Curtin
University of Technology

Design and Analysis of Algorithms

Lecture 1 & Lecture 2

Big O Notation

Recurrence Relations

Curtin University of Technology

Pre-requisite

Data Structures and Algorithms

- We will use advanced data structures to make algorithms run faster

Textbooks

Required

- T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms* (3rd edition).

Curtin University of Technology

Algorithm

- Al Khwarizmi (9^{th} century Persian mathematician, Bagdad) wrote a textbook (in Arabic) about basic methods for adding, multiplying, and dividing numbers, extracting square roots, and calculating digits of π.
  - Al Khwarizmi → when written in Latin, the name became Algorismus / Algoritmi

- An algorithm is any well-defined computational procedure that
  - takes some value as input
  - produces some value as output
  - solves a specified computational problem

- An algorithm
  - must be correct (*i.e.*, always gives the right result)
  - should be tractable & terminate (*i.e.*, gives a result in reasonable time)
  - can be specified in English, as computer program, or as hardware design

Curtin University of Technology

Problem Example-1

Sorting problem: A problem you have learnt before …
Input: a sequence of n numbers $(a_1, a_2, \dots, a_n)$
Output: A reordering $(b_1, b_2, \dots, b_n)$ of input sequence such that $b_1 \le b_2 \le \dots \le b_n$.

Example:
Input: (30, 20, 41, 51, 3, 20)
Output: (3, 20, 20, 30, 41, 51)

- What kind of problems are solved by algorithms?
  See Textbook pp. 6-8

Algorithm for Example-1

INSERTION-SORT (*A*)

```
for j = 2 to length(A) do
    key = A[j]
     // insert A[j] into the sorted sequence A[1 … j – 1]
     i = j – 1
     while i > 0 and A[i] > key do
         A[i+1] = A[i]
         i = i – 1
     A[i+1] = key
```

// The textbook and lecture slides assume **Array *A* starts from index 1**

// *length*(*A*) is the total number of elements in array *A*.

// Insertion sort (IS) is *in-place* sorting algorithm: the numbers are rearranged within the array *A*; it requires only a constant amount of memory for temporary variables, loop control variables, sentinel, etc

// IS is a *stable* sorting algorithm: it maintains the relative order of repeated elements

// IS is fast for small sized input and for input that is almost sorted

Curtin University of Technology

Insertion Sort Example

1	2	3	4	5	6
5	**2**	4	6	1	3

→

1	2	3	4	5	6
2	5	**4**	6	1	3

1	2	3	4	5	6
2	4	5	**6**	1	3

→

1	2	3	4	5	6
2	4	5	6	**1**	3

1	2	3	4	5	6
1	2	4	5	6	**3**

→

1	2	3	4	5	6
1	2	3	4	5	6

Curtin University of Technology

Problem Example-2

Greatest Common Divisor (GCD): learnt in high school …
// also known as Greatest Common Factor (GCF)
Input: Integers **X** and **Y**
Output: The largest integer **Z** that divides both X and Y, i.e., **Z =** GCD (X, Y)
Note: GCD (X, Y) = GCD (Y, X)

Example:
Input: X = 1035, Y = 759
Output: Z = GCD (1035, 759) = 69

Algorithm-1 for Example-2
Step 1: Find all **prime factors** of both X and Y
Step 2: Multiply all **common** prime factors to form Z

Example: Find the GCD of X = 1035 and Y = 759

Step-1: 1035 = 3^2 * 5 * 23 and 759 = 3 * 11 * 23

→ The common prime factors are: 3 and 23

Step-2: Z = 3 * 23 = 69

Curtin University of Technology

Problem Example-2 (cont.)

Algorithm-2 for Example-2: by Euclid (325 – 265 BC)

For two positive integers X and Y, where X ≥ Y, we have

$$\mathrm{GCD}(X, Y) = \mathrm{GCD}(X \textbf{ mod } Y, Y)$$

Note: You can use a **recursive** or **iterative** function to implement Algorithm-2

// a recursive version

GCD (x, y)

if $y = 0$ **then return** x

return GCD (y, x **mod** y)

Example: X = 1035, Y = 759

GCD (1035, 759) = GCD (1035 **mod** 759, 759) = GCD (276, 759)

GCD (759, 276) = GCD (759 **mod** 276, 276) = GCD (207, 276)

GCD (276, 207) = GCD (276 **mod** 207, 207) = GCD (69, 207)

GCD (207, 69) = GCD (207 **mod** 69, 69) = GCD (0, 69) = GCD (**69**, 0)

Problem Example-3

Least Common Multiple (LCM): also from high school …
// also known as Greatest Common Factor (GCF)
Input: Integers **X** and **Y**
Output: The smallest integer **Z** divisible by both **X** and **Y**, i.e., Z = **LCM** (X, Y)
Note: LCM (X, Y) = LCM (Y, X)

Example:
Input: X = 1035, Y = 759
Output: Z = LCM (1035, 759) = 11385

Algorithm-1 for Example-3
Step 1: Find all **prime factors** of both X and Y
Step 2: Multiply all prime factors to form Z
for each prime factor common to X and Y, use the largest power.
Example: Find the GCD of X = 1035 and Y = 759
Step-1: 1035 = $\mathbf{3^2}$ * 5 * **23** and 759 = **3** * 11 * **23**
→ The common prime factors are: 3 and 23
Step-2: Z = $\mathbf{3^2}$ * 5 * 11 * 23 = 11385

Problem Example-3 (cont.)

Algorithm-2 for Example-3:

We can use the solution of GCD (X, Y) to compute LCM (X, Y) as follows.

LCM (X, Y) = (X * Y) / GCD (X, Y).

Example: X = 1035, Y = 759

LCM (1035, 759) = (1035 * 759) / GCD (1035, 759)

From the previous example, we have GCD (1035, 759) = 69

= (1035 * 759) / 69 = 11385

Problem Example-4

Integer multiplication: **learnt in primary school …**

Input: Integers **X** and **Y**

Output: Z = **X** * **Y**

Example:

Input: X = 12, Y = 34

Output: Z = 12 * 34 = 408

Algorithm-1 for Example-4: We all know this algorithm!

```
   1 2
   3 4   x
   -----
   4 8
 3 6     +
 -------
 4 0 8
```

Curtin University of Technology

Problem Example-4 (cont.)

Algorithm-2 for Example-4: Al Khwarizmi's Algorithm

1. Divide the first number by 2 (in Col 1)
2. Double the second number (in Col 2)
3. Repeat until the first number becomes 1
4. Add all rows in Col 2 that has odd number in Col 1

Example:

12 * 34 = 408

Col 1	Col 2
12	34
6	68
3	**136**
1	**272**

Result = **408**

25 * 70 = 1750

Col 1	Col 2
25	70
12	140
6	280
3	**560**
1	**1120**

Result = **1750**

Problem Example-4 (cont.)

What if X and Y are *n* digit numbers?

Use Algorithm-1 for Example-4

n digits

A	B
C	D

×

$10^n AC + 10^{n/2} (AD + BC) + BD$

2 digits

1	2
3	4

×

$10^2\ 1*3 + 10^1 (1*4 + 2*3) + 2*4 = \mathbf{408}$

The algorithm recursively computes (A*C), (A*D), (B*C), and (B*D)

Problem Example-4 (cont.)

