CS 341: ALGORITHMS (S18) — LECTURE 2
BIG-O NOTATION AND REDUCTIONS

ERIC BLAIS

1. Big-O notation

We ended the first lecture with definitions for algorithms, for solving a problem, and for the model of computation (Word RAM) that we will use to measure the run time of an algorithm on a given input. The next ingredient that we need to determine the time complexity of an algorithm is a decision on how to measure this complexity *globally* instead of on a per-instance basis.

Definition 2.1 (Worst-case time complexity)**.** The *worst-case time complexity* of an algorithm A is the function $T_A : \mathbb{N} \to \mathbb{N}$ obtained by letting $T_A(n)$ be the maximum time complexity of A over any input of size n.

Remark 2.2. Worst-case is not the only way we can measure the time complexity of an algorithm as a function of its input size. (One could take the *average* time complexity over some distribution on inputs of length n instead, for example). But this is a model that is particularly useful and the one we will focus on throughout this course.

We now come to one of the central ideas in the analysis of algorithms: what we care about, when we measure the time complexity of an algorithm, is not the *exact* expression for this time complexity, but rather its *asymptotic* rate of growth as the inputs get larger. This is best done using big-O notation.

Definition 2.3 (Big-O notation)**.** Two functions $f, g : \mathbb{N} \to \mathbb{R}^{\geq 1}$ satisfy $f = O(g)$ if there exist $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, we have $f(n) \leq c\, g(n)$.

Remark 2.4. Here and throughout,

- $\mathbb{N} = \{1, 2, 3, \ldots\}$ is the set of natural numbers,
- $\mathbb{R}^+$ be the set of positive real numbers, and
- $\mathbb{R}^{\geq 1}$ be the set of real numbers that have value at least 1.

Big-O notation is so useful in part because it lets us simplify even very complicated expressions and only worry about the "most significant" aspects of time complexity. For example, we have the following fact.

Proposition 2.5. *The function $f : n \mapsto 4n^7 + 100n^3 + \frac{1}{3}n^2 + \pi$ satisfies $f = O(n^7)$.*

Proof. For every $n \geq 4$,

$$f(n) = 4n^7 + 100n^3 + \tfrac{1}{3}n^2 + \pi \leq 4n^7 + 100n^3 + n^2 + n \leq 106n^7$$

so $f = O(n^7)$ by the definition with $c = 106$ and $n_0 = 4$. □

One word of warning: since $n^7 \leq n^{100}$ for every $n \geq 1$, it is also correct to say that the function f defined in the proposition satisfies $f = O(n^{100})$. To describe asymptotics of a function more precisely, we need the big-Ω and the big-Θ notation. The big-Ω notation is used to give lower bounds on the asymptotic growth of a function.

Definition 2.6 (Big-Ω notation)**.** Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ satisfy $f = \Omega(g)$ if there exist $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, we have $f(n) \geq c\, g(n)$.

The big-Θ notation is used to show that we have matching upper and lower bounds on the asymptotic growth of a function.

Definition 2.7 (Big-Θ notation)**.** Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ satisfy $f = \Theta(g)$ if and only if $f = O(g)$ and $f = \Omega(g)$.

We can use this notation to strengthen our last proposition.

Proposition 2.8. *The function* $f : n \mapsto 4n^7 + 100n^3 + \frac{1}{3}n^2 + \pi$ *satisfies* $f = \Theta(n^7)$.

Proof. We have already seen that $f = O(n^7)$. We also have that for every $n \geq 1$, $f(n) \geq 4n^7$ so by definition $f = \Omega(n^7)$ (with $n_0 = 1$ and $c = 4$) and, therefore, $f = \Theta(n^7)$ as well. □

There are also situations where we want to argue that a function grows *asymptotically slower* than another reference function. We can state this formally with the little-o notation.

Definition 2.9 (little-o notation)**.** Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ satisfy $f = o(g)$ if for every $c \in \mathbb{R}^+$, there exists $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, we have $f(n) < c\, g(n)$.

Remark 2.10. Equivalently, $f = o(g)$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. This definition can be easier to work with; feel free to do so.

Similarly, we can say that a function f grows asymptotically *faster* than another function g using the little-ω notation.

Definition 2.11 (little-ω notation)**.** Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ satisfy $f = \omega(g)$ if and only if $g = o(f)$.

We will be using this notation extensively throughout the course, so it is critical that you are all very comfortable working with it. We will be covering some exercises related to this notation in this week's tutorial and in the first assignment, but you can also find much more information about it and many other exercises to work on in the textbook.

2. The 2SUM problem

Now that we have all our definitions in place, let's put them to practice.

Definition 2.12 (2SUM)**.** An instance of the *2SUM problem* is an array $A \in \mathbb{Z}^n$ of n integers and an integer $m \in \mathbb{Z}$. The valid solution to an array A is True if there exist 2 indices $i, j \in \{1, 2, \ldots, n\}$ such that $A[i] + A[j] = m$, and False otherwise.

The following algorithm solves the 2SUM problem.

Algorithm 1: SIMPLE2SUM $(A[1..n], m)$

```
for i = 1, ..., n − 1 do
    for j = i, ..., n do
        if A[i] + A[j] = m return True;
return False;
```

The claim that SIMPLE2SUM solves the 2SUM problem requires a (simple) proof. But let's focus on the time complexity of this algorithm.

Proposition 2.13. *The* SIMPLE2SUM *algorithm has time complexity* $\Theta(n^2)$.

Proof. What we really care about in terms of time complexity for this problem is the number of basic operations (addition, incrementing, and comparison) on integers. We do this in the Word RAM model by setting the width w of words to be large enough that each integer in a problem instance can be stored with w bits.

