CS335 Class Notes

Luis F. Uceta

2020-01-28

Contents

1 Lecture 3: Algorithm analysis

An **algorithm** is a clearly defined set of simple instructions which must are to be followed in order to solve a particular problem.

1.1 Mathematical background

Algorithm analysis is grounded on mathematics and the definitions below set up a formal framework to study algorithms. These definitions try to establish a **relative order among functions**. Given two functions, there are usually points where one function is smaller than the other so it doesn't make sense to claim, for instance, $f(N) < g(N)$. Instead, we compare their **relative rates of growth**.

Big-Oh $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq c \cdot f(N)$ when $N \geq n_0$.

Informally, the growth rate $T(N)$ is less than or equal to that $f(N)$.

Big-Omega $T(N) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) \geq c \cdot g(N)$ when $N \geq n_0$.

Informally, the growth rate $T(N)$ is greather than or equal to that $g(N)$.

Big-Theta $T(N) = \Theta(h(N))$ **if and only if** $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.

Informally, the growth rate $T(N)$ equals the growth rate of $h(N)$. This means that $T(N)$ is eventually *sandwiched* between two different constant multiples of $h(N)$.

Little-Oh $T(N) = o(p(N))$ if, for all positive constants c, there exist an n_0 such that $T(N) < c \cdot p(N)$ when $N > n_0$.

Informally, the growth rate $T(N)$ is less than the growth rate of $f(N)$. This is different from Big-Oh, given that Bog-Oh allows the possibility that the growth rates are the same.

NOTE: In the above definitions, both c and n_0 are constants and they cannot depend on N. If you see yourself saying "take $n_0 = N$" or "take $c = log_2 N$" in an alleged Big-Oh proof, then you need to start with choices of c and n_0 that are indepedent of N.

1.2 Notation

1.2.1 Big-Oh notation

When we say that $T(N) = O(f(N))$, we're guaranteeing that the function $T(N)$ grows at a rate no faster than $f(N)$; thus $f(N)$ is an **upper bound** on $T(N)$. Alternatively, we could say that $T(N)$ is

bounded above by a constant multiple of $f(N)$.

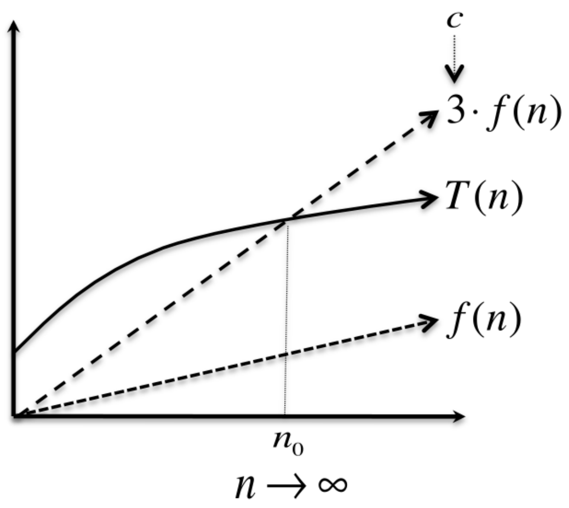


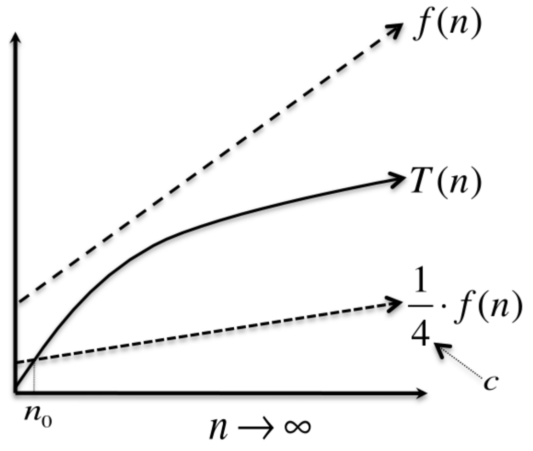
Figure 2.1: A picture illustrating when $T(n) = O(f(n))$. The constant c quantifies the "constant multiple" of $f(n)$, and the constant n_0 quantifies "eventually."

Figure 1: Big-Oh

For example, $T(N) = O(N^2)$ means that the function $T(N)$ grows at a rate no faster than N^2; its growth could equal that of N^2 but it could never surpass it.

1.2.2 Big-Omega notation

When we say that $T(N) = \Omega(g(N))$, we're guaranteeing that the function $T(N)$ grows at a rate no lower than $g(N)$; thus $g(N)$ is a **lower bound** on $T(N)$. Alternatively, we could say that $T(N)$ is **bounded below** by constant mutiple of $g(N)$.



