Notes for the course Algorithms and Data Structures CSE1305

(G. Bertalan)

Week 2.1

- Introduction to Algorithms and Data Structures.

- Complexity analysis and big-Oh notation [Sections 4.1, 4.2, 4.3].

- Recursion [Sections 5.1, 5.3, 5.4].

- Complexity analysis: proof methods [Section 4.4].

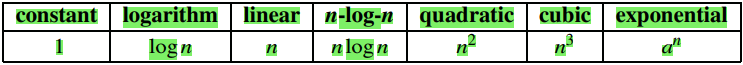
- Recursive analysis [Section 5.2].

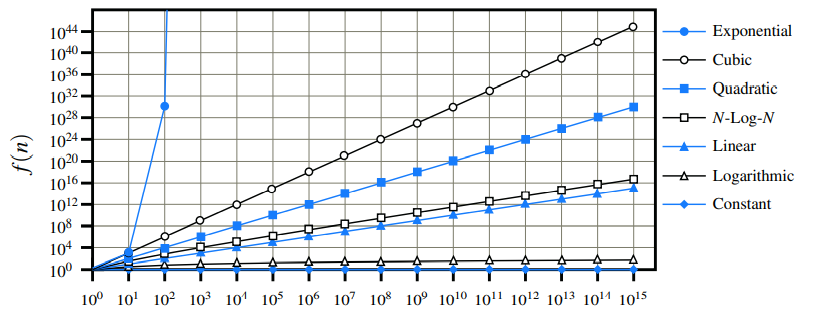
- Space complexity.

- Tail recursion and pitfalls of recursion [Section 5.5].

**Complexity analysis and big-Oh notation**

**Seven functions:**



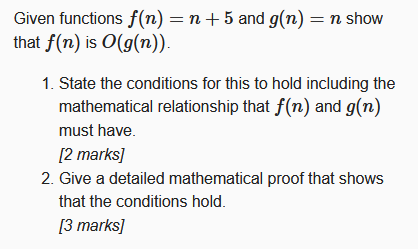


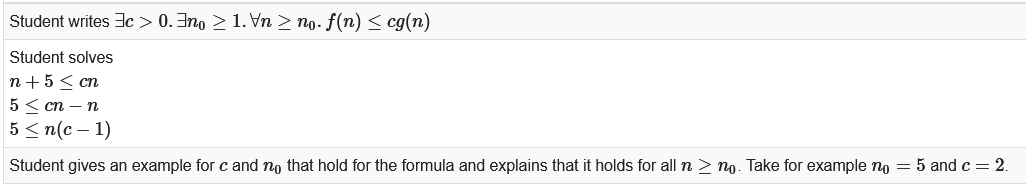
**The big-O notation:**

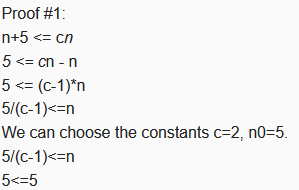
Let f(n) and g(n) be functions mapping positive integers to positive real numbers. We say that f(n) is O(g(n)) if there is a real constant c > 0 and an integer constant n0 ≥ 1 such that f(n) ≤ c · g(n), for n ≥ n0.

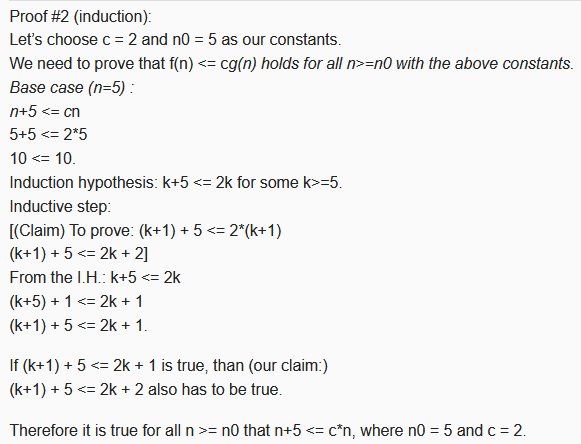
Short:  
∃c>0.∃n0≥1.∀n≥n0.f(n)≤cg(n)

Exercise:









Exercise:Prove that 5n^4 +3n^3 +2n^2 +4n+1 is O(n4).

Proof: Note that 5n^4 +3n^3 +2n^2 +4n+1 ≤ (5+3+2+4+1)n^4 = cn^4, for c = 15, when n ≥ n0 = 1.

Exercise(We rely on the mathematical fact that logn ≤ n for n ≥ 1):

Prove that 5n2 +3nlog n+2n+5 is O(n2).

Proof: 5n2 +3nlog n+2n+5 ≤ (5+3+2+5)n2 = cn2, for c = 15, when n ≥ n0 = 1.

Exercise:

Prove that 20n3 +10nlog n+5 is O(n3).

Proof: 20n3 +10nlog n+5 ≤ 35n3, for n ≥ 1.

Exercise:

Prove that 3log n+2 is O(logn).

Proof: 3logn+ 2 ≤ 5log n, for n ≥ 2. Note that logn is zero for n = 1. That is why we use n ≥ n0 = 2 in this case.

Exercise:

Prove that 2^(n+2) is O(2n).

Proof: 2n+2 = 2n ·22 = 4·2n; hence, we can take c = 4 and n0 = 1 in this case.

