

## Topic 16

# Analysis of Algorithms

12-1

## Analysis of Algorithms- Review

- **Efficiency** of an algorithm can be measured in terms of :
  - **Time complexity**: a measure of the amount of time required to execute an algorithm
  - **Space complexity**: amount of memory required
- Which measure is more important?
  - It often depends on the limitations of the technology available at time of analysis (e.g. processor speed vs memory space)

12-2

## Time Complexity Analysis

- **Objectives** of time complexity analysis:
  - To determine the feasibility of an algorithm by estimating an **upper bound** on the amount of work performed
  - To compare different algorithms before deciding which one to implement
- Time complexity analysis for an algorithm is **independent** of the programming language and the machine used

12-3

## Time Complexity Analysis

- Time complexity expresses the relationship between
  - the **size of the input**
  - and the **execution time** for the algorithm
- Usually expressed as a **proportionality**, rather than an exact function

12-4

## Time Complexity Measurement

- Essentially based on the number of **basic operations** in an algorithm:
  - Number of arithmetic operations performed
  - Number of comparisons
  - Number of times through a critical loop
  - Number of array elements accessed
  - etc.
- Think of this as the **work** done

12-5

## Example: Polynomial Evaluation

Consider the polynomial

$$P(x) = 4x^4 + 7x^3 - 2x^2 + 3x + 6$$

Suppose that exponentiation is carried out using multiplications. Two ways to evaluate this polynomial are:

**Brute force method:**

$$P(x) = 4*x*x*x*x + 7*x*x*x - 2*x*x + 3*x + 6$$

**Horner's method:**

$$P(x) = (((4*x + 7) * x - 2) * x + 3) * x + 6$$

12-6

## Method of analysis

- What are the **basic operations** here?
  - multiplication, addition, and subtraction
    - We'll only consider the number of multiplications, since the number of additions and subtractions are the same in each solution
- We'll examine the **general form** of a polynomial of degree **n**, and express our result **in terms of n**
- We'll look at the **worst case** (maximum number of multiplications) to get an **upper bound** on the work

12-7

## Method of analysis

**General form** of a polynomial of degree **n** is

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \dots + a_1 x^1 + a_0$$

where  $a_n$  is **non-zero** for all  $n \geq 0$

(this is the worst case)

12-8

## Analysis of Brute Force Method

$$\begin{array}{l}
 P(x) = a_n \cdot \underbrace{x \cdot x \cdot \dots \cdot x \cdot x}_n + \\
 a_{n-1} \cdot \underbrace{x \cdot x \cdot \dots \cdot x \cdot x}_{n-1} + \\
 a_{n-2} \cdot \underbrace{x \cdot x \cdot \dots \cdot x \cdot x}_{n-2} + \\
 \dots + \\
 a_2 \cdot \underbrace{x \cdot x}_2 + \\
 a_1 \cdot \underbrace{x}_1 + \\
 a_0
 \end{array}$$

**n** multiplications  
**n-1** multiplications  
**n-2** multiplications  
 ...  
**2** multiplications  
**1** multiplication

12-9

Number of multiplications needed in the **worst case** is

$$\begin{aligned}
 T(n) &= n + (n-1) + (n-2) + \dots + 3 + 2 + 1 \\
 &= n(n+1)/2 \quad (\text{see below}) \\
 &= n^2/2 + n/2
 \end{aligned}$$

Sum of first **n** natural numbers:

Write the **n** terms of the sum in forward and reverse orders:

$$T(n) = 1 + 2 + 3 + \dots + (n-2) + (n-1) + n$$

$$T(n) = n + (n-1) + (n-2) + \dots + 3 + 2 + 1$$

Add the corresponding terms:

$$\begin{aligned}
 2 \cdot T(n) &= (n+1) + (n+1) + (n+1) + \dots + (n+1) + (n+1) + (n+1) \\
 &= n(n+1)
 \end{aligned}$$

Therefore,  $T(n) = n(n+1)/2$

## Analysis of Horner's Method

$$\begin{array}{l}
 P(x) = ( \dots ( ( a_n \cdot x + \\
 a_{n-1} ) \cdot x + \\
 a_{n-2} ) \cdot x + \\
 \dots + \\
 a_2 ) \cdot x + \\
 a_1 ) \cdot x + \\
 a_0
 \end{array}$$

**1** multiplication  
**1** multiplication  
**1** multiplication  
 ...  
**1** multiplication  
**1** multiplication  
**n times**

12-11

## Analysis of Horner's Method

Number of multiplications needed in the **worst case** is :

$$T(n) = n$$

12-12

## Big-O Notation

- Analysis of Brute Force and Horner's methods came up with **exact formulae** for the maximum number of multiplications
- In general, though, we want **upper bounds** rather than exact formulae: we will use the Big-O notation introduced earlier ...

