January 5, 2022

Analysis of Algorithms

UNIVERSITY OF CALIFORNIA UC RIVERSIDE

Analysis of Algorithms: Issues

- Correctness
- Running time ("*time complexity*")
- Memory requirements ("*space complexity*")
- Power
- I/O utilization
- Ease of implementation
- …

Correctness

An algorithm is **correct** if, for every input size,

it halts

with the correct output.

Analysis of Algorithms

- **Primitive Operations**: Low-level computations independent from the programming language can be identified in pseudo-code
- Examples:
  - calling a method and returning from a method
  - arithmetic operations (e.g. addition)
  - comparing two numbers, etc.
- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm

Input size and basic operation examples

Problem	***Input size measure***	***Basic operation***
Searching for key in a list of n items	Number of items in the list, i.e., n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	size of n = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or #edges	Visiting a vertex or traversing an edge

Why Running Time ?

Definition of the Fibonacci function

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

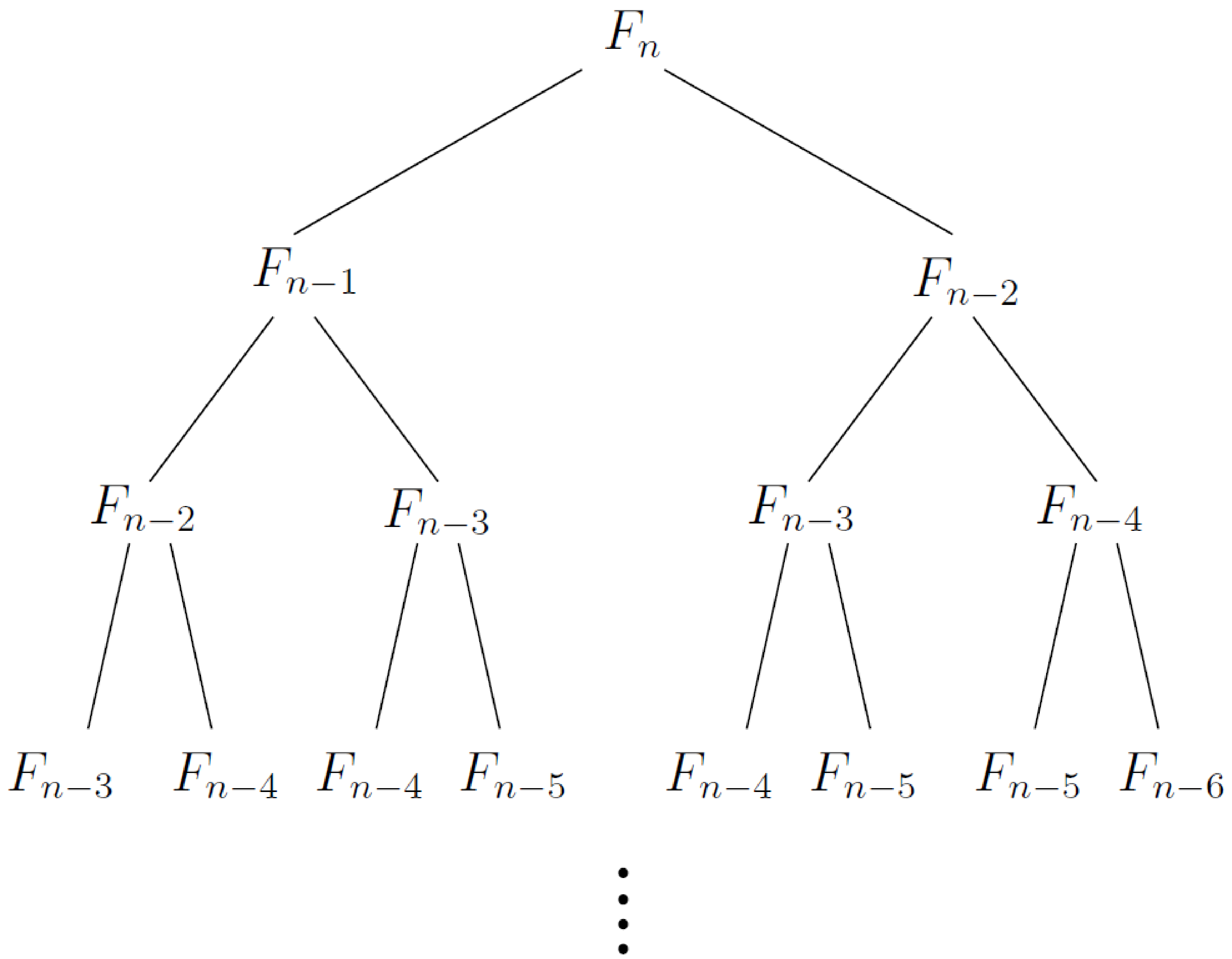
Recursive implementation

```
function fib1(n)
if n = 0:   return 0
if n = 1:   return 1
return fib1(n − 1) + fib1(n − 2)
```