Algorithm-3 for Example-4: Karatsuba's algorithm

Notice that AD + BC = (A + B) (C + D) – AC – BD because

(A + B) (C + D) – AC – BD = AC + AD + BC + BD – AC – BD

= AD + BC

Thus, we need multiplications only for: AC, BD, and (A + B) (C + D) instead of AC, BD, AD and BC, meaning that Karatsuba's multiplication is faster!

Note: a **multiply** operation is more time consuming than an **add** operation

n digits

A	B
C	D

×

10^n AC + $10^{n/2}$ ((A + B) (C + D) – AC – BD) + BD

2 digits

1	2
3	4

×

10^2 3 + 10^1((1 + 2) (3 + 4) – 1*3 – 2*4) + 2*4 = **408**

Curtin University of Technology

Problem Example-5

Addition of *n* consecutive numbers 1, 2, 3, …, *n*
Input: Integer n
Output: $1 + 2 + 3 + \ldots + n - 1 + n$

Example:
Input: $n = 10$
Output: $1 + 2 + 3 + \ldots + 9 + 10 = 55$

Algorithm-1 for Example-5: Obvious – consecutively add those ten numbers!
What if *n* is a large number, i.e., 1M? 1B?

Algorithm-2 for Example-5: Some says by Carl Friedrich Gauss who knew the algorithm when he was 8 years old!

$1 + 2 + 3 + \ldots + n = n(n+1)/2;$

Thus, $1 + 2 + 3 + \ldots + 9 + 10 = 10\,(10 + 1)/2 = 55$

What other problems?

How many digits are there in Pi (π)? $\pi = 3.14159265$ ….

- In 2011, the record was more than 10 trillion digits!
- Note: Pi day is March 14
- **Algorithms to compute** π?

How many digits are there in Phi (Φ - pronounced fi)? Million digits!

- Φ is the golden ratio
  - also called Golden Number, Golden Proportion, Golden Mean, Golden Section.
- $\Phi = 1 + 1/\Phi \rightarrow \Phi^2 - \Phi - 1 = 0 \rightarrow \Phi = \frac{1+\sqrt{5}}{2} =$ 1.6180339887498948204586834...
- OR $\phi = 1/\Phi = 0.6180339887498948204586834$ …
  - Phi day is June 18
- The ratio of each successive pair of Fibonacci numbers approximates phi, e.g., 2584/1597 = 1.618033813

Textbook pp. 6-8 discusses other problems

Curtin University of Technology

Algorithm as Technology

- Algorithms devised for the same problem often differ dramatically in their efficiency
  - Which algorithm for each of the five examples is the most efficient?

- **Example for sorting problem:**
  - Insertion sort (**IS**) takes time $c_1 n^2$ to sort n items.
    - We will discuss how to analyse algorithm time complexity later.
  - Merge sort (**MS**) takes time $c_2 n \lg n$
  - Assume *constants* $c_1 = 2$, $c_2 = 50$
  - Assume you use the algorithms to sort $n = 10^6$ elements.

Curtin University of Technology

Algorithm as Technology (cont.)

- Assume CPU **A** runs **IS**, and CPU **B** runs **MS**
  - CPU **A** executes 10^9 instructions/sec while CPU **B** executes 10^7 instructions/sec
    - Thus, CPU B is 100 times slower as compared to CPU A.

- **A** takes: $\dfrac{2\cdot(10^6)^2 \, instructions}{10^9 \, instructions/\sec} = 2000$ seconds

- **B** takes: $\dfrac{50\cdot 10^6 \cdot \lg 10^6 \, instructions}{10^7 \, instructions/\sec} \approx 100$ seconds

- **MS** runs faster even using the slower CPU and assuming larger constant value!
  - This example shows the importance of designing more efficient algorithm for faster problem solutions!

How to design an algorithm?

- Reduce the problem to one with good known solution
- Write a brute force algorithm and use “tricks” to improve it
- This unit (DAA) focuses on time-efficient algorithm
- Another unit, Theoretical Foundations of Computer Science, covers problems
  - with no known efficient solution → NP-complete
  - with no solution → undecidable

Curtin University of Technology

How to design an algorithm? (cont.)

Trick 1

- *Use better data structures*
  - Often the implementation of an algorithm can be improved by choosing appropriate data structures
  - Lists vs. Arrays for split/join
  - Heaps for repeated min/max
  - Balanced trees or hashing for fast search
  - Graphs, etc.

Trick 2

- *Preprocessing the input* (*e.g., sorting*)
  - Allow binary search vs. linear search
  - Bring important items to the front of a list
  - Attempt to format the data so the algorithm performs at its best, rather than at its worst.

How to design an algorithm? (cont.)

Trick 3: *Use good algorithm design techniques*

- **Divide & Conquer technique**
  - Split a problem into sub parts
  - Solve each part
  - Re-join to get a solution
  - Will learn this trick in the following lecture

- **Greedy approach**
  - Define a "cost"
    - Cost can be $ value, time, etc.
  - Sort items by best impact on cost
  - Greedily choose the best until problem solved
  - Note that the solution/result may or may not be "optimal"
  - Will learn this trick in the following lecture

- **Dynamic programming**
  - Will learn this trick in the following lecture

Curtin
University of Technology

Algorithm Analysis

- Efficiency measure:
  - **Speed**: How long an algorithm takes to produce results
    - This is the usual measure.
  - **Space/memory**: How much memory is required to run the algorithm.
    - This measure is less commonly used.
- In general, the time taken by an algorithm grows with the size of input
  - Input size: depends of problems being studied (*e.g.*, #elements, #nodes, #links)
  - What is the input size of a sorting algorithm?
- Use the same **computation model** for the analyzed algorithms
  - Running time on a particular input: #of primitive operations or steps executed using the computation model/computer

Curtin
University of Technology

Computation Model

- The usual (often not stated) computation model is the Random Access Machine (RAM)
  - Sequential → RAM executes one instruction at a time
  - RAM contains instructions of real computers: arithmetic (*e.g.*, **add**, **multiply**), data movement (*e.g.*, load, store), control (*e.g.*, **if**, **subroutine call**).
  - Each instruction takes a constant amount of time/step
  - Running time: the total number of steps
- Data type: integer and floating point
- We need to limit the size of each word of data used

Example

$n = 1$	1 step	$n = 2$	1 step
$n = n + 1$	2 steps	print n	1 step
print n	1 step		
	4 steps		**2 steps**

Assignment operation is 1 step in RAM model

Addition operation is 1 step in RAM model

Let's say print operation is 1 step in RAM model

Curtin University of Technology

Analysis of Insertion Sort

INSERTION-SORT (*A*)	**cost**	**times**
for $j = 2$ to $length(A)$ **do**	c_1	n
$key = A[j]$	c_2	$n - 1$
// insert $A[j]$ into the sorted sequence $A[1 \dots j-1]$		
$i = j - 1$	c_4	$n - 1$
while $i > 0$ and $A[i] > key$ **do**	c_5	$\sum_{j=2}^{n}(t_j + 1)$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j)$
$i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j)$
$A[i+1] = key$	c_8	$n - 1$

c_i = the constant number of steps used to execute the operation in line i, for $i = 1, 2, 3, \dots, 8$

t_j = the number of elements the j^{th} key has to travel to get in its proper place, for $j = 2, 3, \dots, n$

