# *Chapter 3*
# *Growth of Functions*

1

## Outlines: Growth of Functions

- Complexity (Worst, Best, and Average Cases)
- Practical vs. Impractical Complexities
- Mathematical Functions' Properties:
  - **Floors & Ceilings**
  - **Polynomials**
  - **Exponents**
  - **Logarithms**
  - **Summation**
  - **Factorials**
- Proofs by:
  - **Cancellation**
  - **Counter example**
  - **Contradiction**
  - **Induction**

2

## How fast will your program run?

- The running time of your program depends on:
  - Algorithm design
  - **Input size**
  - Your implementation of the algorithm in a programming language
  - The compiler you use
  - The OS on your computer
  - Your computer hardware
  - Maybe other things: temperature outside; other programs on your computer; …
- Our Motivation: analyze the running time of an algorithm as a function of only simple parameters of the **input**.

3

## Complexity

- **Complexity** is the number of steps required to solve a problem.
- The goal is to find the best algorithm to solve the problem with a less number of steps
- Complexity of Algorithms
  - The size of the problem is a measure of the quantity of the input data **n**
  - The **time** needed by an algorithm, expressed as a function of the size of the problem (it solves), is called the (**time**) **complexity** of the algorithm **T(n).**

4

## Basic idea: counting operations

- Running Time: Number of primitive steps that are executed
  - most statements roughly require the same amount of time
    - **y = m * x + b**
    - **c = 5 / 9 * (t - 32 )**
    - **z = f(x) + g(y)**
- Each algorithm performs a sequence of basic operations:
  - Arithmetic: (low + high)/2
  - Comparison: if ( x > 0 ) …
  - Assignment: temp = x
  - Branching: while ( true ) { … }
  - …

5

## Basic idea: counting operations

- Idea: count the number of basic operations performed on the input.
- Difficulties:
  - Which operations are basic?
  - Not all operations take the same amount of time.
  - Operations take different times with different hardware or compilers

6

## Measures of Algorithm Complexity

- Let **T(n)** denote the number of operations required by an algorithm to solve a given class of problems
- Often **T(n)** depends on the input, in such cases one can talk about
  - **Worst-case** complexity,
  - **Best-case** complexity,
  - **Average-case** complexity of an algorithm
- Alternatively, one can determine bounds (**upper or lower or tight**) on T(n) .

7

## Measures of Algorithm Complexity

- **Worst-Case Running Time**: the longest time for any input size of **n**
  - provides an upper bound on running time for any input
- **Best-Case Running Time**: the shortest time for any input size of **n**
  - provides lower bound on running time for any input
- **Average-Case Behavior**: the expected performance averaged over all possible inputs
  - it is generally better than worst case behavior, but sometimes it's roughly as bad as worst case
  - difficult to compute

8

## Example: Sequential Search

| Algorithm | Step Count |
|---|---|
| // Searches for x in array A of n items<br>// returns index of found item, or n+1 if not found | |
| **Seq_Search**( A[n]: array, x: item){ | 0 |
| done = false | 1 |
| i = 1 | 1 |
| while ((i <= n) and (A[i] <> x)){ | n + 1 |
| i = i +1 | n |
| } | 0 |
| return i | 1 |
| } | 0 |
| Total | **2n + 4** |

9

## Example: Sequential Search

- worst-case running time
  - when **x** is not in the original array **A**
  - in this case, **while loop** needs **2(n + 1)** comparisons + **c** other operations
  - So, T(n) = **2n + 2 + c** ➔ **Linear complexity**
- best-case running time
  - when **x** is found in **A[1]**
  - in this case, **while loop** needs **2** comparisons + **c** other operations
  - So, T(n) = **2 + c** ➔ **Constant complexity**

10

## Example: Sequential Search

- average-case running time
  - assume **x** is equally likely to equal A[1], A[2], …, A[n]
  - in this case, Pr[x=A[i]] = 1/n, for 1≤ i ≤ n
  - then, the average-case running time is

$$\sum_{i=1}^{n} \Pr[x = A[i]] * i = \sum_{i=1}^{n} \frac{i}{n} = \frac{1}{n} \sum_{i=1}^{n} i$$