Time complexity?

$$T(n) = \begin{cases} & n \leq \\ & n > \end{cases}$$

The proliferation of recursive calls in ***fib1***



```
function fib1(n)
if n = 0:   return 0
if n = 1:   return 1
return fib1(n − 1) + fib1(n − 2)
```

$$T(n) = \begin{cases} 2 & \text{for } n \le 1 \\ T(n-1) + T(n-2) + O(1) & \text{for } n > 1 \end{cases}$$

$$T(n) \ge F_n \approx 2^{0.694n}$$

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i-1] + f[i-2]
return f[n]
```

$T(n)$ is linear in n !!!

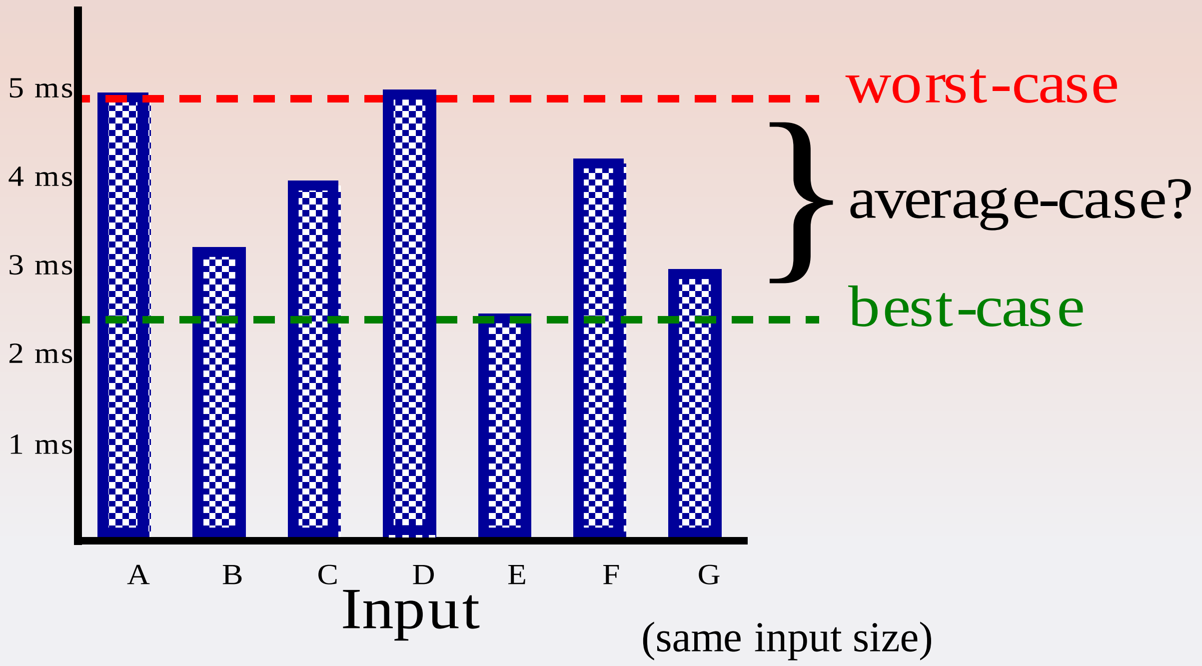
Average Case vs. Worst Case

- An algorithm may run faster on certain data sets than on others (e.g., for the sorting problem, the input is partially sorted)

- Finding the average case can be very difficult, so typically algorithms are measured by the worst case time complexity

Average Case vs. Worst Case

- In time-critical application domains (e.g., air traffic control, surgery, IP lookup, …) knowing the worst case time complexity is crucial



Worst Case Time-Complexity

- Definition: The worst case time-complexity of an algorithm *A* is the *asymptotic* running time of *A* as a *function of the size of the input*, when the input is the one that makes the algorithm *slower* in the limit
- How do we measure the running time of an algorithm?

Example