12-13

## Big-O : Formal Definition

- **Time complexity  $T(n)$**  of an algorithm is  **$O(f(n))$**   
( we say "of the order  $f(n)$ " )  
if for some positive constant **C**  
and for all but finitely many values of **n**  
(i.e. as **n** gets large)  
 **$T(n) \leq C * f(n)$**
- What does this mean? this gives an **upper bound** on the number of operations, **for sufficiently large n**

12-14

## Big-O Analysis

- We want our  **$f(n)$**  to be an easily recognized **elementary function** that describes the performance of the algorithm

12-15

## Big-O Analysis

Example: Polynomial Evaluation

- What is  **$f(n)$**  for Horner's method?
  - **$T(n) = n$** , so choose  **$f(n) = n$**
  - So, we say that the number of multiplications in Horner's method is  **$O(n)$**  ("of the order of  $n$ ") and that **the time complexity of Horner's method is  $O(n)$**

12-16

## Big-O Analysis

Example: Polynomial Evaluation

- What is  **$f(n)$**  for the Brute Force method?
  - Choose the highest order (**dominant**) term of  
 **$T(n) = n^2/2 + n/2$**   
with a coefficient of **1**, so that  
 **$f(n) = n^2$**

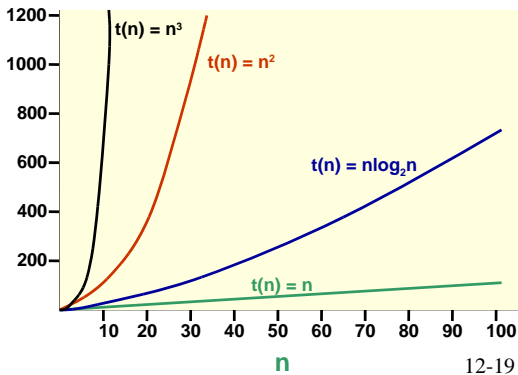
12-17

## Discussion

- Why did we use the dominant term?
  - It determines the basic **shape** of the function
- Why did we use a coefficient of 1?
  - For large **n**,  **$n^2/2$**  is essentially  **$n^2$**

12-18

## Recall: Shape of Some Typical Functions



12-19

1-19

## Big-O Analysis

### Example: Polynomial Evaluation

- Is  $f(n) = n^2$  a good choice? i.e. for large  $n$ , how does  $T(n)$  compare to  $f(n)$ ?
  - $T(n)/f(n)$  approaches  $\frac{1}{2}$  for large  $n$
  - So,  $T(n)$  is approximately  $n^2/2$  for large  $n$
- $n^2/2 \leq T(n) \leq n^2$ , so  $f(n)$  is a close upper bound

12-20

## Big-O Analysis

### Example: Polynomial Evaluation

- So, we say that the number of multiplications in the Brute Force method is  $O(n^2)$  ("of the order of  $n^2$ ") and that *the time complexity of the Brute Force method is  $O(n^2)$*
- Think of this as "*proportional to  $n^2$* "

12-21

## Big-O Analysis: Summary

- We want  $f(n)$  to be an easily recognized elementary function
- We want a *tight upper bound* for our choice of  $f(n)$ 
  - $n^3$  is also an upper bound for the Brute Force method, but not a good one!
    - Why not? Look at how  $T(n)$  compares to  $f(n) = n^3$ 
      - $T(n) / n^3$  approaches 0 for large  $n$ , i.e.  $T(n)$  is miniscule when compared to  $n^3$  for large  $n$

12-22

## Big-O Example: Polynomial Evaluation Comparison

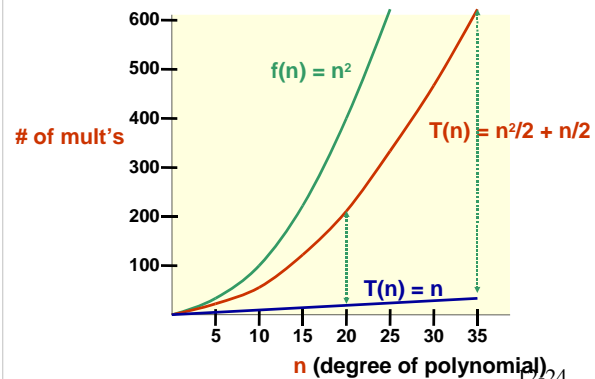
$n$ and $T(n) = n$ (Horner)	$T(n) = n^2/2 + n/2$ (Brute Force)	$f(n) = n^2$ (upper bound for Brute Force)
5	15	25
10	55	100
20	210	400
100	5050	10000
1000	500500	1000000

$n$  is the degree of the polynomial.