  - =(1/n)*n(n+1)/2 = (n+1)/2
  - So, T(n) = (n + 1)/2 ➔ **Linear complexity**

11

## Order of Growth

- For very large input size, it is the rate of grow, or order of growth that matters asymptotically
- We can ignore the lower-order terms, since they are relatively insignificant for very large n
- We can also ignore leading term's constant coefficients, since they are not as important for the rate of growth in computational efficiency for very large n
- Higher order functions of n are normally considered less efficient

12

# Asymptotic Notation

- By now you should have an intuitive feel for asymptotic (big-O) notation:
  - *What does O(n) running time mean? O($n^2$)? O(n lg n)?*
- Our first task is to define this notation more formally and completely

13

# Big-O notation (Upper Bound – Worst Case)

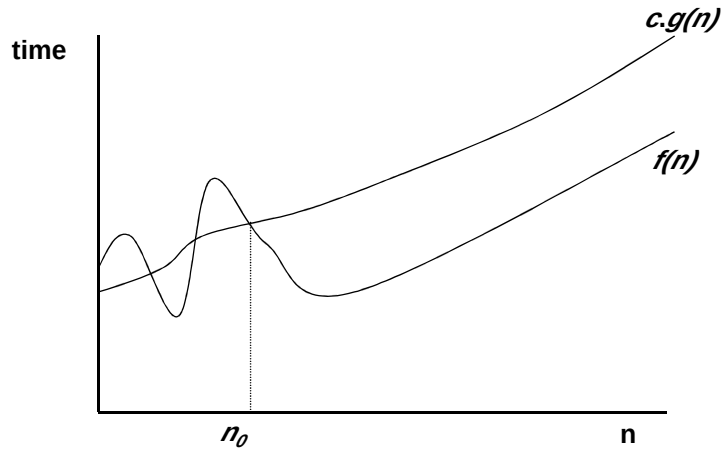
- For a given function *g(n)*, we denote by O(*g*(*n*)) the set of functions
  - O(*g*(*n*)) = {*f(n)*: there exist positive constants *c* >0 and $n_0$ >0 such that 0 ≤ *f(n)* ≤ *cg(n)* for all $n \geq n_0$ }
- We say *g(n)* is an *asymptotic upper bound* for *f(n)*:

$$0 \leq \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

- **O(g(n))** means that as **n → ∞**, the execution time **f(n)** is at **most c.g(n)** for some constant **c**
- *What does O(g(n)) running time mean?*
  - *The worst-case running time (upper-bound) is a function of g(n) to a within a constant factor*

14

## Big-O notation
## (Upper Bound – Worst Case)




*f(n) = O(g(n))*

15

## Big-O notation
## (Upper Bound – Worst Case)

- This is a mathematically formal way of ignoring constant factors, and looking only at the "shape" of the function
- *f(n)=O(g(n))* should be considered as saying that "*f(n)* is at most *g(n)*, up to constant factors".
- We usually will have *f(n)* be the running time of an algorithm and *g(n)* a nicely written function
- E.g. The running time of insertion sort algorithm is $O(n^2)$

16

## Big-O notation
## (Upper Bound – Worst Case)

- **Example1:** Is 2n + 7 = O(n)?
- Let
  - T(n) = 2n + 7
  - T(n) = n (2 + 7/n) ≤ cn

    2 + 7/n ≤ c
  - Note for n=7;
    - **2 + 7/n = 2 + 7/7 = 3**
  - T(n) ≤ **3** n ; ∀ n ≥ **7** ⟵ $n_0$

    c
  - Then T(n) = *O*(*n*)
  - $lim_{n\to\infty}[T(n) / n)] = 2 \geq 0$ ➔ T(n) = *O(n)*

17

## Big-O notation
## (Upper Bound – Worst Case)

- **Example2:** Is $5n^3 + 2n^2 + n + 10^6 = O(n^3)$?
- Let
  - $T(n) = 5n^3 + 2n^2 + n + 10^6$
  - $T(n) = n^3 (5 + 2/n + 1/n^2 + 10^6/n^3) \leq cn^3$