```
def iMax(A):
    currentMax = A[0]
    for i in range(len(A)):
        if currentMax < A[i]:
            currentMax = A[i]
    return currentMax
```

Max iterative

Max recursive

```
def rMax(A, n):
    if n == 1:
        return A[0]
    return max(rMax(A(1:n-1),A[n])
```

Time-complexity is $O(n)$

Asymptotic notation

Section 0.3 of the textbook

The "Big-Oh" Notation

- Definition: Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$

  if and only if

  there are positive constants c and n_0 such that $f(n) \leq c\, g(n)$ for $n \geq n_0$

The "Big-Oh" Notation

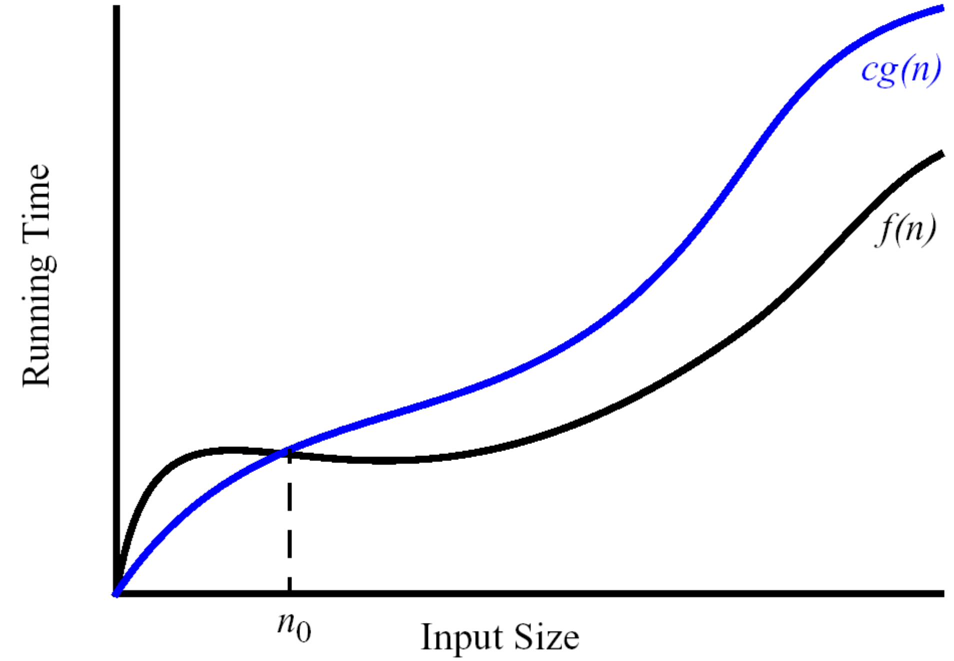


Figure 1.3: Illustrating the "big-Oh" notation. The function $f(n)$ is $O(g(n))$, for $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

Asymptotic Notation
Big - O

Theorem

Suppose that $f_1(x) = O(g_1(x))$ *and* $f_2(x) = O(g_2(x))$. *Then*

$$(a)\quad f_1(x) + f_2(x) = O(g_1(x) + g_2(x))$$

$$= O(\max(g_1(x), g_2(x)))$$

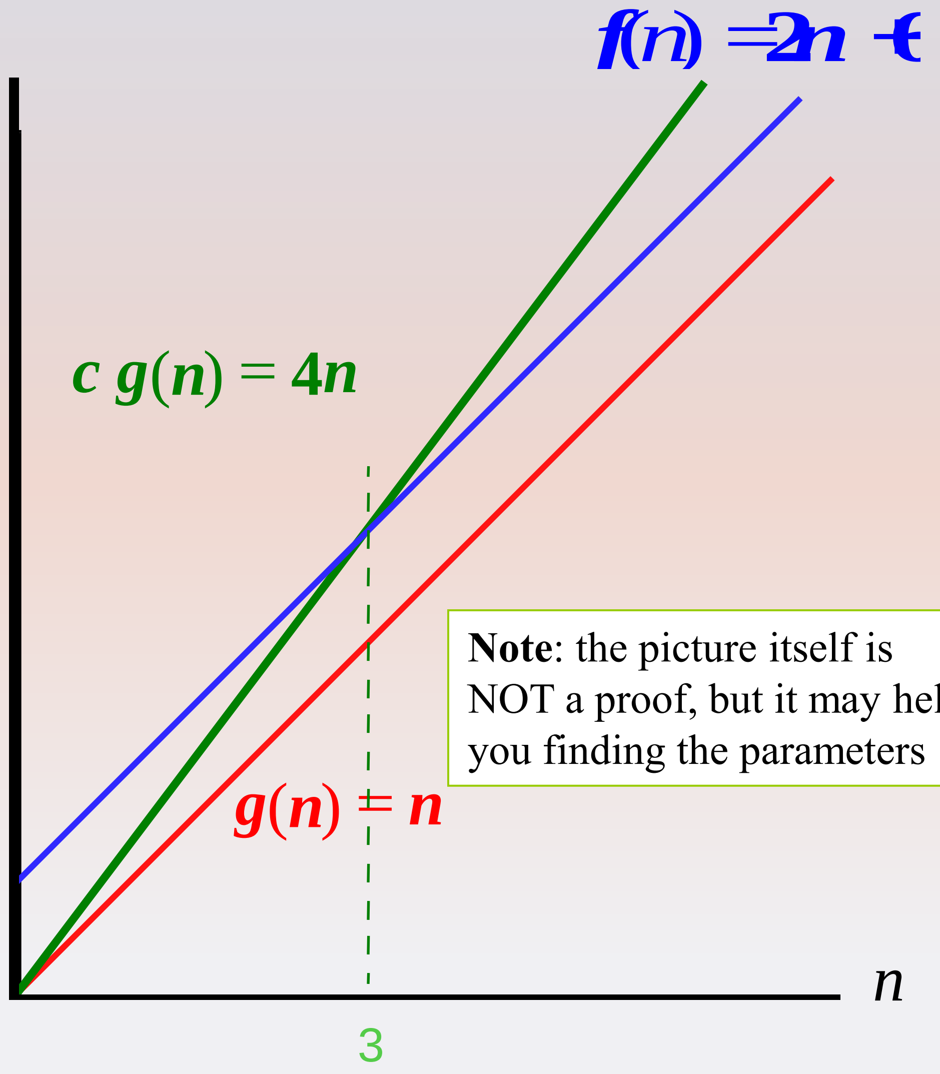
$$(b)\quad f_1(x)f_2(x) = O(g_1(x)g_2(x))$$

Example

$f(n) = 2n+6$

$g(n) = n$

For functions $f(n)$ and $g(n)$ (to the right) there are positive constants c and n_0 such that:
$f(n) \le c\, g(n)$ for $n \ge n_0$



Note: the picture itself is NOT a proof, but it may help you finding the parameters

Proof

- $f(n)=2n+6$
- $g(n)=n$
- $2n+6 \leq 4n$???
- $2n+6 \leq 4n$ when $n \geq 3$
- So, if we choose $c=4$, then $n_0=3$ satisfies $f(n) \leq c\ g(n)$ for $n \geq n_0$
- <u>Conclusion</u>: $2n+6$ is $O(n)$

Asymptotic Notation
Big - O

Theorem

Let $f(x) = \sum_{i=0}^{k} a_i x^i$. Then $f(x) = O(x^k)$.

Proof: Let $A = \max |a_i|$, be the maximum absolute value of the coefficient in $f(x)$. We can estimate $f(x)$ as follows. For $x \geq 1$ we have