$T(n)$ again corresponds to the function with the solid line. The function $f(n)$ is the upper dashed line. This function does not bound $T(n)$ from below, but if we multiply it by the constant $c = \frac{1}{4}$, the result (the lower dashed line) does bound $T(n)$ from below for all n past the crossover point at n_0. Thus $T(n) = \Omega(f(n))$.

Figure 2: Big-Omega

For example, $T(N) = \Omega(N^2)$ means that the function $T(N)$ grows at a rate no lower than N^2; its growth could equal that of N^2 but it could never drop below it.

Whenever talking about Big-Oh, there's always an implication about Big-Omega. For instance, $T(N) = O(f(N))$ (i.e., $f(N)$ is an **upper bound** on $T(N)$) implies that $f(N) = \Omega(T(N))$ (i.e., $T(N)$ is a **lower bound** on $f(N)$). As an example, N^3 grows faster than N^2, so we can say that $N^2 = O(N^3)$ or $N^3 = \Omega(N^2)$.

1.2.3 Big-Theta notation

When we say that $T(N) = \Theta(h(N))$, we're guaranteeing that the function $T(N)$ grows at the same rate as $g(N)$. However, when two functions grow at the same rate, then the decision of whether or not to signify this with $\Theta()$ can depend on the context.

1.3 Typical growth rates

Function	Name
c	Constant
$log_2 N$	Logarithmic
$log^2 N$	Log-squared
N	Linear
$N log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

1.4 Rules

Rule 1 If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then

1. $T_1(N) + T_2(N) = O(f(N) + g(N))$. Less formally it is $O(\text{max}(f(N), g(N)))$, and
2. $T_1(N) \cdot T_2(N) = O(f(N) \cdot g(N))$

Rule 2 If $T(N)$ is a polynomial of degree k, then $T(N) = \Theta(N^k)$.

Rule 3 $log^k N = O(N)$ for any constant k. In other words, logarithms run at a lower rate than linear functions which means logarithms grow very slowly.

1.5 Few points

- In simple terms, the goal of asymptotic notation is *to suppress constants factors and lower-order*. Thus, they aren't included in Big-Oh. For instance, don't say $T(N) = O(2N^2)$ or $T(N) = O(N^2 + N)$. In both cases, the correct form is $T(N) = O(N^2)$.

When analyzing the running time of an algorithm, why would we want to throw away information like constant factors and lower-order terms? 1) Lower-order terms become increasingly irrelevant as you focus in large inputs, which are the inputs that require algorithmic ingenuity and 2) constant factors are generally highly dependent on the details of the environment (e.g., programming language, architecture, compiler, etc,) and thus ditching them allows to generalize by not commiting ourselves to a specific programming language, architecture, etc.

- The relative growth rates of two functions $f(N)$ and $g(N)$ can always be determined by computing $lim_{N \to \infty} \frac{f(N)}{g(N)}$, using L'Hopital's rule if necessary. The limit can have four possible values:

  - The limit is 0, meaning that $f(N) = O(g(N))$.
  - The limit is $c \neq 0$, meaning that $f(N) = \Theta(g(N))$.
  - The limit if ∞, meaning that $g(N) = o(f(N))$.
  - The limit doesn't exist and thus no relation exist.

- Stylistically it's bad to say that $f(N) \leq O(g(N))$ since the inequality is already implied by the definition of Big-Oh.

- It's wrong to write $f(N) \geq O(g(N))$, because it doesn't make sense.

1.6 Model of computation

The model of computation is basically a normal computer with the following characteristics:

- Instructions are executed sequentially.
- The model has the standard repertoires of simple instructions, such as addition, multiplication, comparison, and assignment. Unlike real computers, this computer takes exactly one time unit to do anything.
- The computer has fixed-size (e.g., 32-bit) integers and no fancy operations (e.g., matrix invertion, sorting, etc.).
- The computer has infinite memory.

1.7 What to analyze

The most important resource to analyze is generally the *running time* and typically, the size of the input is the main consideration. We define two functions, $T_{avg}(N)$ (for the average-case running time) and $T_{worst}(N)$ (for the worst-case running time) used by an algorithm on input of size N. It's worth noting that $T_{avg}(N) \leq T_{worst}(N)$ (i.e., $T_{avg}(N)$ has a lower rate growth than that of $T_{worst}(N)$).

1.7.1 Types of performance

Best-case performance Although it can be ocassionally analyzed, it's often of little interest since it doesn't represent typical behavior.

