maximum by moving to end and decreasing n
        swap items at A[root()] and A[last()]
        decrease n
        fix-down(A, root(), n)
```

The for-loop takes $\Theta(n)$ time and the while-loop takes $O(n \log n)$ time.

Merging Meldable Heaps

- Idea: Merge heap with smaller root into other one, *randomly* choose into which sub-heap to merge.
- Structural property not maintained

```
meldableHeap::merge(r1, r2)
r1, r2: roots of two heaps (possibly NIL)
returns root of merged heap
    if r1 is NIL return r2
    if r2 is NIL  return r1
    if r1.key < r2.key   swap(r1, r2)
    // now r1 has max-key and becomes the root.
    randomly pick one child c of r1
    replace subheap at c by heapMerge(c, r2)
    return r1
```

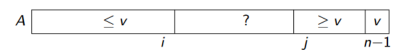

Making Binomial Heaps Proper

```
binomialHeap::makeProper()
    n ← size of the binomial heap
    compute ℓ ← ⌊log n⌋
    B ← array of size ℓ + 1, initialized all-NIL
    L ← list of flagged trees
    while L is non-empty do
        T ← L.pop(), h ← T.height
        while T′ ← B[h] is not NIL do
            if T.root.key < T′.root.key do swap T and T′
            // combine T with T′
            T′.right ← T.left, T.left ← T′, T.height ← h+1
            B[h] ← NIL, h++
        B[h] ← T
    // copy B back to list
    for (h = 0; h ≤ ℓ; h++) do
        if B[h] ≠ NIL do L.append(B[h])
```


Efficient In-Place partition (Hoare)

Idea: Keep swapping the outer-most wrongly-positioned pairs.

Loop invariant:




```
partition(A, p)
A: array of size n,   p: integer s.t. 0 ≤ p < n
1.   swap(A[n−1], A[p])
2.   i ← −1,   j ← n−1,   v ← A[n−1]
3.   loop
4.       do i ← i+1 while A[i] < v
5.       do j ← j−1 while j ≥ i and A[j] > v
6.       if i ≥ j then break   (goto 9)
7.       else swap(A[i], A[j])
8.   end loop
9.   swap(A[n−1], A[i])
10.  return i
```

Running time: $\Theta(n)$.

Insert in Skip Lists

```
skipList::insert(k, v)
    P ← getPredecessors(k)
    for (i ← 0; random(2) = 1; i ← i+1) {}    // random tower height
    while i ≥ P.size()                         // increase skip-list height?
        root ← new sentinel-only list, linked in appropriately
        add left sentinel of root at bottom of stack P
    p ← P.pop()                                // insert (k, v) in S_0
    z_below ← new node with (k, v), inserted after p
    while i > 0                                // insert k in S_1, ..., S_i
        p ← P.pop()
        z ← new node with k added after p
        z.below ← z_below; z_below ← z
        i ← i − 1
```


Randomizing QuickSelect: Shuffle

Goal: Create a randomized version of *QuickSelect*.

First idea: Randomly permute the input first using *shuffle*:

```
shuffle(A)
A: array of size n
    for i ← 1 to n−1 do
        swap( A[i], A[random(i+1)] )
```

This works well, but we can do it directly within the routine.