$$\begin{aligned} f(x) &= a_k x^k + a_{k-1} x^{k-1} + \ldots + a_1 x + a_0 \\ &\leq A(x^k + x^{k-1} + \ldots + x + 1) \\ &\leq A(k+1)x^k. \end{aligned}$$

Thus $f(x) \leq cx^k$ for $c = A(k+1)$ and $x \geq 1$. The theorem follows. $\square$

Asymptotic Notation

- Note: Even though it is correct to say "$7n - 3$ is $O(n^3)$", a more precise statement is "$7n - 3$ is $O(n)$", that is, one should make the approximation as tight as possible
- Simple Rule: Drop lower order terms and constant factors

  $7n-3$ **is** $O(n)$

  $8n^2 \log n + 5n^2 + n$ **is** $O(n^2 \log n)$

Asymptotic Notation

Big - O

Theorem

Let a > 0, b > 0, c > 1. Then

(a) $1 = O(\log^a n)$. (b) $\log^a n = O(n^b)$. (c) $n^b = O(c^n)$

Proof: (c) Let $d = c^{1/b}$, then $d > 1$, and

$$\begin{aligned} n &\leq 1 + d + d^2 + \ldots + d^{n-1} && \text{, since } d > 1 \\ &= \frac{d^n - 1}{d - 1} && \text{- summation of the geom. sequences} \\ &\leq Ad^n, && \text{, where } A = 1/(d-1) \\ &= Ac^{(1/b)n} \end{aligned}$$

$$n^b \leq Bc^n \quad \text{, where } B = A^b$$

$$n^b = O(c^n)$$

Asymptotic Notation

- Special classes of algorithms
  - constant: $O(1)$
  - logarithmic: $O(\log n)$
  - linear: $O(n)$
  - quadratic: $O(n^2)$
  - cubic: $O(n^3)$
  - polynomial: $O(n^k), k \geq 1$
  - exponential: $O(a^n), n > 1$

Asymptotic Notation

- "Relatives" of the Big-Oh
  - $\Omega(f(n))$: Big Omega
    - asymptotic *lower* bound
  - $\Theta(f(n))$: Big Theta
    - asymptotic *tight* bound

Big Omega

- Definition: Given two functions ***f(n)*** and ***g(n)***, we say that ***f(n)*** is Ω(***g(n)***)
  if and only if
  there are positive constants $\mathbf{c}$ and n_0 such that $f(n) \geq c\, g(n)$
  for $n \geq n_0$

- Property: ***f(n)*** is Ω(***g(n)***) *iff* ***g(n)*** is *O*(***f(n)***)

Big Theta