    $(5 + 2/n + 1/n^2 + 10^6/n^3) \leq c$
  - Note for n=100;
    - **$5 + 2/n + 1/n^2 + 10^6/n^3$ =**
    - **5 + 2/100 + 1/10000 + 1 = 6.05**
  - T(n) ≤ **6.05** $n^3$ ; ∀ n ≥ **100** ⟵ $n_0$

    c
  - Then T(n) = $O(n^3)$
  - $lim_{n\to\infty}[T(n) / n^3)] = 5 \geq 0$ ➔ T(n) = $O(n^3)$

18

## Big-O notation
## (Upper Bound – Worst Case)

- Express the execution time as a function of the input size **n**
- Since only the growth rate matters, we can ignore the multiplicative constants and the lower order terms, e.g.,
  - *n, n+1, n+80, 40n, n+log n* *is O(n)*
  - $n^{1.1}$ + ***10000000000n*** is $O(n^{1.1})$
  - ***3***$n^2$ + ***6n + log n + 24.5*** is $O(n^2)$
- $O(1) < O(\log n) < O((\log n)^3) < O(n) < O(n^2) < O(n^3) < O(n^{\log n}) < O(2^{\text{sqrt}(n)}) < O(2^n) < O(n!) < O(n^n)$
- Constant < Logarithmic < Linear < Quadratic< Cubic < Polynomial < Factorial < Exponential

19

## Ω-notation (Omega)
## (Lower Bound – Best Case)

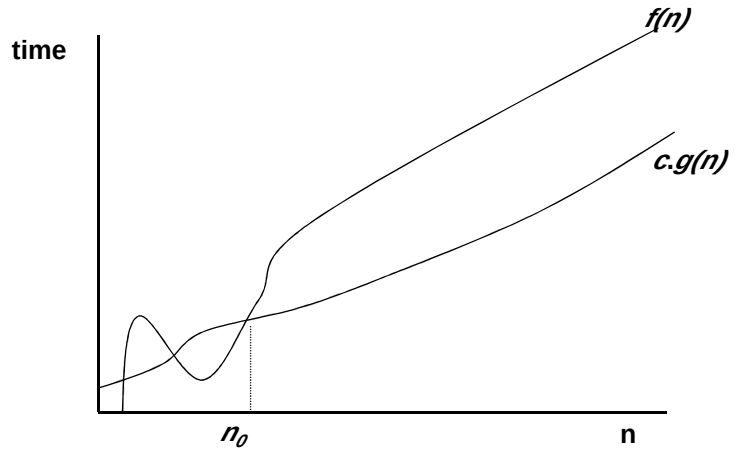
- For a given function *g(n)*, we denote by $\Omega(g(n))$ the set of functions
  - $\Omega(g(n)) = \{f(n)$: there exist positive constants $c > 0$ and $n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$
- We say *g(n)* is an *asymptotic lower bound* for *f(n)*:

$$0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} \leq \infty$$

- **$\Omega$(g(n))** means that as **n $\rightarrow \infty$**, the execution time **f(n)** is at **least c.g(n)** for some constant **c**
- *What does $\Omega(g(n))$ running time mean?*
  - *The best-case running time (lower-bound) is a function of g(n) to a within a constant factor*

20

## Ω-notation
## (Lower Bound – Best Case)




$f(n) = \Omega(g(n))$

21

## Ω-notation (Omega)
## (Lower Bound – Best Case)

- We say Insertion Sort's run time T(n) is Ω(n)
- Proof:
  - Suppose run time is T(n) = a*n* + b
  - Let g(n) = *n*
    - **a.g(n) = a.*n* ≤ T(n) = a*n* + b**
    - ***T(n)* = Ω(g(n)) = Ω(n)**
- For example
  - the worst-case running time of insertion sort is $O(n^2)$, and
  - the best-case running time of insertion sort is Ω(n)
  - Running time falls anywhere between a linear function of **n** and a quadratic function of **n**

22

## Ω-notation (Omega)
## (Lower Bound – Best Case)

- Examples:
  - *n, n+1, n+80, 40n* — *is* $\Omega(n)$
  - $n^{1.1} + 10000000000n$ — is $\Omega(n^{1.1})$
  - $\mathbf{3}n^2 + 6n + \log n + 24.5$ — is $\Omega(n^2)$

