CSCI 0500: Data Structures, Algorithms, and Intractability (Fall 2025)

Assignment 1

Due at 11:59pm ET, Sunday, Oct 5

1. (1 point) There are several ways to define asymptotic notations such as big-O.

   Definition 1 (Simplified big-O in lecture)**.** For two functions f and g that map positive integers to positive real numbers, we say $f = O(g)$ if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq 1$.

   This definition is sufficient in most cases. However, sometimes we want to use big-O notation for functions that are not positive everywhere. For example, a function $T(n)$ with $T(1) = 0$, or a function like $\ln(\ln(n))$ that is negative when $n \leq 2$.

   In this question, we introduce more general definitions of big-O and big-Omega:

   Definition 2 (Big-O)**.** For two functions f and g that map positive integers to real numbers, we say $f = O(g)$ if there are constants $c > 0$ and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

   Definition 3 (Big-Ω)**.** For two functions f and g that map positive integers to real numbers, we say $f = \Omega(g)$ if there are constants $c > 0$ and n_0 such that $f(n) \geq c \cdot g(n) \geq 0$ for all $n \geq n_0$.

   Arrange the following functions in ascending order of their asymptotic growth rate: If $g(n)$ immediately follows $f(n)$ in your list, it should hold that $f(n) = O(g(n))$. (You do not need to prove each $f(n) = O(g(n))$ but should know how to do so using the more general definitions.)

   $$
   \begin{aligned}
   f_1(n) &= \tfrac{1}{10}\, n^2 \ln n \\
   f_2(n) &= 2^n \\
   f_3(n) &= \ln^3 n = (\ln n)^3 \\
   f_4(n) &= \ln \ln n = \ln(\ln(n)) \\
   f_5(n) &= 5 \cdot n^{4/3} \\
   f_6(n) &= n!
   \end{aligned}
   $$

   Remark. Think of $O(\cdot)$ as adjectives. Just as we might say that "a tree is green", we say "f is $O(n)$". We do not say "green is a tree" or "$O(n)$ is f".

   The equal sign in "$f = O(n)$" should be viewed as shorthand for "is" rather than "equals". Therefore, "$O(n) = f$" is nonsense.

   Be sure to use uppercase O and Ω. The lowercase versions mean something different.

2. (1 point) In this question, we explore non-comparison-based sorting algorithms.

   Consider sorting an n-element array X, where each element is an integer in $\{0, 1, \ldots, n-1\}$ (possibly with duplicates). Consider the following algorithm `sort1`.

```
def sort1(X):
  n = len(X)
  count = [0 for i in range(n)]
  for a in X:
    count[a] += 1
  S = []
  for i in range(n):
    for j in range(count[i]):
      S.append(i)
  return S
```

   For this problem, we measure computational cost (i.e., runtime) by the number of array entries created plus the number of array accesses (for all three arrays `X`, `S`, and `count`).

   Prove that the worst-case runtime of `sort1` is $\Theta(n)$.

   Remark. (This is not a question.) How can we relax the restriction on the values in X? Consider sorting an n-element array X, where each element is an integer in $\{0, 1, \ldots, n^2 - 1\}$. Each element can be viewed as a two-digit number in base n. Intuitively, we can invoke `sort1` twice: sort the elements first by the lower digit and then by the higher digit. This requires a slight modification to `sort1` to enable a key-based *stable* sort (the relative order of elements with the same key is preserved). The runtime is still $\Theta(n)$.

3. (1 point) In this question, we consider two applications of sorting and selection.

   (a) Consider the problem of finding the t largest elements in an array X.

   Given a 0-indexed n-element array X and an integer t (where $1 \le t \le n$), the goal is to return the t largest elements in X (in any order). The computational cost (i.e., runtime) is measured by the number of pairwise comparisons between elements in X.

   Design a randomized algorithm with worst-case expected runtime $O(n)$ and prove that it achieves this runtime. You can assume that all elements in X are distinct.

   (Hint: You can use the `quickselect(X, k)` algorithm discussed in class: For any X, `quickselect(X, k)` runs in expected time $O(n)$ and returns `sorted(X)[k]`, the $(k+1)$-th smallest element in X. You do not need to implement it or prove its runtime.)

   (b) Consider the problem of deciding whether an element appears in a sorted array.

   Given a 0-indexed array S with n elements in ascending order and another element v, the goal is to return 1 if v appears in S and 0 otherwise. The runtime is measured by the number of pairwise comparisons between v and elements in S.

   Design a deterministic algorithm with worst-case runtime $O(\log n)$ and prove that it achieves this runtime. You can assume that all elements in S are distinct.