Average-case performance It often reflects typical behavior, however it doesn't provide a bound for all input and it can be difficult to compute. Furthermore, it's also hard to define what's the average input

Worst-case performance It represents a guarantee for performance on any possible input. This is the quantity generally required because it provides a bound for all input.

1.8 Running-time calculations

When computing a Big-Oh running time, there are several general rules:

For loops The running time of a `for` loop is *at most* the running time of the statements inside the `for` loop times the number of iterations.

Nested loops Analyze these inside out. The total running tome of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

For example, the following program is $O(N^2)$ because the inner loop does N iterations and the outer loops does N too. This amounts to $N \times N = N^2$ iterations. The first assignment counts as one operation (thus $O(N)$) while the innermost statement counts for 2 operations (1 multiplication and 1 assignment). Thus, to be more precise the program's running time is $1 + 2N^2$, however Big-Oh suppresses constant factors and lower-order terms.

```
k ← 0
for [1, n] → i:
    for [1, n] → j:
        k ← i * j
```

Consecutive statements They just add (which means that the maximum is the one that counts).

For example, the following program fragment, which has $O(N)$ work followed by $O(N^2)$ work, is ultimately $O(N^2)$ which dominates $O(N)$:

```
for [0, n) → i:
    a[i] = 0

for [0, n) → i:
    for [0, n) → j:
        a[i] += a[j] + i + j
```

If/else For the fragment `if condition { S1 } else { S2 }`, the running time of an `if`/`else` statement is never more than the running time of the test plus the larger of the running times of `S1` and `S2`.

1.9 Sample problems

1.9.1 Sum of cubes

```
Input: a positive integer n.
Output: the sum of all cubes from 1 to n^3.

SumOfCubes(n):
    sum ← 0                 # O(1)
    for [1, n] → i:         # O(n)
        sum += i * i * i    # O(4), 1 assignment, 1 addition and 2 products
    return sum              # O(1)
```

Thus, for `SumOfCubes(N)= 1 + 4n + 1 = 4n + 2 = O(n)`. We discard both the constant factors and the lower-order terms.

1.9.2 Factorial

The factorial is defined as

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

Implemented recursively, the algorithm is as follows:

```
Input: a non-negative integer n
Output: the factorial of n

Factorial(n):
    if n ≤ 1:
        return 1
    else:
        return n * Factorial(n-1)
```

However, this is a thinly veiled **for** loop. In this case, the analysis involves the recurrence relation $T(n) = 1 + T(n-1)$ for $n > 1, T(1) = 2$ that needs to be solved:

$$\begin{aligned}
T(N) &= 1 + T(n-1) \\
&= 1 + (1 + T(-2)) \\
&= 1 + (1 + (1 + T(n-3))) \\
&= \ldots \\
&= 1 + (1 + (1 + T(n-k)))\ k \text{ 1's}
\end{aligned}$$

$$\begin{aligned}
T(N) &= k + T(n-k) \\
&= (n-1) + T(n-(n-1)) \\
&= (n-1) + T(1) = (n-1) + 2 = n + 1
\end{aligned}$$

Thus, $T(n) = O(n)$.

1.9.3 Maximum subsequence sum problem

> Given (possibly negative) integers $A_1, A_2, \ldots, A_N$, find the maximum value of $\sum_{k=1}^{i} A_k$.

In other words, given a one-dimensional array of numbers, find a contiguous subarray with the largest sum. For example, with the input $-2, 11, -4, 13, -5, -2$ the answer is $11 - 4 + 13 = 20$.

This problem is interesting mainly because there are many algorithms to solve it, and the performance of these algorithms varies drastically.

Input Size	Algorithm Time 1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 100$	0.000159	0.000006	0.000005	0.000002
$N = 1{,}000$	0.095857	0.000371	0.000060	0.000022
$N = 10{,}000$	86.67	0.033322	0.000619	0.000222
$N = 100{,}000$	NA	3.33	0.006700	0.002205
$N = 1{,}000{,}000$	NA	NA	0.074870	0.022711