23

## Θ notation (Theta)
## (Tight Bound)

- In some cases,
  - $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
  - This means, that the worst and best cases require the same amount of time t within a constant factor
  - In this case we use a new notation called "theta Θ"
- For a given function *g(n)*, we denote by $\Theta(g(n))$ the set of functions
  - $\Theta(g(n)) = \{f(n)$: there exist positive constants $c_1 > 0$, $c_2 > 0$ and $n_0 > 0$ such that $c_1\, g(n) \le f(n) \le c_2\, g(n)\ \forall\ n \ge n_0\}$

24

## Θ notation (Theta)
## (Tight Bound)

❖ We say *g(n)* is an *asymptotic tight bound* for *f(n)*:

$$0 \;<\; \lim_{n \to \infty} \frac{f(n)}{g(n)} \;<\; \infty$$

❖ **Theta notation**

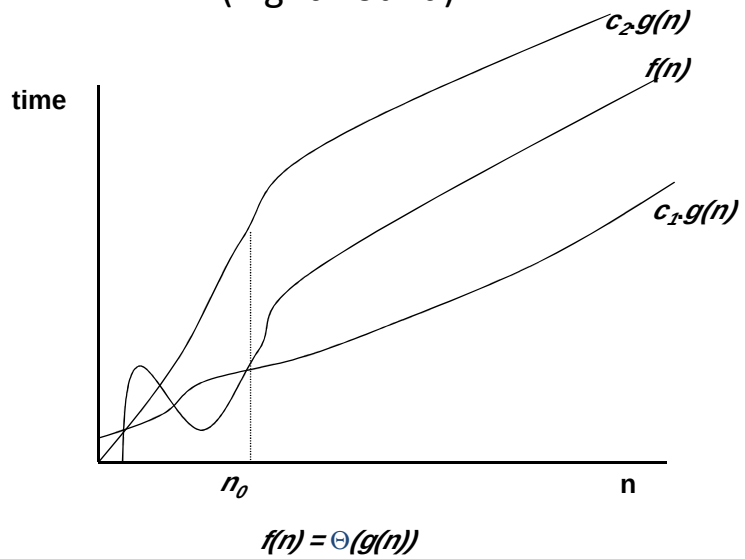
- **θ(g(n))** means that as **n → ∞**, the execution time **f(n)** is at **most $c_2$.g(n)** and at **least $c_1$.g(n)** for some constants **$c_1$** and **$c_2$**.

❖ f(n) = Θ(g(n)) if and only if

- f(n) = O(g(n)) & f(n) = Ω(g(n))

25

## Θ notation (Theta)
## (Tight Bound)




*f(n) = Θ(g(n))*

26

## Θ notation (Theta)
## (Tight Bound)

- **Example1:**
- Show that $6n^3 \neq \Theta(n^2)$
- Suppose for the purpose of contradiction that $c_2$ and $n_0$ exist such that $6n^3 \leq c_2 n^2$ for all $n \geq n_0$
  - Dividing by $n^2$ yields
    - $n \leq c_2/6$
  - which cannot possibly hold for arbitrary large **n**, since $c_2$ is constant
  - Also, $lim_{n\to\infty}[6n^3 / n^2] = lim_{n\to\infty}[6n] = \infty$, *which is not a non-zero constant*

27

## Practical Complexities

- Is $O(n^2)$ too much time?
- Is the algorithm practical?

| *n* | *n* | *nlogn* | $n^2$ | $n^3$ |
|---|---|---|---|---|
| *1000* | 1mic | 10mic | 1milli | 1sec |
| *10000* | 10mic | 130mic | 100milli | 17min |
| $10^6$ | 1milli | 20milli | 17min | 32years |

*At CPU speed $10^9$ instructions/second*

28

## Impractical Complexities

| $n$ | $n^4$ | $n^{10}$ | $2^n$ |
|---|---|---|---|
| 1000 | 17min | $3.2 \times 10^{13}$ years | $3.2 \times 10^{283}$ years |
| 10000 | 116 days | ??? | ??? |
| 10^6 | 3 x 10^7 years | ?????? | ?????? |