Curtin University of Technology

Analysis of Insertion Sort

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} (t_j + 1)$$

$$+ c_6 \sum_{j=2}^{n} (t_j) + c_7 \sum_{j=2}^{n} (t_j) + c_8(n-1)$$

So, running time, $T(n)$, depends on input *size* $n = length(A)$

Curtin
University of Technology

Analysis of Insertion Sort

- **Best Case:** when the array is already sorted
  - Line 5: $A[i] \le key \rightarrow t_j = 0$, for $j = 2, 3, \ldots, n$

Thus,

$$c_5 \sum_{j=2}^{n} (t_j + 1) = c_5 \sum_{j=2}^{n} (0 + 1) = c_5 (n - 1)$$

$$c_6 \sum_{j=2}^{n} (t_j) = c_7 \sum_{j=2}^{n} (t_j) = \sum_{j=2}^{n} (0) = 0$$

$$T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 (n-1) + c_8 (n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8) n - (c_2 + c_4 + c_5 + c_8)$$

$$\approx an + b$$

Curtin University of Technology

Analysis of Insertion Sort

Worst Case: the **longest running time** for *an* input of size n

→ the array is in reverse sorted order

→ Must compare each $A[j]$ with each element in sorted sub-array $A[1 \ldots j-1]$

→ $t_j = j - 1$, for $j = 2, 3, \ldots, n$

Thus,

$$c_5 \sum_{j=2}^{n} (t_j + 1) = c_5 \sum_{j=2}^{n} (j - 1 + 1) = c_5 \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$c_6 \sum_{j=2}^{n} (t_j) = c_6 \sum_{j=2}^{n} (j-1) = c_6 \frac{n(n-1)}{2} \text{ and } c_7 \sum_{j=2}^{n} (j-1) = c_7 \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

$$\rightarrow an^2 + bn + c$$

Curtin University of Technology

Growth of functions

Consider $an^2 + bn + c$.

- For large n, the value of $bn + c$ is relatively insignificant to the value of an^2

Consider an^2

- For large n, the constant coefficient a is less significant than the rate of growth of n

Thus, we express $an^2 + bn + c$ as $\Theta(n^2)$

→ Read as "Big theta of n-squared"

Curtin University of Technology

$O(n)$ – Big-oh of n

Asymptotic *upper* bound

- A given function $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that:

$$0 \le f(n) \le c\, g(n) \text{ for all } n \ge n_0$$

- $O(g(n))$ represents a set of functions

$$\left\{ f(n) : \exists c > 0, n_0 > 0 \text{ such that } 0 \le f(n) \le c \cdot g(n), \forall n \ge n_0 \right\}$$

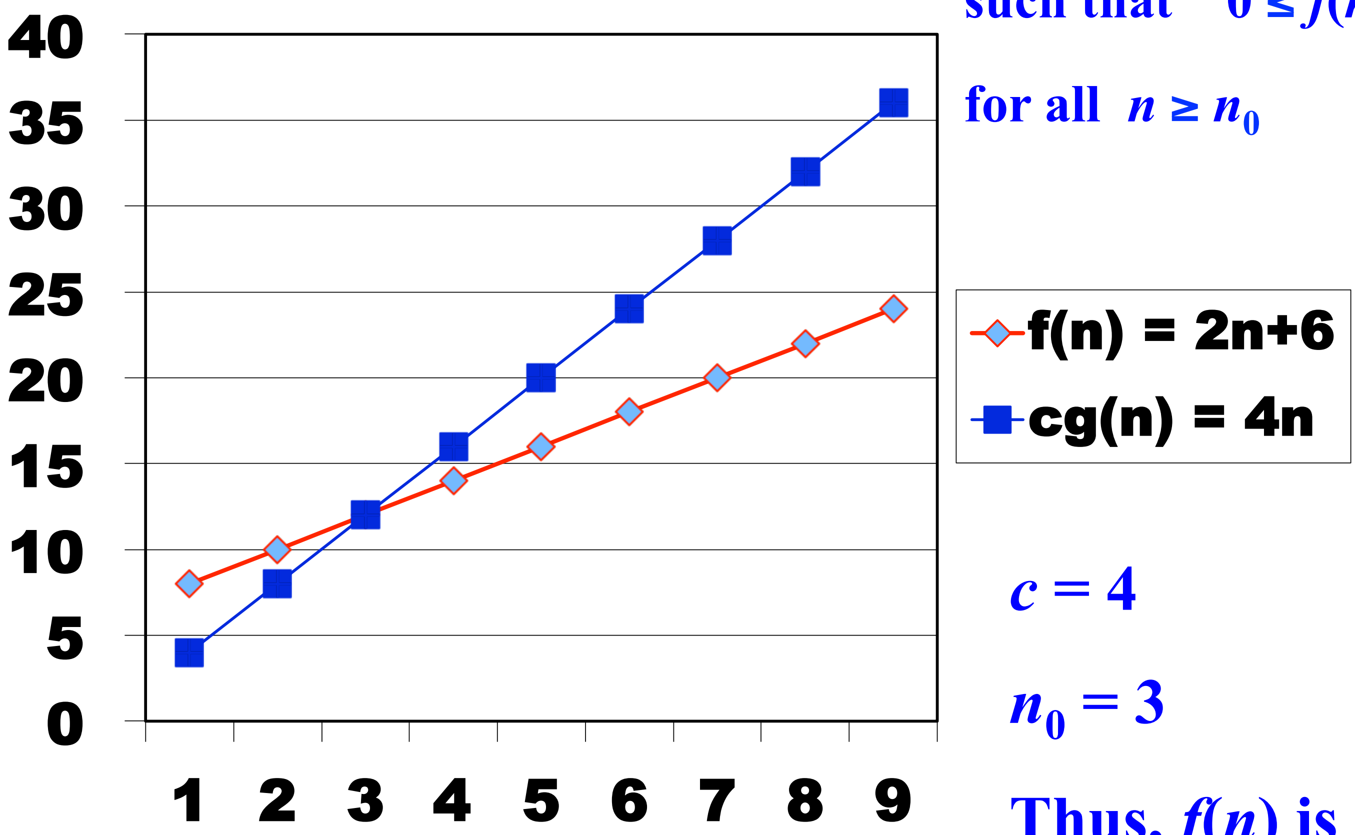
$\exists$ is read as: there exists, there is, or there are;

$\forall$ is read as: for all, for any, or for each

Curtin University of Technology

Big O Progression

$f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that $0 \le f(n) \le c\, g(n)$ for all $n \ge n_0$



$c = 4$

$n_0 = 3$

Thus, $f(n)$ is $O(n)$

Curtin University of Technology

O - example

Show that $2n + 6 = O(n)$

- $0 \le 2n + 6 \le cn$ // definition of Big O
- $0 \le 2 + 6/n \le c$ // divide by $n \ge n_0$ (> 0)

- TRUE for $n \ge 1$ and $c \ge 8$
- TRUE for $n \ge 2$ and $c \ge 5$

You can find other possible constants, but you need to show only *one pair* of all possible constants

Curtin University of Technology

Back to the example

$$sum = 0$$

1 assignment

$$\text{for } i = 1 \text{ to } n$$

$$sum = sum + A[i]$$

- What is the order (big-O) of the first line?
  - It's a constant, so any constant would do as big-O.
    - The convention is to use $O(1)$
- This says that there is some constant c such that

  $0 \le 1 \le c \times 1$ for all sufficient high value of n.
- It can be seen that any $c \ge 1$ would do.

