CSCE 222

Discrete Structures for Computing

Complexity of Algorithms

Dr. Hyunyoung Lee

Based on slides by Andreas Klappenecker

Overview

- Example - Fibonacci
- Motivating Asymptotic Run Time Analysis
- Asymptotic Notations
- Convenience of the Asymptotic Notations
- Complexity

Example

Fibonacci

The sequence of Fibonacci numbers is defined as

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

Fibonacci Implementation

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

```
def fib(n)
  return 0 if n == 0
  return 1 if n == 1
  return fib(n-1) + fib(n-2)
end
```

For any algorithm, we always ask the following questions:
1) Is it correct?
2) How much time does it take as a function of its input?
3) Can we do better?

Correctness

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

```
def fib(n)
  return 0 if n == 0
  return 1 if n == 1
  return fib(n-1) + fib(n-2)
end
```

Our implementation is obviously correct, since it is nothing but the definition of Fibonacci numbers.

Running Time

```
def fib(n)
  return 0 if n == 0
  return 1 if n == 1
  return fib(n-1) + fib(n-2)
end
```

The running time is a bit more difficult to analyze. Let's try it out in ruby.
Start the interactive ruby interpreter irb, type in the program,
run fib(0), fib(1), ..., fib(10), fib(100)

Running Time

```
def fib(n)
  return 0 if n == 0
  return 1 if n == 1
  return fib(n-1) + fib(n-2)
end
```

Slowpoke alert!

The running time T(n) of this program exceeds the nth Fibonacci number!

Impractical, except for the smallest inputs of n.

Let T(n) denote the number of steps needed to compute fib(n). Then:

T(n) <= 3 if n <=1

T(n) = T(n-1)+T(n-2) + 3 for n>1

that is, two recursive invocations of fib, two checks of values of n, plus one addition.

Motivating
Asymptotic Run Time Analysis

Running Time

The running time of a program depends on

- the input

- how the compiler translates the program

- the hardware

This can be quite complex, especially with modern architectures that might have long pipelines, speculative execution, and so on.

Time Estimates

The execution of a high level instruction might take a short or a long time, depending on the state of the pipeline, success of speculative execution, and many other factors.

We will have to estimate the time, no matter what kind of approach we will take, otherwise the analysis will become too complex.

Input

In general, the size of the input has a big impact on the running time, as we have seen in the Fibonacci example.

In general, we might have many inputs of the same size (e.g. data to be sorted). To simplify matters, we would like to study the running time as a function of the input size.

Problem: Different inputs of the same size can lead to a different running time.

Machine Independence

We would like to make the running time analysis independent of a particular choice of processor or compiler.

We do not want to redo an analysis if we replace a processor by a new processor that is twice faster and has a slightly different architecture.

==> Analyze running time up to a multiplicative constant.

Asymptotics

At the beginning of an execution, the behavior is quite different, since the pipeline might not be completely filled, the number of cache faults is large, and so on.

Therefore, it makes sense to consider the running time just for large input sizes.

As an added benefit, it will allow us to simplify our estimates of the best case and worst case running time even further.

Asymptotic Notations

Big Oh Notation

Let f, g: N -> R be functions from the natural numbers to the set of real numbers.

We write $f \in O(g)$ if and only if there exist some natural number n_0 and a positive real constant U such that