Exercise:

Prove that 2n+100log n is O(n).

Proof: 2n+100log n ≤ 102n, for n ≥ n0 = 1; hence, we can take c = 102 in this case.

**Big-Omega:**

We say that f(n) is Ω(g(n)), pronounced “ f(n) is big-Omega of g(n),” if g(n) is O(f(n)), that is, there is a real constant c > 0 and an integer constant n0 ≥ 1 such that f(n) ≥ cg(n), for n ≥ n0.

Short:  
∃c>0.∃n0≥1.∀n≥n0.f(n)>=cg(n)

Exercise:

Prove that 3n log n − 2n is Ω(nlog n).

Proof: 3nlog n− 2n = nlog n+ 2n(logn− 1) ≥ nlog n for n ≥ 2; hence, we can take c = 1 and n0 = 2 in this case.

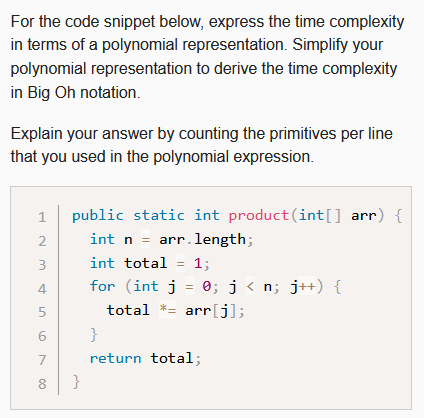
**Big-Theta:**

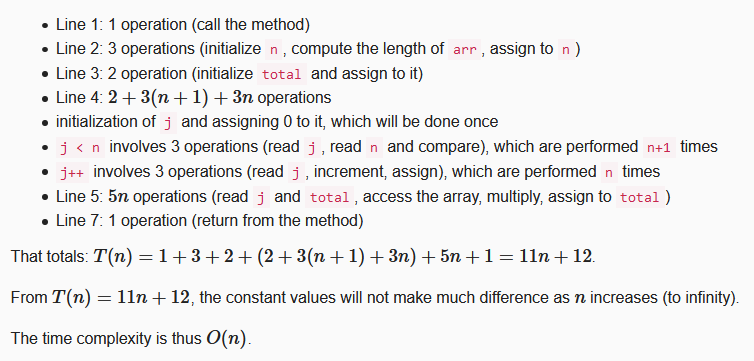
We say that f(n) is Θ(g(n)), pronounced “ f(n) is big-Theta of g(n),” if f(n) is O(g(n)) and f(n) is Ω(g(n)), that is, there are real constants c’ > 0 and c” > 0, and an integer constant n0 ≥ 1 such that c’g(n) ≤ f(n) ≤ c”g(n), for n ≥ n0.

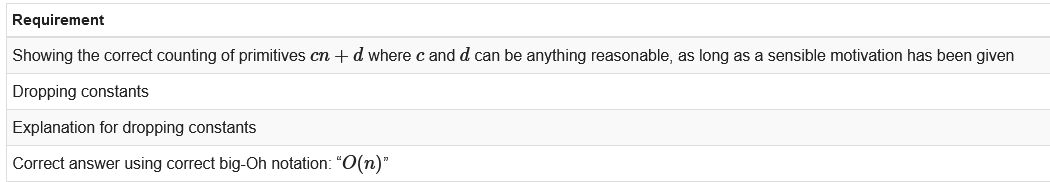
Exercise:

Prove that 3n log n + 4n + 5logn is Θ(nlog n).

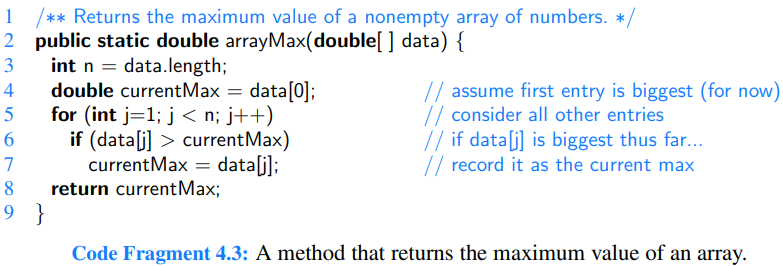
Proof: 3nlog n ≤ 3nlog n + 4n + 5logn ≤ (3+4+5)nlogn for n ≥ 2.







ArrayMax:



The *arrayMax* is O(n).

A more interesting question about arrayMax is how many times we might update the current “biggest” value. In the worst case, if the data is given to us in increasing order, the biggest value is reassigned n − 1 times. But what if the input is given to us in random order, with all orders equally likely; what would be the expected number of times we update the biggest value in this case? To answer this question, note that we update the current biggest in an iteration of the loop only if the current element is bigger than all the elements that precede it. If the sequence is given to us in random order, the probability that the jth element is the largest of the first j elements is 1/ j (assuming uniqueness). Hence, the expected number of times we update the biggest (including initialization) is Hn = ∑ 1/ j (j goes from 1 to n), which is known as the nth **Harmonic number.** It can be shown that Hn is **O(logn)**. Therefore, the expected number of times the biggest value is updated by arrayMax on a randomly ordered sequence is O(logn).

Composing long Strings