- <u>Definition</u>: Given two functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Theta(g(n))$ *if and only if* there are positive constants c_1, c_2 and n_0 such that $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for $n \geq n_0$
- <u>Property</u>: $f(n)$ is $\Theta(g(n))$ if and only if "$f(n)$ is $O(g(n))$ AND $f(n)$ is $\Omega(g(n))$"

Summary

- $A \in O(\mathbf{f(n)})$ *means* "the algorithm *A* won't take longer than *f(n)*, give or take a constant multiplier and lower order terms" (upper bound)
- $A \in \Theta(\mathbf{f(n)})$ *means* "the algorithm *A* will take as long as *f(n)*, give or take a constant multiplier and lower order terms" (tight bound)
- $A \in \Omega(\mathbf{f(n)})$ *means* "the algorithm *A* will take longer than *f(n)*, give or take a constant multiplier and lower order terms" (lower bound)

Establishing order of growth using limits

$$\lim_{n\to\infty} f(n)/g(n) = \begin{cases} \mathbf{0} & \text{order of growth of } \boldsymbol{f(n)} < \text{ order of growth of } \boldsymbol{g(n)} \\ \boldsymbol{c} > \mathbf{0} & \text{order of growth of } \boldsymbol{f(n)} = \text{ order of growth of } \boldsymbol{g(n)} \\ \infty & \text{order of growth of } \boldsymbol{f(n)} > \text{ order of growth of } \boldsymbol{g(n)} \end{cases}$$

Examples:

- **$10n$ vs. n^2**
- **$n(n+1)/2$ vs. n^2**

Orders of growth: some important functions

- All logarithmic functions $\log_a n$ belong to the same class $\Theta(\log n)$ no matter what the logarithm's base $a > 1$ is
- All polynomials of the same degree k belong to the same class: $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$ in $\Theta(n^k)$
- Exponential functions a^n have different orders of growth for different a's
- order $\log n$ < order n < order $n \log n$ < order n^k ($k \geq 2$ constant) < order a^n < order $n!$ < order n^n