$$|f(n)| <= U|g(n)|$$

for all n satisfying $n >= n_0$

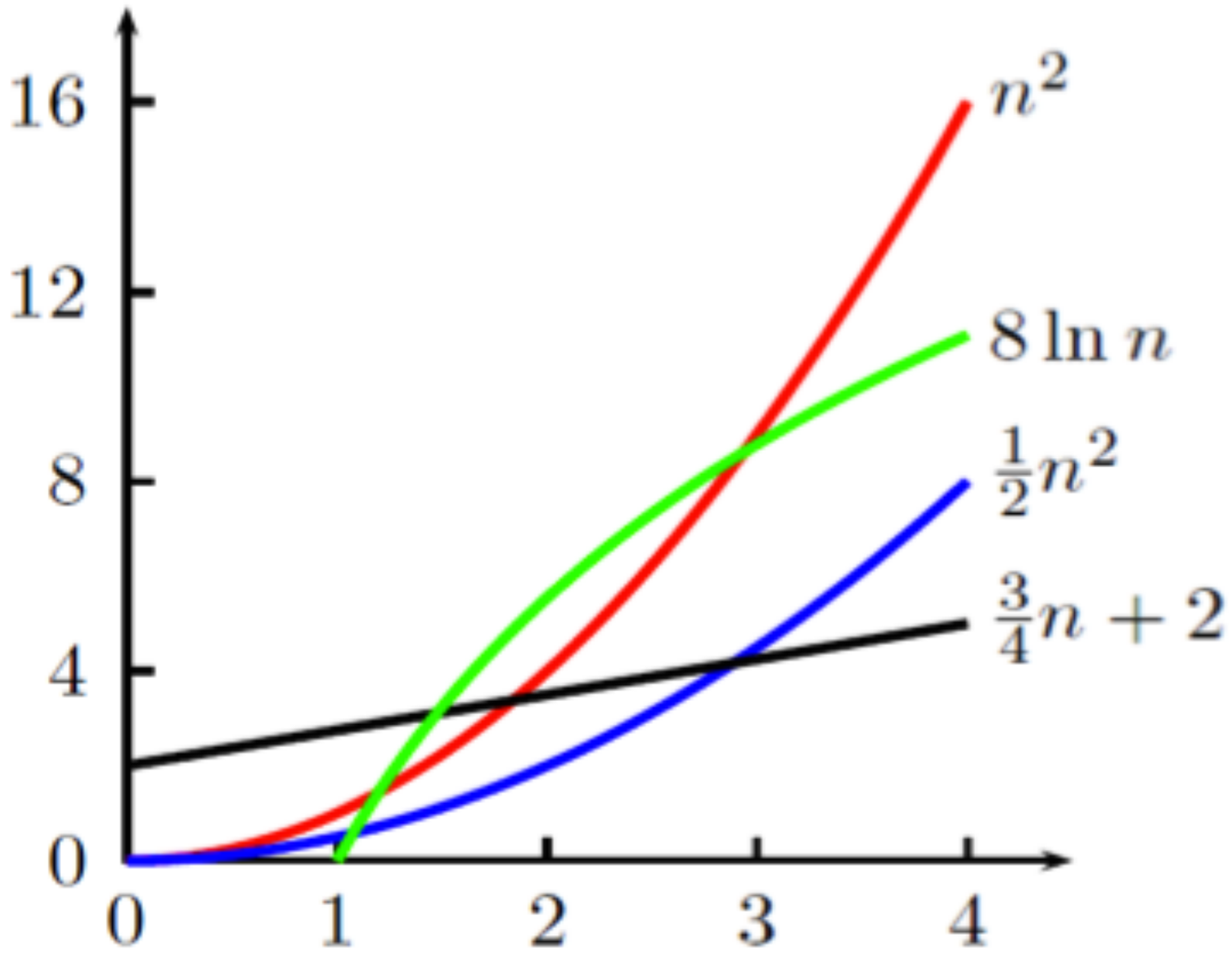
Set Interpretation

We have interpreted f(n)=O(g(n)) as a relation between the two functions f(n) and g(n).

We can give O(g(n)) a meaning by interpreting it as the set of all functions f(n) such that f(n)=O(g(n)), that is,

O(g) = { f: N -> R I there exists a real constant U and a natural number n_0 such that If(n)I <= UIg(n)I for all n >= n_0 }

Example O(n²)



Upper Bound

The big Oh notation provides an upper bound on a function f.

f(n) = O(n) means that f(n) <= Un

for some constant U and for all n >= n_0

The notation does not imply that f has to grow that fast.

E.g. f(n) = 1 satisfies f(n)=O(n).

so f(n) = O(1) would have been a better bound.

==> We need lower bounds as well ==> Big Ω

Big Ω

We define f(n) ∈ Ω(g(n)) if and only if there exist a positive real constant L and a natural number n_0 such that

L lg(n)l <= lf(n)l

holds for all n >= n_0.

In other words, f(n) = Ω(g(n)) if and only if g(n) = O(f(n)).

Example

In Analysis of Algorithms, you will learn that any comparison based sorting algorithm needs at least Ω(n log n) comparisons.

Mergesort needs O(n log n) comparisons, so this is essentially an optimal sorting algorithm.

Big Θ

We define f(n) ∈ Θ(g(n)) if and only if there exist positive real constants L and U and a natural number n_0 such that

Llg(n)l <= lf(n)l <= Ulg(n)l

holds for all n >= n_0.