*At CPU speed $10^9$ instructions/second*

29

## Some Common Name for Complexity

| | |
|---|---|
| O(1) | Constant time |
| O(log n) | Logarithmic time |
| $O(\log^2 n)$ | Log-squared time |
| O(n) | Linear time |
| $O(n^2)$ | Quadratic time |
| $O(n^3)$ | Cubic time |
| $O(n^i)$ for some i | Polynomial time |
| $O(2^n)$ | Exponential time |

30

## Growth Rates of some Functions

$$O(\log n) < O(\log^2 n) < O(\sqrt{n}) < O(n)$$
$$< O(n\log n) < O(n\log^2 n) < O(n^{1.5}) < O(n^2)$$
$$< O(n^3) < O(n^4)$$

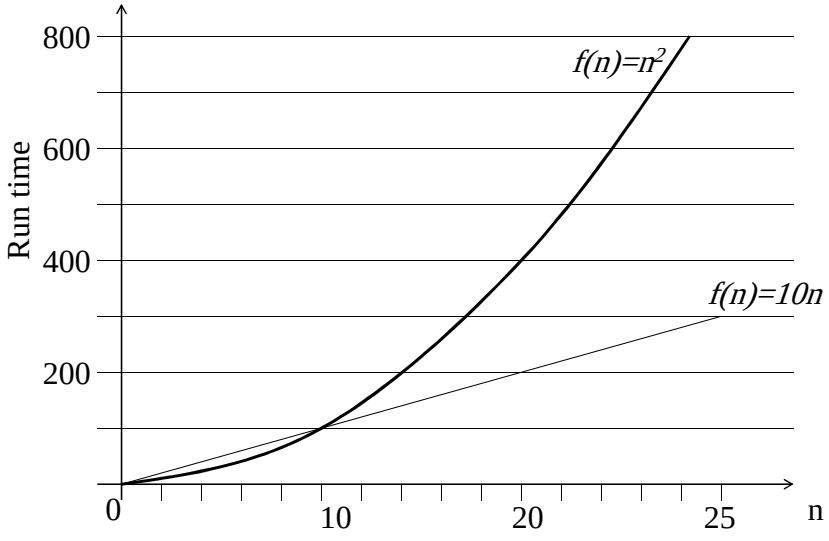
Polynomial Functions

$$O(n^c) = O(2^{c\log n}) \text{ for any constant } c$$
$$< O(n^{\log n}) = O(2^{\log^2 n})$$
$$< O(2^n) < O(3^n) < O(4^n)$$
$$< O(n!) < O(n^n)$$

Exponential Functions

31

## Effect of Multiplicative Constant




32

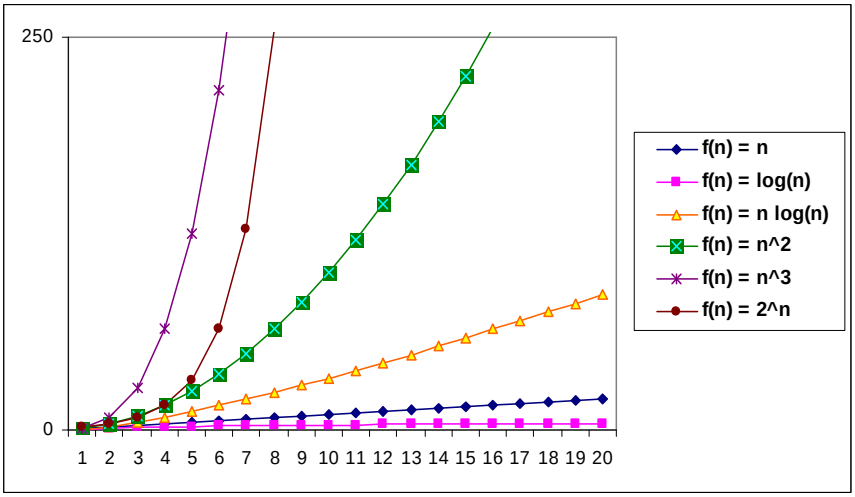
# Exponential Functions

- Exponential functions increase rapidly, e.g., **$2^n$** will double whenever **n** is increased by 1.