Recall that we are comparing the number of multiplications.

12-23

## Big-O Example: Polynomial Evaluation



12-24

## Time Complexity and Input

- Run time can depend on the **size of the input** only (e.g. sorting 5 items vs. 1000 items)
- Run time can also depend on the **particular input** (e.g. suppose the input is already sorted)
- This leads to several kinds of time complexity analysis:
  - **Worst case** analysis
  - **Average case** analysis
  - **Best case** analysis

12-25

## Worst, Average, Best Case

- **Worst case analysis**: considers the **maximum** of the time over all inputs of size **n**
  - Used to find an upper bound on algorithm performance for large problems (large **n**)
- **Average case analysis**: considers the **average** of the time over all inputs of size **n**
  - Determines the average (or expected) performance
- **Best case analysis**: considers the **minimum** of the time over all inputs of size **n**

12-26

## Discussion

- What are some difficulties with average case analysis?
  - Hard to determine
  - Depends on distribution of inputs (they might not be evenly distributed)
- So, we usually use **worst case** analysis (why not best case analysis?)

12-27

## Example: Linear Search

- **The problem**: search an array **a** of size **n** to determine whether the array contains the value **key**
  - Return **array index** if found, **-1** if not found

```
Set k to 0.  
While (k < n-1) and (a[k] is not key)  
    Add 1 to k.  
If a[k] == key  
    Return k.  
Else  
    Return -1.
```

- Total amount of work done:
  - **Before loop**: a constant amount of work
  - **Each time through loop**: 2 comparisons and a constant amount of work (the **and** operation and addition)
  - **After loop**: a constant amount of work
  - **So, we consider the number of comparisons only**
  - **Worst case**: need to examine all **n** array locations
  - So, **T(n) = 2\*n**, and time complexity is **O(n)**

12-29

- Simpler (less formal) analysis:
  - Note that work done before and after the loop is independent of **n**, and work done during a single execution of loop is independent of **n**
  - In worst case, loop will be executed **n** times, so amount of work done is proportional to **n**, and algorithm is **O(n)**

12-30

- **Average** case for a **successful** search:
  - Number of comparisons necessary to find the key? **1** or **2** or **3** or **4** ... or **n**
  - Assume that each possibility is equally likely
  - Average number of comparisons :  

$$(1+2+3+ \dots +n)/n = (n*(n+1)/2)/n = (n+1)/2$$
  - Average case time complexity is therefore **O(n)**

12-31

## Example: Binary Search

- **General case:** search a **sorted** array **a** of size **n** looking for the value **key**
- **Divide and conquer** approach:
  - Compute the middle index **mid** of the array
  - If **key** is found at **mid**, we're done
  - Otherwise repeat the approach on the half of the array that might still contain **key**
  - etc...

12-32

## Binary Search Algorithm

```

Set first to 0.
Set last to n-1.
Do {
  Set mid to (first + last) / 2.
  If key < a[mid], Set last to mid - 1.
  Else Set first to mid + 1.
} While (a[mid] is not key) and (first <= last).
If a[mid] == key Return mid.
Else Return -1.

```

12-33

- Amount of work done before and after the loop is a constant, and is independent of **n**
- Amount of work done during a single execution of the loop is constant
- Time complexity will therefore be proportional to number of times the loop is executed, so that is what we will analyze

12-34

**Worst case:** **key** is not found in the array

- Each time through the loop, at least half of the remaining locations are rejected:
  - After **first** time through,  $\leq n/2$  remain
  - After **second** time through,  $\leq n/4$  remain
  - After **third** time through,  $\leq n/8$  remain
  - After **k<sup>th</sup>** time through,  $\leq n/2^k$  remain

12-35

- Suppose in the **worst case** that the maximum number of times through the loop is **k**; we must express **k** in terms of **n**
- Exit the do..while loop when the number of remaining possible locations is less than 1 (that is, when **first > last**): this means that  $n/2^k < 1$

12-36

- Also,  $n/2^{k-1} \geq 1$ ; otherwise, looping would have stopped after  $k-1$  iterations
- Combining the two inequalities, we get  $n/2^k < 1 \leq n/2^{k-1}$
- Invert and multiply through by  $n$  to get  $2^k > n \geq 2^{k-1}$

12-37

- Next, take base-2 logarithms to get

$$k > \log_2(n) \geq k-1$$

which is equivalent to

$$\log_2(n) < k \leq \log_2(n) + 1$$

- So, binary search algorithm is  $O(\log_2(n))$  in terms of the number of array locations examined

