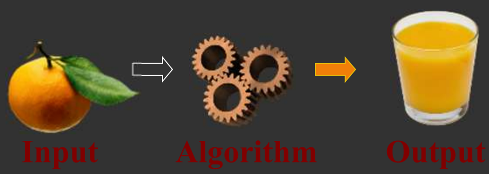
ANALYSIS OF ALGORITHMS



1

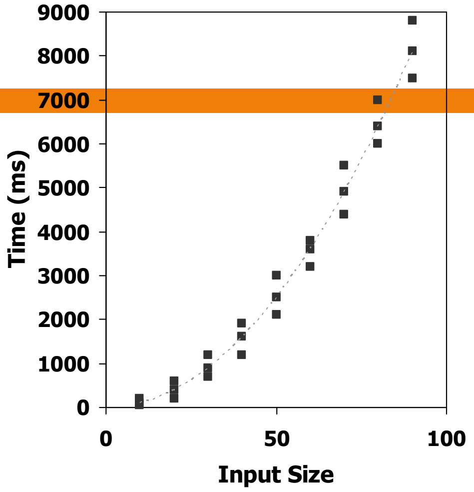
Analysis of Algorithms

- Typically, the primary analysis tool involves characterizing the running times of algorithms and data structure operations, with space usage also being of interest.
- Running time is a natural measure of "goodness," since time is a precious resource - computer solutions should run as fast as possible.

2

Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition, noting the time needed:
- Plot the results




```java
long startTime = System.currentTimeMillis( ); // record the starting time
/* (run the algorithm) */
long endTime = System.currentTimeMillis( ); // record the ending time
long elapsed = endTime - startTime; // compute the elapsed time
```

3

Ex. Experimental Studies

- Two algorithms for constructing long strings in Java.

```java
/** Uses repeated concatenation to compose a String with n copies of character c. */
public static String repeat1(char c, int n) {
    String answer = "";
    for (int j=0; j < n; j++)
    answer += c;
    return answer;
}
/** Uses StringBuilder to compose a String with n copies of character c. */
public static String repeat2(char c, int n) {
    StringBuilder sb = new StringBuilder( );
    for (int j=0; j < n; j++)
    sb.append(c);
    return sb.toString( );
}
```

4

Ex. Experimental Studies

n	Repeat1 (ms)	Repeat 2 (ms)
50.000	2.884	1
100.000	7.437	1
200.000	39.158	2
400.000	170.173	3
800.000	690.836	7
1.600.000	2.874.968	13
3.200.000	12.809.631	28
6.400.000	59.594.275	58
12.800.000	265.696.421	135

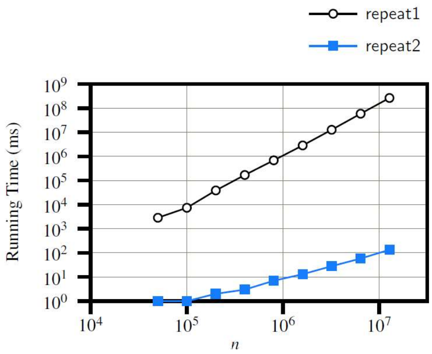


Chart of the results of the timing experiment, displayed on a log-log scale. The divergent slopes demonstrate an order of magnitude difference in the growth of the running times.

HW.

- Implement repeat1 algorithm on Java, Matlab and C# and compare the results for n=1000, 10000 to 10.000.000 various iterations. Write a report about possible reasons of the performance differences.

```
tic
N=100000;
s='';
for i=0:N
    s=s + 'c';
end
a=toc
```

```
using System;
namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
            int N = 10000; string s = "";
            DateTime dt = DateTime.Now;
            for (int i = 0; i < N; i++)
                s = s + "c";
            TimeSpan ts = DateTime.Now - dt;
Console.WriteLine(ts.TotalMilliseconds.ToString());
            Console.ReadKey();
        }
    }
}
```

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

7

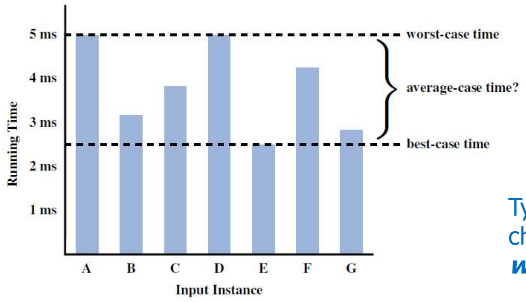
Moving Beyond Experimental Analysis

- Goal is to develop an approach to analyzing the efficiency of algorithms that:
  - Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.
  - Is performed by studying a high-level description of the algorithm without need for implementation.
  - Takes into account all possible inputs.