| n | $2^n$ | 1µs x $2^n$ |
|---|---|---|
| 10 | $10^3$ | 0.001 s |
| 20 | $10^6$ | 1 s |
| 30 | $10^9$ | 16.7 mins |
| 40 | $10^{12}$ | 11.6 days |
| 50 | $10^{15}$ | 31.7 years |
| 60 | $10^{18}$ | 31710 years |

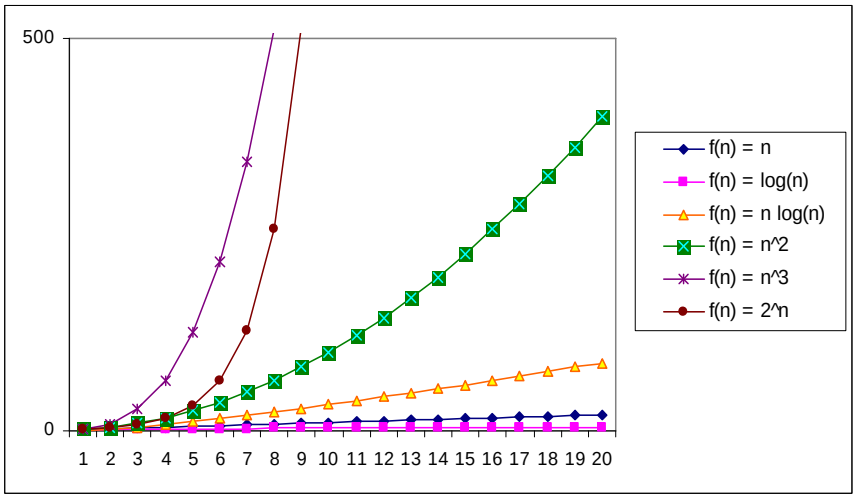
33

# Practical Complexity



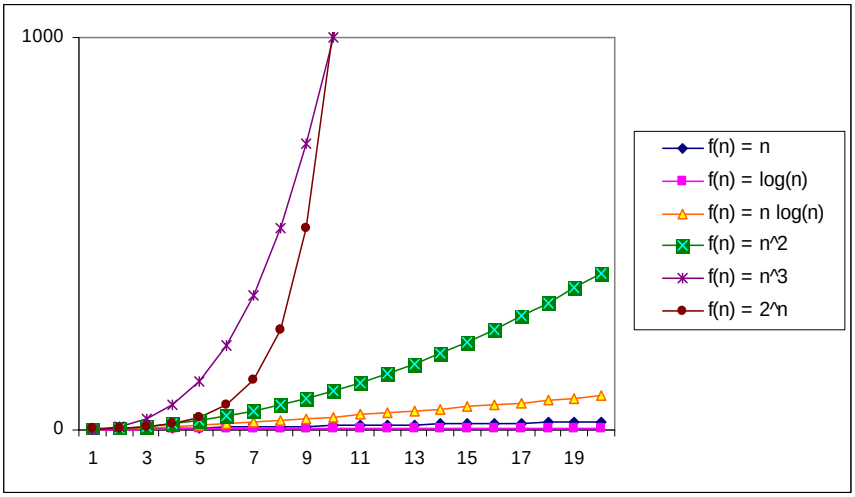

34

# Practical Complexity



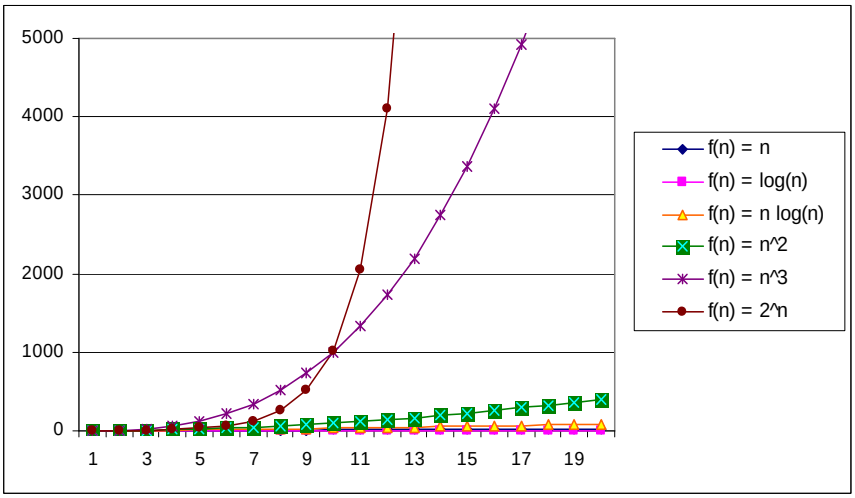

35

# Practical Complexity




36

## Practical Complexity




37

## Floors & Ceilings

- For any **real** number **x**, we denote the **greatest integer** *less than or equal* to **x** by $\lfloor x \rfloor$
  - read "the floor of x"
- For any **real** number **x**, we denote the **least integer** *greater than or equal* to **x** by $\lceil x \rceil$
  - read "the ceiling of x"
- For all real **x**, (example for x=4.2)
  - $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
- For any integer **n** ,
  - $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$

38

# Polynomials

- Given a positive integer ***d***, a **polynomial** in ***n*** of degree ***d*** is a function ***P*(*n*)** of the form
  - ***P*(*n*) =** $\sum_{i=0}^{d} a_i n^i$
  - **where $a_0, a_1, \ldots, a_d$ are coefficient of the polynomial**
  - **$a_i \neq 0$**
- A polynomial is **asymptotically positive** iff $a_d > 0$
  - **Also $P(n) = \Theta(n^d)$**

39

# Exponents

- $x^0 = 1$ $x^1 = x$ $x^{-1} = 1/x$
- $x^a \cdot x^b = x^{a+b}$
- $x^a / x^b = x^{a-b}$
- $(x^a)^b = (x^b)^a = x^{ab}$
- $x^n + x^n = 2x^n \neq x^{2n}$
- $2^n + 2^n = 2.2^n = 2^{n+1}$

40

## Logarithms (1)

- In computer science, all logarithms are to **base 2** unless specified otherwise