12-38

## Big-O Analysis in General

- To determine the time complexity of an algorithm:
  - Look at the loop structure
  - Identify the basic operation(s)
  - Express the number of operations as  $f_1(n) + f_2(n) + \dots$
  - Identify the **dominant term**  $f_i$
  - Then the time complexity is  $O(f_i)$

12-39

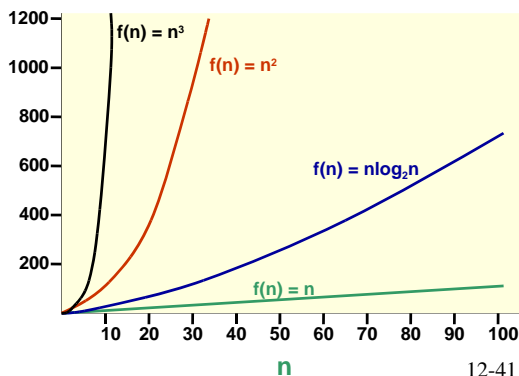
- **Examples of dominant terms:**
  - $n$  dominates  $\log_2(n)$
  - $n \log_2(n)$  dominates  $n$
  - $n^2$  dominates  $n \log_2(n)$
  - $n^m$  dominates  $n^k$  when  $m > k$
  - $a^n$  dominates  $n^m$  for any  $a > 1$  and  $m \geq 0$
- That is,

$$\log_2(n) < n < n \log_2(n) < n^2 < \dots < n^m < a^n$$

for  $a > 1$  and  $m > 2$

12-40

## Recall: Shape of Some Typical Functions



12-41

1-41

## Examples of Big-O Analysis

- **Independent nested loops:**

```
int x = 0;
for (int i = 1; i <= n/2; i++){
  for (int j = 1; j <= n*i; j++){
    x = x + i + j;
  }
}
```

- Number of iterations of inner loop is **independent** of the number of iterations of the outer loop (*i.e.* the value of  $i$ )
- How many times through outer loop?
- How many times through inner loop?
- Time complexity of algorithm?

12-42

- **Dependent nested loops:**

```
int x = 0;
for (int i = 1; i <= n; i++){
  for (int j = 1; j <= 3*i; j++){
    x = x + j;
  }
}
```

- Number of iterations of inner loop **depends on** the value of **i** in the outer loop
- On **k<sup>th</sup>** iteration of outer loop, how many times through inner loop?
- Total number of iterations of inner loop = sum for **k** running from **1** to **n**
- Time complexity of algorithm?

12-43

## Usefulness of Big-O

- We can **compare algorithms** for efficiency, for example:
  - **Linear search** vs **binary search**
  - Different sort algorithms
  - Iterative vs recursive solutions (recall Fibonacci sequence!)
- We can **estimate actual run times** if we know the time complexity of the algorithm(s) we are analyzing

12-44

## Estimating Run Times

- Assuming a million operations per second on a computer, here are some typical complexity functions and their associated runtimes:

f(n)	n = 10 <sup>3</sup>	n = 10 <sup>5</sup>	n = 10 <sup>6</sup>
log <sub>2</sub> (n)	10 <sup>-5</sup> sec.	1.7*10 <sup>-5</sup> sec.	2*10 <sup>-5</sup> sec.
n	10 <sup>-3</sup> sec.	0.1 sec.	1 sec.
n log <sub>2</sub> (n)	0.01 sec.	1.7 sec.	20 sec.
n <sup>2</sup>	1 sec.	3 hours	12 days
n <sup>3</sup>	17 mins.	32 years	317 centuries
2 <sup>n</sup>	10 <sup>285</sup> cent.	10 <sup>10000</sup> years	10 <sup>100000</sup> years

12-45

## Discussion

- Suppose we want to perform a sort that is **O(n<sup>2</sup>)**. What happens if the number of items to be sorted is 100000?
- Compare this to a sort that is **O(n log<sub>2</sub>(n))**. Now what can we expect?
- Is an **O(n<sup>3</sup>)** algorithm practical for large **n**?
- What about an **O(2<sup>n</sup>)** algorithm, even for small **n**? e.g. for a Pentium, runtimes are:

n = 30	n = 40	n = 50	n = 60
11 sec.	3 hours	130 days	365 years

12-46

## Intractable Problems

- A problem is said to be **intractable** if solving it by computer is impractical
- Algorithms with time complexity **O(2<sup>n</sup>)** take too long to solve even for moderate values of **n**
  - What are some examples we have seen?

12-47