8

Measuring Operations as a Function of Input Size

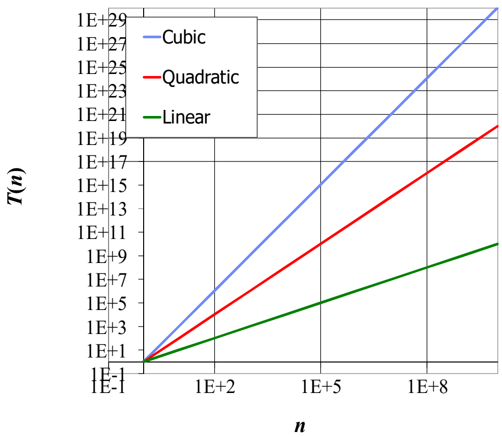
- To capture the order of growth of an algorithm's running time, we will associate, with each algorithm, a function *f (n) that characterizes the number of primitive* operations that are performed as a function of the input size *n.*



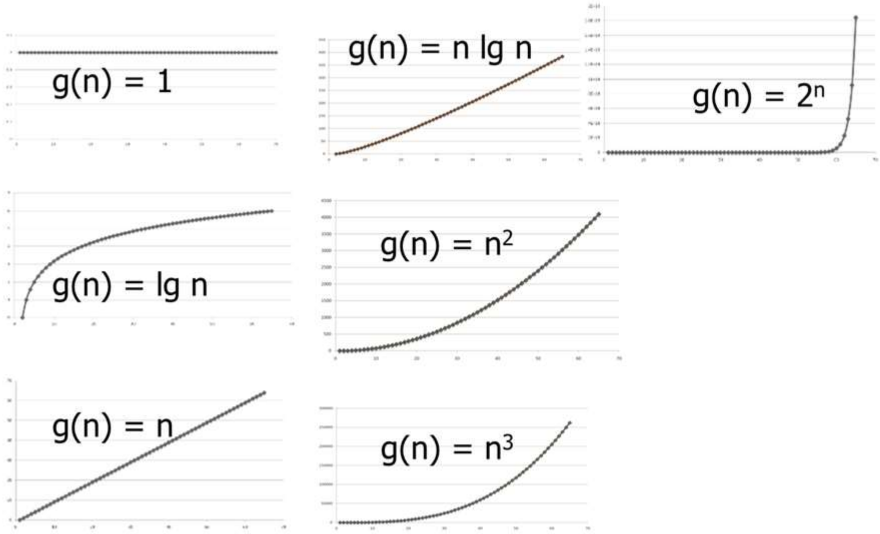
Typically, running times are characterized in terms of the ***worst case !!!!***

Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ ***1***
  - Logarithmic ≈ log ***n***
  - Linear ≈ ***n***
  - N-Log-N ≈ ***n*** log ***n***
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n
- In a log-log chart, the slope of the line corresponds to the growth rate

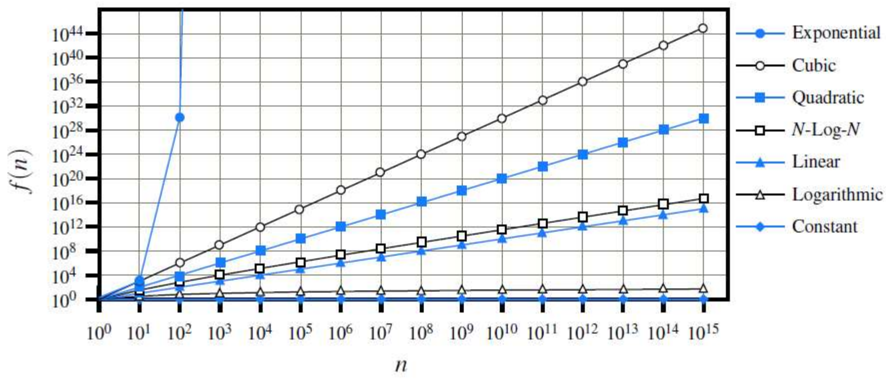


Functions Graphed Using “Normal” Scale



Comparing Growth Rates

constant	logarithm	linear	*n*-log-*n*	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n



Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Algorithm 1 Intent Communication Algorithm

procedure DEC-MDP(S, A, P, R, O, Ω)
  $A \leftarrow A_1 \times A_2$
  $s_1, s_2 \leftarrow S$
  $a_1, a_2 \leftarrow A$
  $R(s_i, a_i) = 0, i = 0, j = 0$
  repeat
    $i \leftarrow i + 1, j \leftarrow j + 1$
    for o_1, o_2 **do**
      Determine scenario $\in [1, 4]$
      $p_1, p_2 \leftarrow P(s' \mid s, a_1, a_2)$
      $a_1, a_2 \leftarrow A$
      $\max_{a_1, a_2} r_{1,2}(s_1, s_2, a_1, a_2)$
      for s_1, s_2 **do** check
        if $d(s_1, s_2) \leq$ *scenario threshold* **then**
          Update θ_i, θ_j using $d(s_1, s_2)$
        end if
        $\pi[s_1, s_2] = \arg\max_{a_1, a_2} r_{1,2}$
      end for
    end for
  until $s_1 = s_{g_1}$ or $s_2 = s_{g_2}$
  return $\pi, R(s_i, a_i)$
end procedure

