Homework #1
Invariants, Asymptotic and Amortized Analysis
Due: Oct. 22, 2018

- The assignment should be delivered digitally by email. Your message should be sent to **pribeiro@dcc.fc.up.pt** with subject *"[ALG HWK1] - FirstName Last Name StudentNumber"*
- Your delivery should be an **electronic document**. This could be something like:
  - scanned handwritten answers (make sure they are legible)
  - a plain text file (.txt)
  - a PDF created with another program (we recommend the usage of LaTeX)
- You may work in a (small) group, but you should do your **individual writeup**. This means you can collaborate by talking about the problems, but **you should not copy writeups**.
- Please **acknowledge any help you got** and state any references you consulted (including internet pages) and any students with whom you talked about the problem.
- Answers should be **submitted until 23:59 of the due date**. Up to 24h of delay will get you a 25% penalty. 24h to 48h of delay will get you a 50% penalty. After 48h your work will not be counted.

Invariants

1. Consider the **maximum subarray problem** in which given an array A with n numbers you want to find the contiguous subarray with the largest possible sum. In other words, if the positions in A go from 1 to n, you want to find i and j, with $1 \leq i \leq j \leq n$ such that $\sum_{k=i}^{j} A[k]$ is maximal. One possible $\mathcal{O}(n)$ algorithm for this (**Kadane's algorithm**) can be described by the following pseudo-code:

```
maxSubArray(A, n)
    tmp = best = A[1]
    for i = 2 to n
        tmp = max(A[i], tmp + A[i])
        best = max(best, tmp)
    return best
```

   (a) Consider an array A containing 8 integers: $\{2, 4, -8, 4, 3, -1, 5, -2\}$. **What would the output be?** Explain what happens in the iterations, showing the values of *tmp* and *best* at each iteration.

   (b) **Prove** that the algorithm is always correct using **invariants**.
   Hint: what is the meaning of variables *tmp* and *best*? What do you know about them if you stop the program at the start of an iteration? After establishing a suitable invariant, use the 4 items given in the class (*initialization*, *maintenance*, *termination* and *progress*) to make the proof.

Asymptotic Notation

2. For each pair $< f, g >$ of functions below, list **which of the following are true:** $f(n) = o(g(n))$, $f(n) = \Theta(g(n))$, or $g(n) = o(f(n))$.

   (a) $f(n) = n \log_3(n)$, $g(n) = n \log_4(n)$

   (b) $f(n) = n!$, $g(n) = n^n$.

   (c) $f(n) = n/\log_2(n)$, $g(n) = n^{\log_4(3)}$

   Give a brief explanation of how you arrived to each result.

   Hint: (a) what is the effect of changing a base of the logarithm? (b) can you compare both functions *"term by term"*? (c) when in doubt... draw!

3. **Prove** that for any real constants c and k where $k > 0$,

   $(n + c)^k = \Theta(n^k)$

   Hint: note that c and k might not be integers, so you argument cannot be based on that.

Recurrences

For the following exercises, assume that $T(n)$ takes constant time for sufficiently small n.

4. Each recurrence below solves to one of the following:
   A. $\Theta(\log\log n)$, B. $\Theta(\log n)$, C. $\Theta(n)$, D. $\Theta(n \log n)$, $E.\Theta(n^2)$, F. $\Theta(n^2 \log n)$, G. $\Theta(n^3)$, H. $\Theta(n^4)$
   For each one, **write down the letter of the correct answer.**

   (a) $\square$ $T(n) = 3T(n/3) + n^2$

   (b) $\square$ $T(n) = T(n-1) + 2n^3$

   (c) $\square$ $T(n) = T(n/2) + T(n/3) + n$

   (d) $\square$ $T(n) = T(\sqrt{n}) + 1$

   Give a brief explanation of how you arrived to each result.

   Hint: Use any method you like for each recurrence.

5. Solve the following recurrence, giving your answer in Θ notation.

   $$T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n.$$

   (for instance, we might get this from a divide-and-conquer procedure that uses linear time to break the problem into $\sqrt{n}$ pieces of size $\sqrt{n}$ each.)

   Hint: write out the recursion tree. You may assume n to be of the form 2^k.)

Amortized Analysis

6. An **ordered stack** is a data structure that stores a sequence of items and supports the following operations.

   - `orderedpush(`x`)` - removes all items smaller than x from the beginning of the sequence and then adds x to the beginning of the sequence.
   - `pop()` - deletes and returns the first item in the sequence (or `NULL` if the sequence is empty.

   Suppose we implement an ordered stack with a simple linked list, using the obvious `orderedpush` (traverse the list until correct position) and `pop` algorithms. **Show** that if we start with an empty data structure, the **amortized cost** of each `orderedpush` or `pop` operation is $\mathcal{O}(1)$.

   Hint: before an element is removed from the list... it needs to be inserted in it!

7. In the class, we analyzed a simple **binary counter** in which the cost was equal to the number of bits that we needed to flip. We saw that starting with 0, after n **increments** the amortized cost is just $\mathcal{O}(1)$, leading to a total cost of $\mathcal{O}(n)$. This counter does not give us a small amortized cost when we allow **increments and decrements** so we are going to build one in this problem.

   (a) **Start by showing** that, starting from 0 and without making the counter go negative, it is possible for a sequence of n increment and decrement operations to cost as much as $\Omega(\log n)$ amortized per operation (i.e. $\Omega(n \log n)$ total cost) in the simple counter.

   In order to reduce the previous cost, we will build a counter that uses a *ternary number system*. A number is represented by a sequence of *trits*, each of which is 0, +1 or -1. The value of a k-trit number represented by $t_{k-1}, ..., t_0$ is defined to be $\sum_{i=0}^{k-1} t_i 2^i$

   Notice that this is similar to binary arithmetic, but this time we also allow the coefficient to be -1. For example, $[1, 0, -1]$ is a representation of $1 \times 2^2 + 0 \times 2^1 - 1 \times 2^0 = 3$. Since $[0, 1, 1]$ also represents 3 $(0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 3)$, you can see that this is a *redundant number system.*

   The process of incrementing a trit is similar to the operation on binary numbers: you add 1 to the low order trit. If the result is 2, then it is changed to 0, a carry is propagated to the next trit and this process is repeated until no carry results. Decrementing works similarly: subtract 1 from the low order trit and, if it becomes -2, replace it by 0 and propagate accordingly. As before, the cost of an increment or a decrement is the number of trits that change in the process.

   (b) As before, starting from 0, a sequence of n increments and decrements is performed. Give a **clear proof** that, with this representation, the amortized cost per operation is $\mathcal{O}(1)$. This implies that the total cost for *any* sequence of n operations is $\mathcal{O}(n)$, no matter how the increments and decrements occur in this sequence.

   Hint: use any one of the three methods given (*aggregate*, *accounting* or *potential*)