Curtin University of Technology

Back to the example

$$sum = 0$$

$O(1)$

$$\text{for } i = 1 \text{ to } n$$

n+1 assignments and n+1 comparisons

$$sum = sum + A[i]$$

- What is the order of the second line?
  - It is executed (n+1) times and contains one comparison and one assignment for a total complexity of 2n+2.
- What is a suitable upper bound for a sufficiently large n?
  - 3n, 4n, 5.732n, 2πn, and other possible values of c and n
  - This makes it $O(n)$.

Curtin University of Technology

Back to the example

$$sum = 0 \qquad O(1)$$

$$\text{for } i = 1 \text{ to } n \qquad O(n)$$

$$\qquad sum = sum + A[i] \qquad \left.\begin{array}{l}\text{1 assignment}\\ \text{2 additions?}\\ \text{1 memory access}\end{array}\right\} n \text{ times}$$

- Four steps repeated n times is $O(n)$.
- Hence the fragment is $O(1) + O(n) + O(n)$ for a total of $O(n)$.
  - Because we can add individual constants, just take the largest term when adding.

Curtin University of Technology

Back to the example

- Alternative calculation:

	cost	**times**
$sum = 0$	c_1	1
for $i = 1$ to n	c_2	$n+1$
$\quad sum = sum + A[i]$	c_3	n

$$T(n) = c_1 + c_2\,(n+1) + c_3\,n = (c_1 + c_2) + (c_2 + c_3)\,n$$

$$= c_4 + c_5\,n \quad \rightarrow O\,(n)$$

Proof: $c_4 + c_5\,n \leq c\,n \rightarrow$ TRUE for $n \geq 1$ and $c \geq c_4 + c_5$

Simple Big-O

$$\left.\begin{array}{l} s \leftarrow 0 \\ s+1 \\ s \times 1 \\ s \leq 1 \end{array}\right\} \text{All } O(1)$$

---

for $i = 0$ to n

some $O(1)$ process

$$\sum_{i=0}^{n} O(1) = O(n)$$

Why?

If $f(n)$ is $O(1)$ then $\exists c_1, n_0$ s.t. $f(n) \le c_1 \quad \forall n > n_0$

Let the ith term in $\sum_{i=0}^{n} O(1)$ be bounded by $c_i, \forall n > n_i$

$$\sum_{i=0}^{n} O(1) \le c_1 + c_2 + \ldots + c_n, \forall n > \max(n_i)$$

$$\Rightarrow \sum_{i=0}^{n} O(1) \le n \times \max(c_1, c_2, \ldots, c_n)$$

$$\Rightarrow \sum_{i=0}^{n} O(1) \text{ is } O(n)$$

Curtin University of Technology

Simple Big-O (cont)

$$\left.\begin{array}{l} \text{for } i = 0 \text{ to } n \\ \qquad \text{for } j = 0 \text{ to } n \\ \qquad\qquad \text{some } O(1) \text{ process} \end{array}\right\} \sum_{i=0}^{n} \sum_{j=0}^{n} O(1) \text{ is } O(n^2)$$

$$\left.\begin{array}{l} \text{for } i = 0 \text{ to } n \\ \qquad \text{for } j = 0 \text{ to } i \\ \qquad\qquad \text{some } O(1) \text{ process} \end{array}\right\} \begin{array}{l} O(n^2) \\ O(f(n)) < O(n^2) \end{array}$$

Aim for the least upper bound

- Many algorithms are $O(2^n)$, but saying an algorithm is $O(2^n)$ is not always very useful if that isn't truly representative.
  - An $O(n)$ algorithm is $O(2^n)$; but an $O(2^n)$ algorithm is NOT $O(n)$
- Aim for the **smallest** $O(\ g\ (n)\ )$ expression
- If $f(n)$ is the smallest possible, then it is said to be a **tight upper bound;**
  - if $f(n)$ is $O(n^2)$, it is also $O(n^3)$, however, the first expression is better.

Note: tight *upper* bound is different from **asymptotic *tight* bound**

- **asymptotic tight bound** is called **big theta** (Θ)

$\Omega(n)$ – big-omega of n

Asymptotic *lower* bound

- A given function $f(n)$ is $\Omega(g(n))$ if there exist positive constants c and n_0 such that
  $0 \le c\,g(n) \le f(n)$ for all $n \ge n_0$

- $\Omega(g(n))$ represents a set of functions

$$\left\{ f(n) : \exists c > 0, n_0 > 0 \text{ and } 0 \le c \cdot g(n) \le f(n), \forall n \ge n_0 \right\}$$

Curtin University of Technology

Ω- example

Show that $2n + 6 = \Omega(n)$

$0 \leq c\,n \leq 2n + 6$ // definition of Big Ω

$0 \leq c \leq 2 + 6/n$ // divide by $n \geq n_0$ (> 0)

- TRUE for $n \geq 1$ and $c \leq 2$
- **Note:** $n \geq 1$ and $c \leq 4$ is not TRUE because for $c = 4$, when $n \geq 4$, $2 + 6/n$ can be no larger than 3.5, which is less than c

$\Theta(n)$ –Big-theta of n

Asymptotic *tight* bound

- A given function $f(n)$ is $\Theta(g(n))$ if there exist positive constants c_0 *and* c_1 and n_0 such that

  $$0 \leq c_0\, g(n) \leq f(n) \leq c_1\, g(n) \text{ for all } n \geq n_0$$

- $\Theta\,(g(n))$ represents a set of functions

  $$O(g(n)) \cap \Omega(g(n))$$

- $\Theta\,(g(n)) \subseteq O(g(n))$
- $\Theta\,(g(n)) \subseteq \Omega(g(n))$

Curtin University of Technology

Θ - Example

Show that $\frac{1}{2} n^2 - 3n = \Theta(n^2)$

- $c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2$ // divide by n^2, for $n \geq n_0$

  $c_1 \leq \frac{1}{2} - 3/n \leq c_2$
- RHS is true for $n \geq 6$ and $c_2 \geq \frac{1}{2}$
  - For $n < 6$, $\frac{1}{2} - 3/n \leq c_2$, but $\frac{1}{2} - 3/n < 0$, and thus not possible
  - The largest positive value of $\frac{1}{2} - 3/n$ is $\frac{1}{2}$
- LHS is true for $n \geq 7$ and $c_1 \leq 1/14$
  - For Ω, what is the smallest n such that $\frac{1}{2} - 3/n$ is positive?
    - Answer: $n_0 = 7$ such that $1/14$
- The n_0 for Θ is selected from the larger between the n_0 for O and Ω, i.e., max (6, 7) = 7
  - $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ for $c_1 = 1/14$, $c_2 = \frac{1}{2}$, and $n_0 = 7$

You can find other possible constants.