- $x^a = b$ iff $\log_x(b) = a$
- $\lg(n) = \log_2(n)$
- $\ln(n) = \log_e(n)$
- $\lg^k(n) = (\lg(n))^k$
- $\log_a(b) = \log_c(b) / \log_c(a)$ ; $c > 0$
- $\lg(ab) = \lg(a) + \lg(b)$
- $\lg(a/b) = \lg(a) - \lg(b)$
- $\lg(a^b) = b \cdot \lg(a)$

41

## Logarithms (2)

- $a = b^{\log_b(a)}$
- $a^{\log_b(n)} = n^{\log_b(a)}$
- $\lg(1/a) = -\lg(a)$
- $\log_b(a) = 1/\log_a(b)$
- $\lg(n) < n$ for all $n > 0$
- $\log_a(a) = 1$
- $\lg(1) = 0$, $\lg(2) = 1$, $\lg(1024=2^{10}) = 10$
- $\lg(1048576=2^{20}) = 20$

42

# Summation

- Why do we need to know this?

  We need it for computing the running time of a given algorithm.

- Example: Maximum Sub-vector

  Given an array a[1…n] of numeric values (can be positive, zero and negative) determine the sub-vector a[i…j] ($1 \le i \le j \le n$) whose sum of elements is maximum over all sub-vectors.

43

# Example: Max Sub-Vectors

```
MaxSubvector(a, n) {
    maxsum = 0;
    for i = 1 to n {
        for j = i to n {
            sum = 0;
            for k = i to j { sum += a[k] }
            maxsum = max(sum, maxsum);
        }
    }
    return maxsum;
}
```

$$T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} 1$$

44

## Summation

$$\sum_{k=1}^{n} k = 1+2+\ldots+n = n(n+1)/2 = \Theta(n^2)$$

$$\sum_{k=0}^{n} x^k = 1+x+x^2\ldots+x^n = \frac{x^{n+1}-1}{x-1}$$

$$\sum_{k=1}^{n} (ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k$$

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0, \quad \text{for } a_0, a_1, \ldots, a_n$$

$$\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n, \quad \text{for } a_0, a_1, \ldots, a_n$$