In other words, f(n) = Θ(g(n)) if and only if

f(n) = Ω(g(n)) and f(n) = O(g(n)).

Informal Summary

f(n) = O(g(n)) means that |f(n)| is upper bounded by a constant multiple of |g(n)| for all large n.

f(n) =Ω(g(n)) means that |f(n)| is lower bounded by a constant multiple of |g(n)| for all large n.

f(n) = Θ(g(n)) means that |f(n)| has the same asymptotic growth as |g(n)| up to multiplication by constants.

Convenience of the Asymptotic Notations

Rule 1

Suppose that f(n) and g(n) are functions such that |f(n)| <= |g(n)| holds for all n >= n_0. Then f(n) + g(n) = O(g(n))

Indeed, |f(n)+g(n)| <= |f(n)|+|g(n)| <= 2|g(n)| holds for all n >= n_0. Therefore, by definition, f(n)+g(n) = O(g(n))

Examples:

$n^2+2n = O(n^2)$

$n^2+n \log n + 2n + 1723 = O(n^2)$

Rule 2

Suppose that c is a nonzero real number. Then

c f(n) = O(f(n))

Indeed, there exists a constant U=|c| such that

|cf(n)| <= |c||f(n)| = U|f(n)| holds for all n.

Therefore, cf(n) = O(f(n)).

Limits

The limit

$$\lim_{n \to \infty} f(n)$$

exists and is equal to L if and only if for all $\varepsilon > 0$ there exists a natural number $n_0 = n_0(\varepsilon)$ such that

$$|f(n) - L| < \epsilon$$

holds for all $n \geq n_0$.

Roughly, the sequence f(n) has a limit iff the values of f(n) approach L for large n.

Rule 3

Let f and g be two functions such that

$$\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|}$$

exists and is equal to L. Then $f(n) = O(g(n))$.

Indeed, since $\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = L$, it follows that for all $\varepsilon > 0$ there exists a natural number n_0 such that

$$L - \varepsilon \leq \frac{|f(n)|}{|g(n)|} \leq L + \varepsilon$$

holds for all $n \geq n_0$.

Hence, $|f(n)| \leq U|g(n)|$ holds for all $n \geq n_0$ with $U = L + \varepsilon$.

Rule 4

If f, g, h are functions such that

f(n) = O(g(n)) and g(n) = O(h(n))

then f(n) = O(h(n))

The proof is left as an exercise!

Lower bounds!!!

Big Omega

(1) Suppose that f(n) and g(n) are functions such that
|f(n)| <= |g(n)| holds for all n >= n_0.
Then f(n)+g(n) = Ω(f(n))

(2) c f(n) = Ω(f(n))

(3) If the limit $\lim_{n\to\infty} |g(n)|/|f(n)|$ exists,
then f(n) = Ω(g(n))

(4) If f(n) = Ω(g(n)) and g(n) = Ω(h(n)),
then f(n) = Ω(h(n))

Big Theta

Let f and g be two functions such that

$$\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|}$$

exists and is equal to $L > 0$. Then $f(n) = \Theta(g(n))$.

Indeed, since $\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = L > 0$, for all ε in the range $0 < \varepsilon < L$ there exists a natural number n_0 such that

$$L - \epsilon \leq \frac{|f(n)|}{|g(n)|} \leq L + \varepsilon$$

holds for all $n \geq n_0$. Hence $(L - \varepsilon)|g(n)| \leq |f(n)| \leq (L + \varepsilon)|g(n)|$ holds for all $n \geq n_0$. Thus, $f(n) = \Theta(g(n))$.

Big Theta Example

Let $f(n) = (2 + (-1)^n)n^2$ and $g(n) = n^2$. Then the limit

$$\lim_{n \to \infty} \frac{|f(n)|}{|g(n)|}$$

does not exist, as the quotient fluctuates between 3 and 1, but

$$|g(n)| \leq |f(n)| \leq 3|g(n)|$$

holds for all $n \geq 1$; hence, $f \in \Theta(g)$.

Exercise

Does $\Theta(n^3 + 2n + 1) = \Theta(n^3)$ hold?

<u>Step 1.</u> Prove that $n^3 + 2n + 1 \in \Theta(n^3)$ and $n^3 \in \Theta(n^3 + 2n + 1)$.