Curtin University of Technology

Θ - Example – an alternative proof

Show that $\frac{1}{2} n^2 - 3n = \mathbf{\Theta}(n^2)$

- Prove for Big O and Big Ω separately
- Use the larger n_0 between that for Big O and Big Ω

Big O

$\frac{1}{2} n^2 - 3n \leq c_2 n^2$

$\frac{1}{2} n^2 - 3n \leq \frac{1}{2} n^2$ // this is true for any $n \geq 1$; should set $n \geq 6$ since otherwise the value of LHS is negative

$\leq c_2 n^2$ for $c_2 \geq \frac{1}{2}$ // thus $n_0 = 6$

Big Ω

$\frac{1}{2} n^2 - 3n \geq c_1 n^2$ // divide by n^2, for $n \geq n_0$

$\frac{1}{2} - 3/n \geq c_1$ // true for $n \geq 7$ and $c_1 \leq 1/14$

Select from the larger between the n_0 for O and Ω, i.e., max (6, 7) = 7

- $\frac{1}{2}n^2 - 3n = \mathbf{\Theta}(n^2)$ for $c_1 = 1/14$, $c_2 = \frac{1}{2}$, and $n_0 = 7$

You can find other possible constants.

Curtin University of Technology

Other Examples

$$\left.\begin{array}{l} \text{for } i = 1 \text{ to } n \\ \qquad \text{for } j = 1 \text{ to } n \\ \qquad\qquad \text{some } O(1) \text{ process} \end{array}\right\} \begin{array}{l} O(n^2) \\ \Omega(n^2) \\ \Theta(n^2) \end{array}$$

$$\left.\begin{array}{l} \text{for } i = 1 \text{ to } n \\ \qquad \text{for } j = 1 \text{ to } i \\ \qquad\qquad \text{some } O(1) \text{ process} \end{array}\right\} \begin{array}{l} O(n^2) \\ \Omega(n) \end{array}$$

Are these tight?

Curtin University of Technology

Other Example

- Consider the following algorithm:

$$
\left.
\begin{array}{ll}
\text{for } i = 1 \text{ to } n & O(n) \\
\quad \text{if } A[i] = 0 & O(1) \\
\quad\quad \text{stop // exit the loop} & O(1) \\
\quad \text{else} & O(1) \\
\quad\quad \text{sum up elements in } A & O(n)
\end{array}
\right\}
$$

n times?
1 time?
$n/2$ times?

Curtin University of Technology

Worst case

- The n inputs are chosen so that the algorithm runs for as long as possible

$$\left.\begin{array}{lr} \text{for } i = 1 \text{ to } n & O(n) \\ \quad \text{if } A[i] = 0 & O(1) \\ \quad\quad \text{stop} & O(1) \\ \quad \text{else} & O(1) \\ \quad\quad \text{sum up elements in } A & O(n) \end{array}\right\}$$

n times?
1 time?
$n/2$ times?

Worst case running time is $O(n^2)$, when $A[i] \neq 0$ for $1 \leq i \leq n$

Curtin University of Technology

Best case

- The n inputs are chosen so that the algorithm runs for as little as possible

$$
\left.
\begin{array}{ll}
\text{for } i = 1 \text{ to } n & O(n) \\
\quad \text{if } A[i] = 0 & O(1) \\
\quad\quad \text{stop} & O(1) \\
\quad \text{else} & O(1) \\
\quad\quad \text{sum up elements in } A & O(n)
\end{array}
\right\}
\begin{array}{l}
n \text{ times?} \\
1 \text{ time?} \\
n/2 \text{ times?}
\end{array}
$$

Best case running time is $O(1)$, when A[1] = 0

Average case

- The n inputs are assumed to be chosen uniformly random
- e.g., if A were a permutation of $[1..n]$

$$\left.\begin{array}{ll}
\text{for } i = 1 \text{ to } n & O(n) \\
\quad \text{if } A[i] = 0 & O(1) \\
\quad\quad \text{stop} & O(1) \\
\quad \text{else} & O(1) \\
\quad\quad \text{sum up elements in } A & O(n)
\end{array}\right\}
\begin{array}{l}
1/n \text{ times, stop} \\
1/n \text{ times, 1 sum} \\
1/n \text{ times, 2 sums} \\
1/n \text{ times, 3 sums} \\
\cdots
\end{array}$$

Average case running time is $O(n^2)$

Curtin University of Technology

Expected case

- What is the usual or normal input?
- Still of size n
- Can be different from average case
- *e.g.*, a compression algorithm tailored to English text would "expect" words of average length 5 characters.

Algorithm time analysis – Summary

- Choose model of computation
- Choose case
  - Worst case,
  - Average case, or
  - Best case
- You are NOT free to choose size of the problem n
  - Usually interested in large n
- Count the steps and report asymptotic time using O, Ω, *or* Θ
  - Express the bound as tight as possible
    - If $f(n)$ is $O(n^2)$, it is also $O(n^3)$; present it as $O(n^2)$
    - If $f(n)$ is $\Omega(n^2)$, it is also $\Omega(n)$; present it as $\Omega(n^2)$

Curtin University of Technology

Space or memory analysis

- A model of computation should define a $O(1)$ unit of memory
  - a register or word size ω bits
- In RAM
  - Numbers use $O(1)$ space
  - $A[n]$ uses $O(n)$ space
  - Binary tree of n leaves is $O(???)$
- All assumed data units are less than 2^{ω}

Curtin University of Technology

Selection sort

SELECTION_SORT (A[1..n])
Input: unsorted array A
Output: sorted array A

1. **for** $i = 1$ to $n - 1$
2. $\quad small = i$
3. $\quad$ **for** $j = i+1$ to n
4. $\quad\quad$ **if** $A[j] < A[small]$ **then**
5. $\quad\quad\quad small = j$
6. $\quad temp = A[small]$
7. $\quad A[small] = A[i]$
8. $\quad A[i] = temp$

Set *small* as the pointer to the smallest element in $A[i..n]$

Swap $A[i]$ and the smallest

Curtin University of Technology

Selection Sort -Example

<u>5</u> 2 4 6 <u>1</u> 3 → 1 <u>2</u> 4 6 5 3 → 1 2 <u>4</u> 6 5 <u>3</u>

i *small* *i* *i* *small*

small

→ 1 2 3 <u>6</u> 5 <u>4</u> → 1 2 3 4 <u>5</u> 6 → 1 2 3 4 5 6

i *small* *i*

small

SELECTION_SORT (*A*[1..*n*])
Input: unsorted array *A*
Output: sorted array *A*

```
for i = 1 to n – 1
    small = i
    for j = i+1 to n
        if A[j] < A[small] then
            small = j
    temp = A[small]
    A[small] = A[i]
    A[i] = temp
```

