Worksheet on Algorithm analysis and Big O

Benjamin Cosman, Patrick Lin and Mahesh Viswanathan

Fall 2020

TAKE-AWAYS

- $f(n)$ is $O(g(n))$ if there are positive c, k such that for every $n \geq k$, $f(n) \leq c \cdot g(n)$.
- Informally, $f(n)$ is $O(g(n))$ if, as long as you ignore small inputs and constant factors, $f(n) \leq g(n)$.
- The following simple functions form a strictly increasing hierarchy for big-O:
  $1, \log(n), n, n\log(n), n^2, n^3, ..., 2^n, 3^n, ..., n!$.
- We say $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$
- Dominant Term Method: If $f(n) = g_1(n) + g_2(n) + ... + g_k(n)$ and $g_1(n)$ is the dominant term (grows the fastest for large n; all other $g_i(n)$ are $O(g_1(n))$), then $f(n)$ is $\Theta(g_1(n))$
- The run time of an algorithm is expressed as a function $T(n)$ whose value is the worst-case run time for inputs of size n.
- Usually, an algorithm with (nested) loops will have a run time that is most easily expressed as a summation and a recursive algorithm will have a run time most easily expressed as a recurrence. Both can be simplified to closed forms (e.g. with unrolling or recursion trees), which are then frequently simplified further using big-O notation.

Problem 1.

a) Prove that 2^n is $O(n!)$. *(Apply the definition of big-O; do not just appeal to our hierarchy of common functions.)*

b) Prove or disprove the following: If $f(n)$ is $O(2^n)$ and $g(n)$ is $O(n!)$ then $f(n)$ is $O(g(n))$

Problem 2. In lecture we saw two sorting algorithms with (worst-case) run times $\Theta(n^2)$ and $\Theta(n \log(n))$ (where n is the length of the input array). A friend tells you that they've heard of an even better algorithm with (worst-case) run time $\Theta(\log(n))$. Convince your friend that such an algorithm cannot exist. [1]

[1] Hint: Come up with some bare minimum amount of work that every sorting algorithm must do.

Problem 3. In this problem we partially justify the Dominant Term Method by proving that it works for a sum of two terms. We first also show that it does not work for products, only sums.

a) Let $f(n) = g(n) \cdot h(n)$, where $g(n)$ is a "dominant factor" in the product, i.e. $h(n)$ is $O(g(n))$. Prove that $f(n)$ is not necessarily $O(g(n))$.

b) Let $f(n) = g(n) + h(n)$, where $g(n)$ is a dominant term in the sum, i.e. $h(n)$ is $O(g(n))$. Prove that $f(n)$ is $O(g(n))$.

Problem 4. Consider the following algorithm, presented in pseudocode, which multiplies two nxn matrices that are in row-echelon form[2]:

Row-Echelon Square Matrix Multiplier

```
1. RESMatrixMul(A, B):
2.   answer = nxn all-zeros matrix
3.   for each i from 1 to n:
4.     for each j from i to n:
5.       for each k from i to j:
6.         answer[i][j] += A[i][k] * B[k][j]
7.   return answer
```

[2] The only important fact about row-echelon form is that entries below the diagonal are all zeros, which allows us to optimize the for-loop bounds instead of making them all 1 to n.

a) What is the run time in terms of n? You may assume that line 6 takes a constant amount of time, and the run time of all other lines is negligible.

b) The way we define the "size" of the input will affect the expression we compute for the run time. What is the run time of RESMatrixMul in terms of r, where $r = n^2$ is the *number of elements* in each of the two input matrices?