- Caution: Be aware of very large constant factors

Suppose each operation takes 1 nanoseconds (10^{-9} seconds)

n	lg n	n	n lg n	n^2	2^n	n!
10	0.003μs	0.01μs	0.033μs	0.1μs	1μs	3.63ms
20	0.004μs	0.02μs	0.086μs	0.4μs	1ms	77.1years
30	0.005μs	0.02μs	0.147μs	0.9μs	1sec	>10^{15}years
100	0.007μs	0.1μs	0.644μs	10μs	>10^{13}years	
10,000	0.013μs	10μs	130μs	100ms		
1,000,000	0.020μs	1ms	19.92μs	16.7min		

- For $n < 10$, the difference is insignificant.
- Θ (n!) algorithms are useless well before $n = 20$.
- Θ (2^n) algorithms are practical for $n < 40$.
- Θ (n^2) and Θ (n lg n) are both useful, but Θ (n lg n) is significantly faster.

Time analysis for iterative algorithms

Steps

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst* case(s) for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules

Example

Give the number f(n) of letters "Z" printed by Algorithm PrintZs below:

(first using a summation notation, and then - a closed-form formula for f(n))

Analyze the worst-case time complexity of the following algorithm, and give a tight bound using the big-theta notation

Algorithm PRINTZS (n : **integer**)
  for $i \leftarrow 1$ **to** $3n+1$ **do**
    for $j \leftarrow 1$ **to** i^2+2 **do** print("Z")

Example

Give the number f(n) of letters “Z” printed by Algorithm PrintZs below:

(first using a summation notation, and then - a closed-form formula for f(n))

Analyze the worst-case time complexity of the following algorithm, and give a tight bound using the big-theta notation

Algorithm PRINTZS (n : **integer**)
    for $i \leftarrow 1$ **to** $3n+1$ **do**
        for $j \leftarrow 1$ **to** i^2+2 **do** print(“Z”)

$$\sum_{i=1}^{3n+1}(i^2+2) = 9n^3 + \frac{27}{2}n^2 + \frac{25}{2}n + 3.$$

These slides were shared with me
by Dr. Stefano Lonardi
and modified with his permission.