$\Theta(1)$

$\Theta(1)$ $\quad \sum_{j=i+1}^{n} \cdots$

$\Theta(1)$

$$\sum_{i=1}^{n-1} \cdots$$

Curtin University of Technology

Selection Sort – time complexity

$$\text{Steps } 3, 4, 5 \text{ take } \sum_{j=i+1}^{n} \Theta(1) \text{ time}$$

$$= \sum_{j=1}^{n} \Theta(1) - \sum_{j=1}^{i} \Theta(1)$$

$$= \Theta(n) - \Theta(i)$$

$$\text{so Step 1 to 8 takes } \sum_{i=1}^{n-1} \big(\Theta(1) + \Theta(n) - \Theta(i) + \Theta(1)\big) \text{time}$$

$$= \Theta(n) + \Theta(n^2) - \sum_{i=1}^{n-1} \Theta(i) + \Theta(n)$$

$$= \Theta(n^2)$$

Proof by Induction

- Induction is very similar to recursion
  - Proof by induction is very useful in analyzing **recursive** algorithms
- To prove a statement $S(n)$, proof by induction requires proving two cases:
  - **Base case:** prove statement $S(n)$ only for a small value of n, e.g., $n = 1$ or $n = 2$.
  - **Inductive step**: prove that if the statement is true for a smaller value of n, for any value of $k \leq n$, i.e., $S(k)$ is true, it is still true for a larger value of k, i.e., $S(k+1)$
    - The $S(k)$ that is assumed to be true is called *induction hypothesis*
    - You can also set $S(n)$ as the induction hypothesis, and prove that it is true for $S(n+1)$

Curtin University of Technology

Induction – example 1

Prove $\sum_{i=i}^{n} i = \frac{n(n+1)}{2}$

Base case: $n = 1$; LHS = 1; RHS = $\frac{1(1+1)}{2} = 1$. So, it is true for the base case

Induction step: assume it is true for $k < n$; *i.e.*, we use induction hypothesis: $\sum_{i=i}^{k} i = \frac{k(k+1)}{2}$

We want to prove that it is also true for $k+1$, i.e., $\sum_{i=i}^{k+1} i = \frac{(k+1)(k+2)}{2}$

$$LHS = \sum_{i=i}^{k+1} i = (k+1) + \sum_{i=i}^{k} i = (k+1) + \frac{k(k+1)}{2} = \frac{2(k+1)+k(k+1)}{2} = \frac{(k+1)(k+2)}{2} = RHS$$

This completes the proof.

Curtin University of Technology

Induction – example 2

$$F(n) = \begin{cases} 1, & \text{if } n < 3 \\ F(n\text{-}1) + F(n\text{-}2), & \text{otherwise} \end{cases}$$

The following is a sequence of *Fibonacci* number $F(n)$:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

Problem: Prove that the nth Fibonacci number is $O(\Phi^n)$

Note: Φ is the root of $x^2 - x - 1 = 0$

$$\Phi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

Curtin University of Technology

Induction proof -Example

- By the definition of Big Oh, prove $F(n) \le c\ \Phi^n$
  - for some constants c and n_0
- **Inductive hypothesis**

$F(N) \le c \cdot \Phi^N$ for some $c > 1$ and all $N < n$

  - i.e., $F(N\text{-}1) \le c\ \Phi^{N-1}$ and $F(N\text{-}2) \le c\ \Phi^{N-2}$
- **Inductive Step**:

$$
\begin{aligned}
F(N) &= F(N-1) + F(N-2) \\
&\le c \cdot \Phi^{N-1} + c \cdot \Phi^{N-2} \\
&= c \cdot \Phi^N (\Phi^{-1} + \Phi^{-2})
\end{aligned}
$$

$F(N) \le k \cdot \Phi^N$; for $k \ge c \cdot (\Phi^{-1} + \Phi^{-2})$ and $N > 3$

- **Base case** $F(1) = 1 \le c \cdot \Phi^1 \quad \forall c > 1$

This completes the proof

Curtin University of Technology

How to analyze a recursive function?

- Use **recurrence** to analyze the time complexity of a recursive function
  - E.g., $T(n) = T(n - 1) + n$, $T(n) = 2T(n/2) + n^2$, etc
- A **recurrence** is a function defined by:
  - One or more base cases
  - Itself with smaller arguments.

<u>Methods for solving recurrence:</u>

- Iteration - not recommended; easy to make mistake!
- Master method if $T(n) = aT(n/b) + f(n)$
  - $T(n) = 2T(n/2) + n \rightarrow$ YES
  - $T(n) = T(n - 1) + n \rightarrow$ NO
- Substitution method (guess and induction)
  - Use this method if the recurrence cannot be solved by master method
  - Use recursion-tree method for the guess in induction

Curtin University of Technology

Selection Sort (recursive) – analysis

Input: Unsorted array A; **Output:** Sorted array A

RECURSIVE_SELECTION_SORT(*A*, *n*, *i*)

1. **if** $i = n$ **then** *stop*
2. Find the smallest element in $A[i..n]$ // at *small*
3. Swap elements $A[i]$ and $A[small]$
4. RECURSIVE_SELECTION_SORT(A, n, i+1)

- Line 1 takes $O(1)$
- Line 2 takes $O(n)$
- The size of the problem is reduced by 1 after each recursion → $T(n-1)$ time
- Thus, the recurrence time is $T(n) = T(n-1) + O(n)$

To run: RECURSIVE_SELECTION_SORT(A, n, 1) // start from $i = 1$

Analysis:

Let $T(n)$ be the time for this algorithm to operate on n elements

The **recurrence** for time complexity of Selection Sort is

$$T(1) = \Theta(1) \quad \text{base case (Step 1)}$$

$$T(n) = T(n-1) + O(n) \quad \text{find small \& recursive call}$$

How to solve $T(n)$?

Curtin University of Technology

Analysis using iteration method

$T(n) = T(n-1) + O(n)$ → Cannot use the master method!

Note

Triangular number sequence:

1, 3, 6, 10, 15, …, $k(k+1)/2$

Using *Iteration Method*:

$T(n) = T(n-1) + O(n)$ // overall → iteration $k = 0$

$T(n-1) = T(n-2) + (n-1)$ // iteration $k = 1$

→ $T(n) = \{T(n-2) + (n-1)\} + n = T(n-2) + 2n - 1$ // $1 = 1\,(1+1)/2$

$T(n-2) = T(n-3) + (n-2)$ // iteration $k = 2$

→ $T(n) = \{T(n-3) + (n-2)\} + 2n - 1 = T(n-3) + 3n - 3$ // $3 = 2\,(2+1)/2$

$T(n-3) = T(n-4) + (n-3)$ // iteration $k = 3$

→ $T(n) = \{T(n-4) + (n-3)\} + 3n - 3 = T(n-4) + 4n - 6$ // $6 = 3\,(3+1)/2$

$T(n-4) = T(n-5) + (n-4)$ // iteration $k = 4$

→ $T(n) = \{T(n-5) + (n-4)\} + 4n - 6 = T(n-5) + 5n - 10$ // $10 = 4\,(4+1)/2$ → 4^{th} triangular number

So in general: $T(n) = T(n-(k+1)) + (k+1)\,n - (k+1)\,k/2$, where k = *iteration #*