And we measure time complexity in the *worst-case*, so we consider input arrays for which the inner loop is not interrupted by a "return True" command. In this case, the number of additions performed by the algorithm is

$$n + (n-1) + \cdots + 2 + 1 = \frac{n(n-1)}{2}.$$

For every $n \geq 2$, we have $\frac{n^2}{4} \leq \frac{n(n-1)}{2} \leq n^2$ so the algorithm performs $\Theta(n^2)$ additions and the same bound also holds for the total number of increments and compare operations. □

With this analysis in hand, we can now answer the central question in algorithm design: can we do better? Indeed we can! Let's break down what the algorithm is doing in detail. For each index $i \in \{1, 2, \ldots, n\}$, the inner loop is trying to determine if there is an integer $A[j]$ in A for which $A[i] + A[j] = m$, or, to rephrase the same statement: if the integer $m - A[i]$ is in the array.

As it turns out, we know a very efficient method for determining whether an integer (such as $m - A[i]$) is in an array: binary search! This, of course, only works when the array is sorted, but that's something we can do in the algorithm as well. The resulting algorithm is as follows.

Algorithm 2: 2SUM $(A = (A[1..n], m)$

```
SORT(A);
for i = 1, ..., n − 1 do
    if FIND(A, m − A[i]) then return True;
return False;
```

What we have just done looks simple, perhaps even obvious, yet it is a great example of an extremely powerful algorithmic technique: reduction.

Reduction idea: Use known algorithms to solve new problems.

We'll see many other examples where the reduction technique proves very useful—and it will even return with a starring role in the last module of this class, when we tackle

NP-completeness. But for now, let's continue our work and establish the correctness and time complexity of the 2SUM algorithm.

Proposition 2.14. *The* 2SUM *algorithm solves the 2SUM problem.*

Proof. Consider first the case where there exist indices i^*, j^* for which $A[i^*] + A[j^*] = m$. We can rewrite the identity as $A[j^*] = m - A[i^*]$, so for the iteration of the for loop where $i = i^*$, the integer $m - A[i]$ is in A, which means that the FIND algorithm returns True and so does 2SUM.

Consider now the case where for every indices i, j we have $A[i] + A[j] \neq m$. then for every index i, we have that for every index j, $A[j] \neq m - A[i]$ or, in other words, the integer $m - A[i]$ is not in A and FIND returns False. Therefore, 2SUM also correctly returns False. □

Proposition 2.15. *When* SORT *has time complexity* $\Theta(n \log n)$ *and* FIND *is a binary search algorithm with time complexity* $\Theta(\log n)$*, the* 2SUM *algorithm has time complexity* $\Theta(n \log n)$*.*

Proof. The initial call to SORT has time complexity $\Theta(n \log n)$ and the total time complexity of the for loop, in the worst case, is $n \cdot \Theta(\log n) = \Theta(n \log n)$. □

Can we do even better? It takes $\Theta(n)$ time just to read the integers stored in the array A, so the best we can probably hope for is to remove the extra $\log n$ term in our runtime. But instead of trying to do this, let's see if we can use the Reduction idea to solve even more complex problems.

3. The 3SUM problem

Definition 2.16 (3SUM)**.** An instance of the *3SUM problem* is an array $A \in \mathbb{Z}^n$ of n integers and an integer m. The valid solution to an array A is True if there exist 3 indices $i, j, k \in \{1, 2, \ldots, n\}$ such that $A[i] + A[j] + A[k] = m$, and False otherwise.

We can again define a simple algorithm to solve the 3SUM problem.

Algorithm 3: SIMPLE3SUM $(A[1..n], m)$

```
for i = 1, ..., n do
    for j = i, ..., n do
        for k = j, ..., n do
            if A[i] + A[j] + A[k] = m return True;
return False;
```

This algorithm is not so efficient.

Proposition 2.17. *The* SIMPLE3SUM *algorithm has time complexity* $\Theta(n^3)$*.*

Proof. The total number of additions performed by the algorithm is

$$\sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=j}^{n} 2 = \Theta(n^3)$$

and this is also asymptotically the total number of elementary operations performed by the algorithm. □

Let's again try to use the Reduction technique to improve our algorithm. We can perhaps see how again an efficient FIND algorithm could help us. Or, since we just designed an efficient 2SUM algorithm, perhaps that algorithm can help us? Let's find out by again fixing an index i and seeing what the two inner loops are doing. They're trying to find out if there are indices j, k for which

$$A[i] + A[j] + A[k] = m \quad \iff \quad A[j] + A[k] = m - A[i].$$

This is an instance of the 2SUM problem!

Algorithm 4: 3SUM $(A[1..n], m)$

for $i = 1, \ldots, n$ **do**
  if 2SUM $(A, m - A[i])$ then **return** True;
return False;

And the analysis of the time complexity is again simple, yielding the following result.

Theorem 2.18. *The 3SUM problem can be solved by an algorithm with time complexity* $\Theta(n^2 \log n)$.

Proof. When the 3SUM algorithm calls the algorithm 2SUM with time complexity $\Theta(n \log n)$ that we designed in the previous section, its total time complexity is $n \cdot \Theta(n \log n) = \Theta(n^2 \log n)$. □

Can you do even better? With some work, you may be able to design an algorithm that solves the 3SUM problem in time $\Theta(n^2)$. Determining whether we can do even better is one of the major open problems in algorithms research today: it was only very recently that researchers were able to show that 3SUM can be solved by an algorithm with time complexity $o(n^2)$, and whether or not there is an algorithm that solves 3SUM with time complexity $O(n^\gamma)$ for some $\gamma < 2$ remains unknown.