45

## Summation

- Constant Series: For a, b ≥ 0,

$$\sum_{i=a}^{b} 1 = b - a + 1$$

- Quadratic Series: For n ≥ 0,

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \ldots + n^2 = (2n^3 + 3n^2 + n)/6$$

- Linear-Geometric Series: For n ≥ 0,

$$\sum_{i=1}^{n} ic = c + 2c + \ldots + nc = [(n-1)c^{n+1} - nc^n]/(c-1)^2$$

46

# Series (Summation)

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$$

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$$

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

if $0 < A < 1$:

$$\sum_{i=0}^{N} A^i \leq \frac{1}{1 - A}$$

$$\sum_{i=0}^{\infty} A^i = \frac{1}{1 - A}$$

# Factorials

- **n!** ("n factorial") is defined for integers $n \geq 0$ as
- $n! = \begin{cases} 1 & \text{if } n=0, \\ n(n-1)! & \text{if } n>0 \end{cases}$
- $n! = 1 \cdot 2 \cdot 3 \ldots n$
- $n! < n^n$ for $n \geq 2$

48

# Proofs

- Cancellation
- Counter example
- Contradiction
- Induction

49

# Proof of Geometric Series

A Geometric series is one in which the sum approaches a given number as N tends to infinity.

Proofs for geometric series are done by cancellation, as demonstrated.

*Proof*

$$S = 1 + A + A^2 + A^3 + A^4 + A^5 + \cdots$$
$$AS = A + A^2 + A^3 + A^4 + A^5 + \cdots$$
$$S - AS = 1$$
$$S = \frac{1}{1 - A}$$

## Proofs by Counterexample & Contradiction

- There are several ways to prove a theorem:
  - **Counterexample:**
    - By providing an example of in which the theorem **does not hold**, you prove the theory to be **false**.
    - For example: All multiples of 5 are even.

      However 3x5 is 15, which is odd. The theorem is false.
  - **Contradiction:**
    - Assume the theorem to be **true**. If the assumption **implies** that some known property is **false**, then the theorem **CANNOT** be **true**.

51

## Proof by Induction

- Proof by induction has three standard parts:
  - The first step is proving a **base case**, that is, establishing that a theorem is true for some small (usually degenerate) value(s), this step is almost always trivial.
  - Next, an **inductive hypothesis** is assumed. Generally this means that the theorem is assumed to be true for all cases up to some limit *n*.
  - Using this assumption, the theorem is then shown to be true for the next value, which is typically *n*+1 (**induction step**). This proves the theorem (as long as *n* is finite).

52

## Example: Proof By Induction

- Claim: S(n) is true for all n >= k
  - **Basis:**
    - **Show formula is true when n = k**
  - **Inductive hypothesis:**
    - **Assume formula is true for an arbitrary n**
  - **Induction Step:**
    - **Show that formula is then true for n+1**

53

## Induction Example: Gaussian Closed Form

- Prove 1 + 2 + 3 + … + n = n(n+1) / 2
  - **Basis:**
    - **If n = 0, then 0 = 0(0+1) / 2**
    - **Or if n=1, then 1=1(1+1) / 2**
  - **Inductive hypothesis:**
    - **Assume 1 + 2 + 3 + … + n = n(n+1) / 2**
  - **Step (show true for n+1):**

    **1 + 2 + … + n + n+1 = (1 + 2 + …+ n) + (n+1)**

    **= n(n+1)/2 + n+1 = [n(n+1) + 2(n+1)]/2**

    **= (n+1)(n+2)/2 = (n+1)(n+1 + 1) / 2**

54

# Induction Example: Geometric Closed Form

- Prove
- $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$
  - **Basis: show that $a^0 = (a^{0+1} - 1)/(a - 1)$**

    $a^0 = 1 = (a^1 - 1)/(a - 1) = 1$
  - **Inductive hypothesis:**
    - **Assume $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$**
  - **Step (show true for n+1):**

    $a^0 + a^1 + \ldots + a^{n+1} = a^0 + a^1 + \ldots + a^n + a^{n+1}$

    $= (a^{n+1} - 1)/(a - 1) + a^{n+1} = (a^{n+1+1} - 1)/(a - 1)$

55