$T(n) = T(n-(k+1)) + (k+1)\,(2n-k)/2$

When $k = n - 2$, we have

$T(n) = T(1) + (n-1)\,(n+2)/2$ // T(1) is base case: 1 element array = $O(1)$

$= O(1) + O(n^2) = \boldsymbol{O(n^2)}$

Solution using Guess and Induction

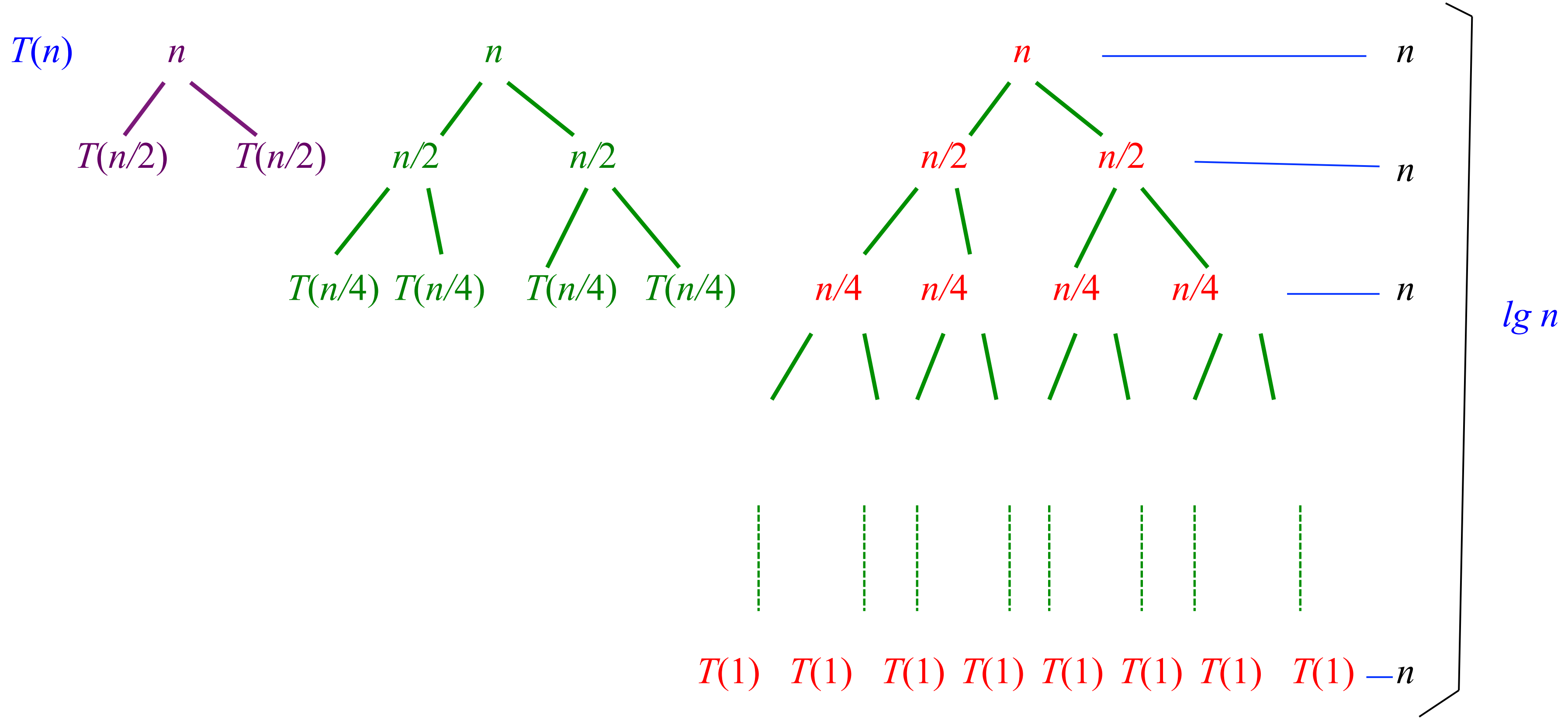
- Guess the solution for $T(n)$ is $O(f(n))$
- Prove $T(n)$ is $O(f(n))$
- Assume answer for small n
  - *e.g.*, $n = 2$ or $n = 1$
- Show it holds for large n
- Show it holds for base case
  - Can use n_0 to exclude nasty cases (*e.g.*, when $n = 1 \rightarrow \log_2 1 = 0$)

Curtin University of Technology

Guess and Induction – How to guess?

Solve $T(n) = 2T(\lfloor n/2 \rfloor) + n$

Use recursion tree to guess the solution for $T(n)$



- The fully expanded tree has height of $log_2\ n$
- Each level takes n time
- Thus, the total time is $n\ log_2\ n$

Note: Ignore the floor or ceiling functions!

Guess and Induction – How to prove by induction?

The guess for the solution of $T(n)$ is $O(n \log n)$

Use Induction to show the solution of $T(n) = 2T\left(\lfloor n/2 \rfloor\right) + n$ is $O(n \log n)$

Assume $T(x) \le cx \log x$ holds when $x = \lfloor n/2 \rfloor$

$$
\begin{aligned}
T(n) &\le 2c\lfloor n/2 \rfloor \log\lfloor n/2 \rfloor + n \\
&\le cn \log(n/2) + n \\
&= cn \log n - cn \log 2 + n \\
&= cn \log n - n(c-1) \\
&\le cn \log n \qquad \text{if } c \ge 1
\end{aligned}
$$

Now for the base case when $n = 1$

$T(1) = 2T(0) + 1 = 1 \le c.1.\log(1) = 0$ --> 1 ≤ 0 ???

But, assume $n_0 > 1$, so base case is $n = 2$

$T(2) = 2T(1) + 2 = 4 \le c.2.\log(2)$ choose $c \ge 2$

Master method

if $T(n) = aT(n/b) + f(n)$ then use the following master theorem

$$
T(n) = \begin{cases}
\Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{(\log_b a)-\varepsilon}\right);\ \text{means}\ f(n) < n^{\log_b a} \quad \text{Case 1} \\
\Theta\left(n^{\log_b a} \lg n\right) & f(n) = \Theta\left(n^{\log_b a}\right);\ \text{means}\ f(n) = n^{\log_b a} \quad \text{Case 2} \\
\Theta\left(f(n)\right) & f(n) = \Omega\left(n^{(\log_b a)+\varepsilon}\right);\ \text{means}\ f(n) > n^{\log_b a} \quad \text{Case 3} \\
 & \text{if } af(n/b) \le cf(n) \text{ for c} < 1 \text{ and large } n
\end{cases}
$$

Curtin University of Technology

Master method (cont.)

- **Case 1** must meet *polynomial* condition
  - $f(n)$ must be asymptotically **smaller** than $n^{\log_b a}$ by a factor of n^{ε}, for some constant $\varepsilon > 0$.

- **Case 3** must meet *polynomial* and *regularity* conditions
  - $f(n)$ must be asymptotically **larger** than $n^{\log_b a}$ by a factor of n^{ε}, for some constant $\varepsilon > 0$.
  - Regularity condition: $a\, f(n/b) \le c\, f(n)$
    - This condition can be ignored

Master Theorem – example (case 1)

Give asymptotic tight bound for

$$T(n) = 9T(n/3) + n$$

Solution:

$a = 9$, $b = 3$, and $f(n) = n$.

$$n^{\log_b a} = n^{\log_3 9} = n^2 \rightarrow f(n) < n^{\log_b a}$$ → Case 1

Does it satisfy the polynomial condition?

$$f(n) = O(n^{\log_3 9-\varepsilon}) \text{ for } \varepsilon = 1 \rightarrow \text{YES} \qquad \therefore T(n) = \Theta(n^2)$$

Alternatively …

- Compute the ratio between $n^{\log_b a}$ and $f(n)$
- If the ratio is $O(n^{\varepsilon})$, it satisfies the polynomial condition
- e.g., $n^2 / n = n^1$ → n^2 is asymptotically larger than n by a factor of n^{ε} for $\varepsilon = 1$

Curtin University of Technology

Master Theorem – example (case 2)

Give asymptotic tight bound for

$$T(n) = T(2n/3) + 1$$

Solution:

$a = 1$, $b = 3/2$, and $f(n) = 1$.

