# Algorithms

*Example Sheet 5*

Petar Veličković

Lent Term 2017

## Warm-up

1. Considering the stack framework with `push(x)`, `pop()` and `multipop(k)`:
   (a) Assuming that the stack started with $s_0$ elements and ended up with $s_n$ elements after $n$ operations, derive an expression for the worst-case time complexity, $T(n, s_0, s_n)$, of those $n$ operations. How does this relate to the results obtained by the potential method?
   (b) How does the amortised analysis change if we also allow a `multipush(A)` operation, where $A$ is an array?
2. What is the worst-case time complexity of the Fibonacci heap operations?
3. Describe a linked-list implementation of the disjoint-set data structure. Derive the time complexities of each relevant operation.
4. Unfortunately, the amortised complexity analysis of the disjoint-set data structure with path compression is overly complicated for us to dedicate enough time to it now. However, it should be insightful for you to get a feel for just how slowly $\alpha(n)$ grows:
   (a) Define the function $A_k(n)$, for integers $k \geq 0$, $n \geq 1$, as follows:

$$A_k(n) = \begin{cases} n+1 & k = 0 \\ \underbrace{A_{k-1}(A_{k-1}(\dots(A_{k-1}}_{n+1 \text{ times}}(n))\dots)) & k \geq 1 \end{cases}$$

   Obtain closed-form expressions for $A_1(n)$ and $A_2(n)$.
   [*Hint:* use induction.]
   (b) Use the above results to compute $A_3(1)$ and $A_4(1)$. Compare $A_4(1)$ to the number of atoms in the observable universe.

   $\alpha(n)$ is defined as the lowest $k$ such that $A_k(1) \geq n$.

## Exercises

1. Propose an implementation for an expandable array data structure (sometimes also called a vector). This structure has to:

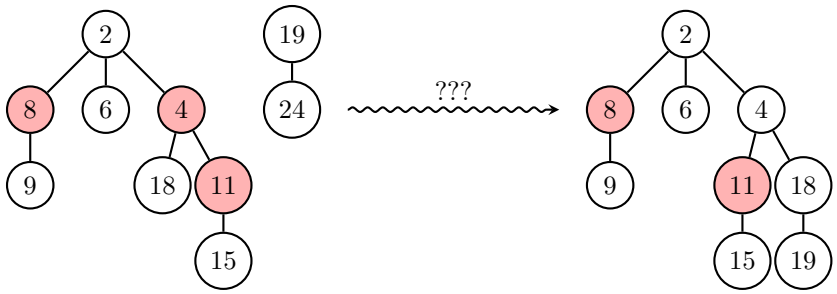
   - Maintain a contiguous block of memory storing all of its elements, allowing random-access times of $O(1)$;
   - Support a `push_back(x)` operation, that will insert an item to the end of an array.
   - Support a `pop_back()` operation, discarding the item at the end of the array.
   - Have an $O(1)$ (amortised) time complexity for both of the above operations, and an $O(n)$ space complexity at all times, when $n$ items are in the array.

   Perform complexity analysis of your structure, using the potential method, to prove that some of the above claims are satisfied.

2. Prove that, if we never call `decreaseKey` or `delete` in a Fibonacci heap, that the number of children of any node is at most $O(\log n)$, where $n$ is the current number of nodes in the heap.

3. While Fibonacci heaps were designed with the intent of preventing its constituent trees from growing too "wide and shallow", nothing stops them from becoming "tall and narrow". Demonstrate this by providing a sequence of operations that, for a given integer $n$, produce a Fibonacci heap that contains a single tree of $n$ nodes, arranged in a chain.

4. Considering the following two Fibonacci heaps (red nodes are marked), one was obtained from the other as a result of three operations. Find these operations: only `extractMin`, `insert` and `decreaseKey` are allowed!




5. Consider the *off-line minimum problem*: we are required to maintain a set of elements under the `insert` and `extractMin` operations. However, now we are given in advance the $n$ `insert` operations and $m$ `extractMin` operations, as well as the order in which they were executed. Assuming no element was inserted twice, devise an algorithm to provide us with the

elements that have been extracted at each call to `extractMin`. Carefully analyse its time complexity.
[*Hint:* it may be beneficial to first consider the case where the elements come from the set $\{1, 2, \ldots, n\}$.]

6. Solve *Exercise 7* from the pre-sheet. [*Hint:* The desirable complexity is $O(N^2(\log N + \alpha(N)))$.]

## Implementation

This week's implementation exercise concerns *Kruskal's algorithm*. You are required to:

1. Implement the optimised disjoint-set data structure with union-by-rank and path compression;
2. Use this data structure as a component in implementing Kruskal's algorithm (using an appropriate data structure representation for graphs);
3. Apply the resulting algorithm to a few graphs of your choice, to verify its correctness.