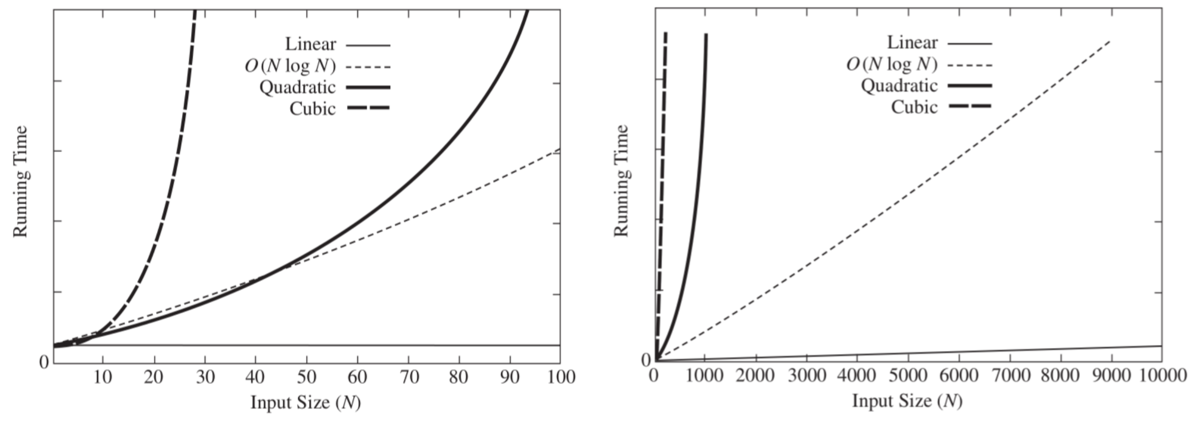


Figure 3: Plot (N vs. time) of various algorithms

For algorithm 4 (linear), as the problem size increases by a factor of 10, so does the running time. For algorithm 3 (quadratic), a tenfold increase in input size yields roughly hundrefold (10^2) increase in running time.

1.9.4 Algorithm 1 (brute force)

implementations/max-subsequence-sum:

```
Input: array A of integers.
Output: the maximum positive subsequence sum.

MaxSubsequenceSum(A):
    maxSum ← 0
    for [0, A.size) → i:
        for [i, A.size) → j:
            currentSum ← 0
            for [i, j] → k:
                currentSum += A[k]
            if currentSum > maxSum:
                maxSum = currentSum
    return maxSum
```

For this algorithm, there's N starting places, average $\frac{N}{2}$ lengths to check, and average $\frac{N}{4}$ numbers to add. Thus we have $O(N^3)$.

1.9.5 Algorithm 2 (brute force)

`implementations/max-subsequence-sum`:

```
Input: array A of integers.
Output: the maximum positive subsequence sum.

MaxSubsequenceSum( A ):
    maxSum ← 0
    for [0, A.size) → i:
        currentSum ← 0
        for [i, A.size) → j:
            currentSum += A[j]
            if currentSum > maxSum:
                maxSum = currentSum
    return maxSum
```

For this algorithm, the innermost **`for`** loop has been removed. There's N starting places and an average $\frac{N}{4}$ numbers to add. Thus we have $O(N^2)$.

1.9.6 Algorithm 3

`implementations/max-subsequence-sum`:

```
Input: array A of integers.
Output: find the maximum sum in subarray spanning A[LEFT..RIGHT]. It does
not attempt to mantain actual best sequence.

MaxSumRec( A, LEFT, RIGHT ):
    # Base case
    if LEFT = RIGHT:
        if A[LEFT] > 0:
            return A[LEFT]
        else
            return 0

    center ← (LEFT + RIGHT) div 2
    maxLeftSum ← MaxSumRec(A, LEFT, center)
    maxRightSum ← MaxSumRec(A, center + 1, RIGHT)

    maxLeftBorderSum ← 0
    leftBorderSum ← 0

    for [center, LEFT] → i:
        leftBorderSum += A[i]
        if leftBorderSum > maxLeftBorderSum:
            maxLeftBorderSum ← leftBorderSum

    maxRightBorderSum ← 0
    rightBorderSum ← 0

    for [center + 1, RIGHT] → j:
        rightBorderSum += A[j]
        if rightBorderSum > maxRightBorderSum:
            maxRightBorderSum ← RightBorderSum

    return max(maxLeftSum, maxRightSum, maxLeftBorderSum +
       maxRightBorderSum)

# Driver for divide-and-conquer maximum contiguous subsequence sum
   algorithm.
MaxSubsequenceSum( A ):
    return MaxSumRec(A, 0, A.size - 1)
```

1.9.7 Algorithm 4

`implementations/max-subsequence-sum`:

```
Input: array A of integers.
Output: the maximum positive subsequence sum.

MaxSubsequenceSum( A ):
    maxSum ← 0
    currentSum ← 0

    for [0, A.size] → j:
        currentSum += A[j]

        if currentSum > maxSum:
            maxSum ← currentSum
        else if currentSum < 0:
            currentSum = 0

    return maxSum
```