<u>Step 2.</u> Prove that $\Theta(n^3 + 2n + 1) \subseteq \Theta(n^3)$ and $\Theta(n^3) \subseteq \Theta(n^3 + 2n + 1)$ by using the results of Step 1 together with the "transitivity" of big-O and big-Ω.

Complexity

Elementary Operations

Elementary operations such as

- assignments a := rhs
- arithmetic with machine sized words (x+y, x-y, x*y, x/y)
- boolean operations (a && b, a ll b, a & b, a l b, ...)
- comparisons (a<b, a<=b, a==b, a>b, a>=b,...)
- array access arr[i]

have constant running time, that is, Θ(1) time.

Compound Statements

Suppose that we are given a sequence of statements:

Block1 ; Block 2; ... ; Block n

If Block k takes T_k time, then a total time of

$$T_1 + T_2 + ... + T_n$$

is required to execute the sequence of the n blocks.

Control Structures

if BoolExpr then

Block 1

else

Block 2

end

If evaluation of BoolExpr takes time T_B, and execution of the Block k takes T_k time, then the above if .. then .. else statement takes $O(T_B) + O(\max(T_1, T_2))$ time.

For Loop

for k in (a..b) do

  Block 1

end

If $T_1(v)$ is the time required for the execution of Block 1 when the loop variable k == v, then the for loop takes

$O(T_1(a)) + O(T_1(a+1)) + ... + O(T_1(b))$

time. In particular, if the running time T_1 of the block is independent from the loop variable, then $(b-a+1)\ O(T_1)$ time.

Function Calls

```
def f( params )
    Block 1
end
```

If T_p is the time to assign the parameters and T_1(params) is the time to execute Block 1 given the parameters params, then the running time of f(params) is given by $O(T_p + T_1)$.

Example

```
def bubble_sort(list)
   list = list.dup   # make copy of input, so we do not modify original list
   for i in 0..(list.length-2) do
     for j in 0..(list.length - i - 2) do
       list[j], list[j + 1] = list[j + 1], list[j]  if list[j + 1] < list[j]
     end
   end
   return list
end
```

Example

```
def bubble_sort(list)
   list = list.dup   # make copy of input, so we do not modify original list
   for i in 0..(list.length-2) do
     for j in 0..(list.length - i - 2) do
       list[j], list[j + 1] = list[j + 1], list[j]  if list[j + 1] < list[j]
     end
   end
   return list
end
```

```
>> bubble_sort([6,5,4,3,2,1])
[5, 4, 3, 2, 1, 6]
[4, 3, 2, 1, 5, 6]
[3, 2, 1, 4, 5, 6]
[2, 1, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

Loop Body

list[j], list[j + 1] = list[j + 1], list[j] if list[j + 1] < list[j]

The evaluation of the boolean expression takes O(1) time, and the parallel assignment realizing the swap operation takes O(1) as well, since it corresponds to 3 assignments.

Therefore, the statement takes O(1) time.

Inner For Loop

```
for j in 0..(list.length - i - 2) do
    list[j], list[j + 1] = list[j + 1], list[j]  if list[j + 1] < list[j]
end
```

Let n = list.length

Then the loop takes (n - i - 1) O(1) time.

Nested For Loops

```
for i in 0..(list.length-2) do # number of iterations: n-1
    for j in 0..(list.length - i - 2) do # number of itns: n-1, n-2, …, 1
      list[j], list[j + 1] = list[j + 1], list[j]  if list[j + 1] < list[j]
    end
  end
```

Let n = list.length. Then the two nested loops take

(n-1 + n-2 + ... + 1) O(1) = (n(n-1)/2) O(1) = $O(n^2)$ time.

Total Time Complexity

```
def bubble_sort(list) # O(1) for parameter assignment
  list = list.dup   # O(n)
  for i in 0..(list.length-2) do # O(n²)
    for j in 0..(list.length - i - 2) do
      list[j], list[j + 1] = list[j + 1], list[j]  if list[j + 1] < list[j]
    end
  end
  return list # O(1)
end # O(1) + O(n) + O(n²) + O(1) = O(n²)
```

Example: While Loop

```
i := 1
t := 0
while i <= n
    t := t + i
    i := 2i
```

First, count the number of addition or multiplication operations, and then, find the representative term for a big-O estimate.