# Assignment 1:
# Traditional Complexity Analysis

15-819: Foundations of Quantitative Program Analysis (Fall 2019)

Out: Tuesday, September 3, 2019
Due: Tuesday, September 17, 2019 11:59pm EDT

## Introduction

Welcome to *15-819: Foundations of Quantitative Program Analysis*.

Grading is based on biweekly homework assignments and a final project. Both letter grades and pass/fail grades are possible.

Assignments have to be submitted on Gradescope at `https://www.gradescope.com/courses/59187`. You can enroll in the course using the code *9N2EY5*.

For letter grades, homework assignments count 70 percent and the final project counts 30 percent. For the homework assignments, you have *4 late days* that you can use during the semester. When you are out of your late day allowance, each late day will reduce the points of the submitted solution by 25 percent. *Submissions of homework solutions that are more than 2 days late will not accepted.*

CS PhD students are assigned a pass/fail grade in the university grading system, but are given an internal letter grade for Black Friday purposes. A final letter grade of B is required to pass this course. To achieve this, you must have (1) completed all homework assignments on-time with a grade of B; and (2) earned a grade of B or better on the course project. During the semester, two homework assignments can be resubmitted if your grade is below B. Each homework can be resubmitted only once.

The same rules apply for other students who elect to receive a pass/fail grade.

## 1 Asymptotic Behavior and Multiple Variables

Asymptotic notations (such as Big-O) are often used without much care and are sometimes downright abused. One such abuse is that textbooks and courses on analysis of algorithms often define asymptotic notations only for univariate functions (e.g., functions with one argument) but use it to analyze functions with a multivariate resource behavior.

To start with, it is of course problematic that an expression like $O(nm)$ is undefined. But more importantly, it is not easy to *come up* with a sensible definition. First, recall the definition of Big-O: Let $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be two functions from natural number to non-negative reals. We write $f(x) \in O(g(x))$ if there exists $C > 0$ and $N \in \mathbb{N}$ such that $f(x) \leq C \cdot g(x)$ for all $x \geq N$.

The following two definitions are straightforward generalizations that often appear in the literature. Let $f, g : \mathbb{N}^k \to \mathbb{R}_{\geq 0}$ be two non-negative real-valued functions.

**Definition 1.** *We write* $g(\vec{n}) \in O_\forall(f(\vec{n}))$ *if there exists* $c \in \mathbb{R}_{\geq 0}$ *and* $N \in \mathbb{N}$ *such that* $g(n_1,\ldots,n_k) \leq c \cdot f(n_1,\ldots,n_k)$ *for every* $\vec{n}$ *with* $n_1 \geq N$, …, *and* $n_k \geq N$.

**Definition 2.** *We write* $g(\vec{n}) \in O_\exists(f(\vec{n}))$ *if there exists* $c \in \mathbb{R}_{\geq 0}$ *and* $N \in \mathbb{N}$ *such that* $g(n_1,\ldots,n_k) \leq c \cdot f(n_1,\ldots,n_k)$ *for every* $\vec{n}$ *with* $n_j \geq N$ *for some positive integer* $j \leq k$.

However, it is unclear if these definitions have similar properties as the univariate version. Consider for example the following OCaml program.

```
let f(m,n) = for i = 1 to m-1 do
               g(i,n)
             done
```

Let $F(m,n)$ be the running time of $f(m,n)$ and let $G(m,n)$ be the running time of $g(m,n)$.

**Task 1.1** (20 pts). **Prove or disprove the following statements.**

a) $mn+1 \in O_\forall(mn)$

b) $mn+1 \in O_\exists(mn)$

c) If $G(m,n) \in O_\forall(nm)$ then $F(m,n) \in O_\forall(m^2 n)$

d) If $G(m,n) \in O_\exists(mn)$ then $F(m,n) \in O_\exists(m^2 n)$

e) Let $f_1, f_2 : \mathbb{N}^2 \to \mathbb{R}_{\geq 0}$. If there exists $N \in \mathbb{N}$ such that $f_1(n_1,n_2) \leq f_2(n_1,n_2)$ for all $n_1, n_2 \geq N$ then $f_1(\vec{n}) \in O_\forall(f_2(\vec{n}))$.

f) Let $f_1, f_2 : \mathbb{N}^2 \to \mathbb{R}_{\geq 0}$. If there exists $N \in \mathbb{N}$ such that $f_1(n_1,n_2) \leq f_2(n_1,n_2)$ for all $n_1, n_2 \geq N$ then $f_1(\vec{n}) \in O_\exists(f_2(\vec{n}))$.

