EECS 340 Algorithms

2018 Spring Semester

Homework 1

Due on January 31, 2018 before class (12:45pm or 7:00pm)

The first assignment covers primitive operation counts and asymptotic notations (big-Oh and relatives), together with their interpretations.

1. **Warm-Up: Big-Oh and Counting Primitive Operations**
   Show your work on the following questions. Use the limit-based definitions of asymptotic notation on the "Big-Oh Cheat Sheet" on Canvas wherever applicable.
   (a) Solve R-1.20 ($\star\star$) , R-1.22 ($\star$) , and R-1.23 ($\star$) in the text.
   (b) Intuitively, $2^x \in O(3^x)$, since 3^x grows faster. Is $3^x \in O(2^x)$? ($\star$)
   (c) Intuitively, $log_3(x) \in O(log_2(x))$. Is $log_2(x) \in O(log_3(x))$? ($\star$)
   (d) For each of the following, write down each algorithm's runtime as a (possibly-nested) summation, assuming that every arithmetic operation ($*, +$) and assignment operation ($\leftarrow$) takes one unit of time. (For the purposes of this question, ignore the time it takes to initialize, increment and test against the upper bound in a ranged `for` loop) Then, use these summations to derive tight asymptotic bounds ($\Theta(-)$) on the runtime of each algorithm.
      i. R-1.12 in the text (Loop2). ($\star$)
      ii. R-1.14 in the text (Loop4). ($\star\star$)
      iii. R-1.15 in the text (Loop5). ($\star\star$)
   (e) Explain why it was reasonable to ignore the overhead of a ranged `for` loop (initialization, increments, and tests) when you derived the tight asymptotic runtime bounds in the previous question. ($\star$)

2. **A Challenging Sum** Solve C-1.22 in the text. ($\star\star\star$)
   (**Alternate Hint:** Theorem A.6 (in the Appendix) may help.)

3. **Theory: Big-Oh and Derivatives**
   In the following questions, suppose that $f, g : \mathbb{R} \rightarrow \mathbb{R}$ are differentiable and strictly increasing ($f'(x) > 0$ and $g'(x) > 0$ for all x). Prove the statement in each question, or construct a counterexample.

(a) Is $f(x) \in O(g(x))$ if and only if $f'(x) \in O(g'(x))$? $(\star\star)$
(**Hint:** Use the limit definition of $O(-)$ notation)

(b) Is it true that if $\lim_{x \to +\infty} f'(x) = 0$, then $f(x) \in O(1)$? $(\star)$

4. **Theory: Properties of Big-Oh Notation**

   (a) Let $f, g : \mathbb{R} \to \mathbb{R}$ be continuous and strictly increasing. Is it necessarily the case that $f \in O(g(x))$ or $g \in O(f(x))$? Prove, or provide a counterexample. $(\star\star\star)$
   (**Hint:** Write out the limit definitions of $f \in O(g(x))$ and $g \in O(f(x))$)

   (b) Solve R-1.18 in the text. $(\star)$

   (c) Solve R-1.19 in the text. $(\star\star)$

5. **Application: Matrix Multiplication**

   (a) Consider the code below for `easy-multiply`, which computes the matrix product of two $n \times n$ square matrices of floating-point numbers. Suppose that all primitive operations $(*, +, \leftarrow, ...)$ are $\Theta(1)$ operations. Provide a tight asymptotic upper bound on the runtime of `easy-multiply` in terms of n. $(\star)$

Algorithm 1: `easy-multiply`(A, B)

Data: Two $n \times n$, 2-D Arrays A and B of floating point numbers
Result: The matrix product AB
Result $\leftarrow$ an $n \times n$ matrix of zeroes (Note: This operation takes $\Theta(n^2)$ time)
for *i = 1 to n* **do**
  for *j = 1 to n* **do**
    for *k = 1 to n* **do**
      Result[i][j] $\leftarrow$ A[i][k] * B[k][j]
    end
  end
end
return Result