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1 \rightarrow f(n) = n^{\log_b a} \quad \rightarrow \text{Case 2}$$

$$\text{or} \quad f(n) = \Theta(n^{\log_b a})$$

$$\therefore T(n) = \Theta(\log n)$$

Curtin University of Technology

Master Theorem – example (case 3)

Give asymptotic tight bound for

$$T(n) = 3T(n/4) + n \log n$$

Solution:

$a = 3$, $b = 4$, and $f(n) = n \log n$

$$n^{\log_b a} = n^{\log_4 3} = n^{0.793} \rightarrow f(n) > n^{\log_b a}$$ → Case 3

Does it satisfy the polynomial condition?

$$f(n) = \Omega(n^{\log_4 3+\varepsilon}), \text{ for } \varepsilon \approx 0.2 \text{ or } f(n) > n^{\log_4 3} \quad \rightarrow \quad \text{YES}$$

Alternatively …

$$f(n)/n^{\log_b a} = n \lg n / n^{\log_4 3} = n \lg n / n^{0.793} \geq n^{\varepsilon} \text{ for } \varepsilon \approx 0.2$$

Note: $n \lg n \geq c.n^{\log_4 3}.n^{0.2}$

Case 3 (cont.)

Does it satisfy the regularity condition? $a\,f(n/b) \le c\,f(n)$

$a = 3$, $b = 4$, and $f(n) = n \log n$

$a\,f(n/b) = 3\,(n/4) \log (n/4)$

$\le (3/4)\, n \log n$

$\le c\,f(n)$, for $c \ge 3/4$ $\rightarrow$ YES

Following case 3: $T(n) = \Theta(n \lg n)$

Another case 3 example

Give asymptotic tight bound for

$$T(n) = 2T(n/2) + n \log n$$

Solution:

$a = 2$, $b = 2$, and $f(n) = n \log n$

$$n^{\log_b a} = n^{\log_2 2} = n \rightarrow f(n) > n^{\log_b a}$$ → Case 3

Does it satisfy the polynomial condition? NO

$f(n)/n^{\log_b a} = (n \log n)/n = \log n$ $\rightarrow \log n$ is asymptotically less than n^{ε}

Thus, we cannot use the master method to solve the recurrence

How to prove an algorithm is correct?

- Use *loop invariant* to prove its correctness.
- Three things to look at in loop invariant:
  - **Initialization**: *It is true prior to the first iteration of the loop*
  - **Maintenance**: *If it is true before an iteration of the loop, it remains true before the next iteration*
  - **Termination**: *When the loop terminates, the invariant gives a useful property that help shows that the algorithm is correct*
- The proof using loop invariant is similar to the proof by *induction*:
  - **Initialization** is the **base case** in induction
  - **Maintenance** is the **inductive step** in induction
  - **Termination** – there is a condition that makes the loop to terminate
    - It is different from that in induction that uses inductive step *infinitely*.

Curtin University of Technology

Is the insertion sort correct?

INSERTION-SORT (*A*)

```
INSERTION-SORT (A)
    for j = 2 to length(A) do
        key = A[j]
        // insert A[j] into the sorted sequence A[1 … j – 1]
        i = j – 1
        while i > 0 and A[i] > key do  // Line 4
            A[i + 1] = A[i]
            i = i – 1
        A[i+1] = key
```

Loop invariant: At the start of the for loop, $A[1 .. j - 1]$ contains the same elements as its original contents but in sorted order

- **Initialization**:
  - when $j = 2$, subarray $A[1 \ldots j - 1]$ contains only one *sorted* element $A[1]$
    - it holds prior to the first loop iteration.
- **Maintenance**:
  - For each $key = A[j]$, assume it is true that $A[1 \ldots j - 1]$ is a sorted sequence
    - We need to show that after each loop, $A[1 \ldots j]$ is also sorted.
    - Line 4-7 move each $A[j - 1]$ to its next position on right until the right position for *key* is found.
      - Thus, after inserting *key*, $A[1 \ldots j]$ is a sorted sequence, which will be used for the next iteration of the loop, with $j + 1$.

Is the insertion sort correct? (cont.)

INSERTION-SORT (A)

for $j = 2$ to $length(A)$ **do**
  $key = A[j]$
  // insert $A[j]$ into the sorted sequence $A[1 \ldots j\text{-}1]$
  $i = j - 1$
  while $i > 0$ and $A[i] > key$ **do**
    $A[i+1] = A[i]$
    $i = i - 1$
  $A[i+1] = key$

- **Termination**:
  - The loop terminates when $j = length(A) = n$. Each iteration always increments the value of j by one.
  - Thus, eventually, the value of j will reach $n + 1$, and the loop will terminate with a *sorted* $A[1 \ldots n]$.

- Hence the insertion sort is correct.

NOTE: A more complex algorithm may require a more complex / formal proof to show its correctness.

Is the selection sort correct? (cont.)

SELECTION_SORT (A[1 .. n])
Input : unsorted array A
Output : sorted array A

1. **for** $i = 1$ to $n - 1$
2. $small = i$
3. **for** $j = i + 1$ to n
4. **if** $A[j] < A[small]$ **then**
5. $small = j$
6. $temp = A[small]$
7. $A[small] = A[i]$
8. $A[i] = temp$

Loop invariant: At the start of each iteration of the loop at Line 1, subarray $A[1 \;..\; i - 1]$ is in sorted order and it is a subset of the original array A

Initialization: When $i = 1$, subarray $A[1 \;..\; 1]$ contains only one *sorted* element of A

- It holds prior to the first loop iteration.

Maintenance: For each $i > 1$, assume it is true that $A[1 \ldots i - 1]$ is a sorted sequence and it is a subset of the original array A

- We need to show that after each loop, $A[1 \ldots i]$ is also sorted and a subset of the original array A.
- Line 3-5 find *small* as the index to the smallest element in $A[i \;..\; n]$
- Line 6 and 7 swap the content of $A[i]$ with $A[small]$.
- After Line 8, $A[i]$ contains the next smallest element after those in $A[1 \ldots i - 1]$, and $A[1 \ldots i]$ is a sorted sequence, to be used in the next iteration $i + 1$. Further, $A[i]$ is an element of the original array A, and $A[i + 1 \;..\; n]$ contains the remaining unsorted elements

Termination: The loop terminates when $i = n - 1$. Each iteration always increments the value of i by one.

- Thus, eventually, the value of i will reach $n - 1$, and the loop will terminate with a *sorted* $A[1 \ldots n - 1]$ and $A[n]$ contains the largest element after possible swap in Line 6 to 8, and hence $A[1 \ldots n]$ is sorted

Curtin
University of Technology