*Bonus questions:* Which of the two definitions is more useful for analyzing programs? Can you define a better generalization of the univariate Big-O notation?

## 2 Recurrence Relations

In this course, we are interested in the exact (non-asymptotic) resource consumption of programs. Moreover, we are interested in automating the reasoning about complexity and proving the soundness of the results. Classical techniques for achieving this goal rely on deriving recurrence relations from programs that describe the cost as well as the sizes of the results. For the purpose of this problem, a recurrence relation is a recursive function $T : \mathbb{N}^n \to \mathbb{N}$.

In lecture, we have discussed how to systematically (and potentially automatically) solve *homogeneous linear recurrence relations with constant coefficients*. Moreover, we have seen how we can reduce solving non-homogeneous recurrence relations to solving homogeneous recurrence relations.

**Task 2.1** (10 pts). Using the aforementioned technique, derive a closed form for the following recurrence relation.

$$\begin{aligned} T(0) &= 10 \\ T(n) &= T(n-1) + 12n + 3 \quad \text{for n>0} \end{aligned}$$

More specifically:

1. Convert the recurrence into a homogeneous recurrence that has the same closed-form solution.
2. Write down the characteristic polynomial of the homogeneous recurrence.
3. Determine the roots of the characteristic polynomial.
4. Derive the solution from the roots using the formula from lecture.

Unfortunately, there is no general recipe for finding exact closed forms for recurrence relations. Most well-known techniques only work for finding asymptotic bounds. Often, the best way of solving a recurrence relation is to guess a solution or a template solution and verify it by substituting it into the recurrence. For example, you might conjecture that the solution to the previous recursion is of the form $c_2 n^2 + c_1 n + c_0$. Then you obtain the equations

$$\begin{aligned} c_2 0^2 + c_1 0 + c_0 &= 10 \\ c_2 n^2 + c_1 n + c_0 &= c_2(n-1)^2 + c_1(n-1) + c_0 + 12n + 3 \quad \text{for n>0} \end{aligned}$$

which you can use to solve for the constants.

**Task 2.2** (15 pts). Find exact (non-asymptotic) closed forms for the following recurrence relations and show that your solutions satisfy the equalities.

a) $R(m,n) = 0$ for $n < 4$ and $R(m,n) = R(m,n-4) + 12(m + \binom{m}{2})$ for $n \geq 4$

b) $T_1(0) = T_2(0) = 0$, $T_1(n) = T_2(n-1) + 20(n-1)$, and $T_2(n) = T_1(n-1) + 8n - 2$ for $n > 0$

c) $S(1) = 0$, $S(2) = 1$, and $S(n) = S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) + n - 1$ for $n > 0$

You can earn partial points for upper bounds.

It is often not trivial to derive recurrence relations from source code. One of the issues that is sometimes overlooked is that you often need to calculate the size of data structes and the value of integers that are returned by auxiliary functions. Another problem is the treatment of nested data structures like lists of lists that need to be abstracted with a finite number of natural numbers. Finally, higher-order arguments can usually not be abstracted by a number.