   (b) *Strassen Matrix Multiplication* is an algorithm for matrix multiplication which achieves a runtime asymptotically bounded by $O(n^{2.807})$ on $n \times n$ matrices. Despite this good runtime bound, this algorithm is slower than `easy-multiply` on small inputs. Call the routine for Strassen matrix multiplication `s-multiply`. Consider the following algorithm for matrix multiplication:

Algorithm 2: `combined-multiply`(A, B)

Data: Two $n \times n$, 2-D Arrays A and B of floating point numbers
Result: The matrix product AB
if $n < 100$ **then**
  return `easy-multiply`(A, B)
else
  return `s-multiply`(A, B)
end

What is the asymptotic runtime of `combined-multiply` in terms of n? $(\star)$

6. **"Application": Spaghetti Sort**

   Spaghetti Sort is a custom-designed sorting algorithm for meticulous chefs inspired by the process of sorting uncooked spaghetti by length as follows:

> Recipe for Sorting Spaghetti:
>
> (a) Grab all noodles in one of your hands.
>
> (b) Hold your hand above a flat surface like a table.
>
> (c) Bring your hand down so that you can make the spaghetti bottoms level.
>
> (d) Hover your other hand above the noodles.
>
> (e) While there are still uncooked noodles:
>
>   i. Lower your non-spaghetti hand at constant velocity until it hits a noodle (this must be the longest one.)
>
>   ii. Remove that noodle using that hand.
>
>   iii. Place it at the beginning of your noodle list.
>
>   iv. Return your hand to the position it was in when it hit a noodle.

If we imagine a different version of spaghetti sort in which, after ensuring the bottoms of the noodles were level, we lowered the noodles into a vat of acid (gloves required!) until they were completely disintegrated [1], we might feel inspired to port this algorithm to pseudocode as follows:

Algorithm 3: `spaghetti-sort`(L)

Data: An unsorted linked-list L of natural numbers
Result: L, but sorted in ascending order
Result ← a new, empty linked-list
counter ← 0
while *L is not empty* **do**
    for *each node n with value $x \in \mathbb{N}$, in the standard linked-list iteration order* **do**
        if *x == 0* **then**
            Append the value of counter to the end of Result
            Remove node n from the linked-list
            Continue the "**for**" loop from the node which came immediately after n
        else
            x ← x - 1 (Subtract one from the element in the list)
        end
    end
    counter ← counter + 1
end
return *Result*

[1] Reconstruction of disintegrated noodles is left as an exercise to the interested reader ($\star\star\star\star\star\star\star$)

(a) Describe the asymptotic runtime of unmodified `spaghetti-sort`. ($\star\star$)

(b) Suppose that we require that the input list is *not* a list of arbitrary natural numbers, but is instead a list of 64-bit unsigned integers. What is a tight bound on the asymptotic runtime of spaghetti sort now? ($\star$)

(c) In light of the answer to the previous question, why/why not would we want to use `spaghetti-sort` for sorting 64-bit unsigned integers in practice? ($\star$)

7. **Algorithm Design: Finding Cycles**
Give pseudocode for an algorithm which takes as input a singly-linked-list [2] (of unknown size, possibly containing loops) and returns $True$ if the singly-linked-list has a loop, and $False$ if the singly-linked-list has no loops. To simplify things, you may assume that each node of the linked-list stores an integer value, and the integers stored at each node of the linked list are distinct. If there are n distinct nodes [3] in the input, show that your algorithm has a worst-case asymptotic runtime bounded by $O(n)$ and worst-case asymptotic space usage bounded by $O(1)$. ($\star\star\star$)

Submission

Hand in your paper in-class by the beginning of your section of the class on the due date. Always check Canvas for updates and corrections.

[2]Here, "singly-linked list" is used in the colloquial sense, and refers to a structure consisting of nodes together with pointers from each node to the next node. Strictly speaking, this does not need to be a *list* (definable as a function from a finite enumerated set of indices to some other set), precisely because of the possibility of pointer cycles.

[3]Note: You should read this as "If there is a cycle, count the nodes in a cycle only once." E.g: If you have a linked list $5 \to -6 \to 2 \to 9 \to -6 \to 2 \to 9 \to -6 \to 2 \to 9$... there are only four distinct nodes (The nodes storing $5, -6, 2, 9$) ($n = 4$)