Consider for example, the implementation of insertion sort in OCaml below. The function *isort_list* sorts of lists lists in lexicographical order. You will use recurrence relations to derive an asymptotic worst-case bound for *isort_list* in the same way we derived a bound for quick sort in the lecture.

```
let rec compare_list l1 l2 =
  match l1 with
  | [] → true
  | x::xs →
     match l2 with
     | [] → false
     | y::ys →
        if x = y then
          compare_list xs ys
        else
          x < y
```

```
let rec insert le x l =
  match l with
  | [] → [x]
  | y::ys →
     if le y x then y::insert le x ys
     else x::y::ys

let rec isort le l =
  match l with
  | [] → []
  | x::xs → insert le x (isort le xs)

let isort_list = isort compare_list
```

*Cost model:* Derive a bound on the *number of function calls* that are performed by a funcstion. Constructor calls such as :: do not count as function calls.

*Higher-order functions:* It is up to you how to treat higher-order functions. One possibility is to use higher-order recurrence relations like $T(n, f) = f(n)$. Another possibility is to simply analyse first-order functions such as *insert compare_list* only.

*Polymorphism:* In OCaml, *isort_list* has the type *'a list list → 'a list list*. Assume for the analysis the type *int list list → int list list*. In particular, you can account cost 1 (1 function call) for the operations < and = in the function *insert*.

**Task 2.3** (20 pts).

a) Define a system of recurrence relation so that there is one recurrence relation for *isort_list* and for every function that can called by *isort_list*. *Hint:* You also need to define recurrence relations that describe the sizes of the returned values of the functions.

b) Solve the system of recurrence relations in a bottom-up manner.

# 3 Amortized Analysis (Functional Queue)

A simple and elegant way of implementing a queue in a functional programming language is to use two lists.

```
type 'a queue = Queue of 'a list * 'a list

let enqueue (Queue (inq, outq)) x =
  Queue (x::inq, outq)

let rec rev_append l1 l2 =
  match l1 with
  | [] → l2
  | x::xs → rev_append xs (x::l2)
```

```
let rev l = rev_append l []

let rec dequeue (Queue (inq, outq)) =
  match outq with
  | [] →
     begin
       match inq with
       | []  → None
       | x::xs → dequeue (Queue ([],rev inq))
     end
  | y::ys → Some (Queue (inq, ys), y)
```

**Task 3.1** (4 pts). Briefly describe the idea of the *queue* data structure.

**Task 3.2** (6 pts). Determine how many *cons* operations (uses of *::*) are performed by *dequeue* in the worst-cose? *Hint:* We are looking for the exact (non-asymptotic) number of operations.

Next consider a sequence of *n enqueue* or *dequeue* operations of the form

```
let b1 in
let b2 in
...
let bk in
()
```

where

$b_i \equiv$ `q = enqueue q` $n_i$ for an integer $n_i$ or $b_i \equiv$ `(q,_) = dequeue q`

**Task 3.3** (12 pts). Use the potential method to determine the amortized number of *cons* operations that is performed per operation. *Hint:* Clearly define the potential function that you are using. Again, we're looking for the exact number.

Finally, consider the following OCaml expression.

```
let q = Queue ([3;2;1],[]) in
let (_,_) = dequeue q in
let (_,_) = dequeue q
```

**Task 3.4** (8 pts). How many *cons* operation are performed if the above code is evaluated? What program property is necessary to make the analysis you performed in Task 3.3 applicable?

A prominent application a functional queue is a breadth-first search (BFS) for trees. Below we use the implementation of a queue from Problem 3 to implement BFS for binary trees in OCaml.

```
let bfs f t =

  let rec bfs_aux queue =
    match dequeue queue with
    | None →
       None
    | Some (queue, t) →
       begin
         match t with
         | Leaf → bfs_aux queue
         | Node(x,t1,t2) →
            if f(x) then
              Some (Node(x,t1,t2))
            else
              bfs_aux (enqueue_list queue [t1;t2])
       end
  in

  bfs_aux (enqueue (Queue ([], [])) t)
```

**Task 3.5** (10 pts). Determine the exact worst-case number of cons operation that is performed by the function *bfs_aux*. Justify your answer (a proof is not required).

Some approaches to automatic complexity analysis rely on deriving and solving recurrence relations. Try to describe the complexity of the function *bfs_aux* with a recurrence relation. (Feel free to submit your solution if you think that you found one.)