Pseudocode Details

- Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***…])
    Input …
    Output …

Type of operation	Symbol	Example
Assignment	$\leftarrow$ or :=	$c \leftarrow 2\pi r$, $c := 2\pi r$
Comparison	$=, \neq, <, >, \leq, \geq$	
Arithmetic	$+, -, \times, /$, mod	
Floor/ceiling	$\lfloor, \rfloor, \lceil, \rceil$	$a \leftarrow \lfloor b \rfloor + \lceil c \rceil$
Logical	**and**, **or**	
Sums, products	$\sum \prod$	$h \leftarrow \sum_{a \in A} 1/a$

- Method call
  - ***method*** (***arg*** [, ***arg***…])
- Return value
  - **return** *expression*
- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …

Pseudocode Examples

```
neutral_vars ← Ø //Begin Generation
covered ← ∪_{t∈T} statements visited by P(t)
repeat
  variant ← single_mutation(P, covered)
  if is_neutral(var, T) then
    neutral_vars ← neutral_vars ∪ {variant}
  x ← x − 1
until x ≤ 0
clusters ← Ø //Begin Composition
y' ← y
while |clusters| < N do
  candidate ← choose_from(neutral_vars, k)
  if is_neutral(candidate, T) then
    clusters ← clusters ∪ {candidate}
    y' ← y
  else
    y' ← y' − 1
    if y' ≤ 0 then
      k ← ⌊k/2⌋
      if k ≤ 1 then
        return clusters
      y' ← y
return clusters
```

```
initialize p_0 agents, each with energy E = θ/2
loop:
    foreach alive agent a:
        pick link from current document
        fetch new document D
        E_a ← E_a − c(D) + e(D)
        Q-learn with reinforcement signal e(D)
        if (E_a ≥ θ)
            a' ← mutate(recombine(clone(a)))
            E_a, E_a' ← E_a/2
        elsif (E_a ≤ 0)
            die(a)
    process optional relevance feedback from user
```

15

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

- Examples:
  - Performing an arithmetic operation
  - Following an object reference
  - Assigning a value to a variable
  - Accessing a single element of an array by index
  - Calling a method
  - Returning from a method
  - Comparing two numbers

16

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

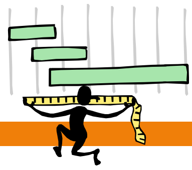
```
/** Returns the maximum value of a nonempty array of numbers. */
public static double arrayMax(double[] data) {
3. int n = data.length;
4. double currentMax = data[0]; // assume first entry is biggest (for now)
5. for (int j=1; j < n; j++) // consider all other entries
6. if (data[j] > currentMax) // if data[j] is biggest thus far...
7.        currentMax = data[j]; // record it as the current max

8. return currentMax;
}
```

Step 3: 2 ops, 4: 2 ops, 5: 2n ops + 1, 6: 2n ops, 7: 0 to n ops, 8: 1 op

17

Estimating Running Time



- Algorithm arrayMax executes $5n + 6$ primitive operations in the worst case, $4n + 6$ in the best case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of arrayMax. Then

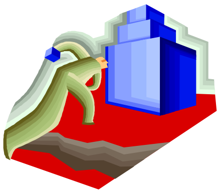
$$a\,(4n + 6) \le T(n) \le b(5n + 6)$$

- Hence, the running time $T(n)$ is bounded by two linear functions

18

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm arrayMax



19

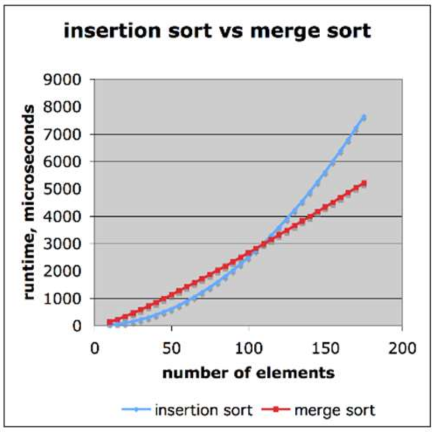
Why Growth Rate Matters

if runtime is...	time for n + 1	time for 2 n	time for 4 n
c lg n	c lg (n + 1)	c ((lg n) + 1)	c((lg n) + 2)
c n	c (n + 1)	2c n	4c n
c n lg n	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
c n^2	~ c n^2 + 2c n	4c n^2	16c n^2
c n^3	~ c n^3 + 3c n^2	8c n^3	64c n^3
c 2^n	c 2^{n+1}	c 2^{2n}	c 2^{4n}

runtime quadruples when problem size doubles

20

Comparison of Two Algorithms



insertion sort is
$n^2 / 4$

merge sort is
$2 n \lg n$

sort a million items?

insertion sort takes roughly 70 hours

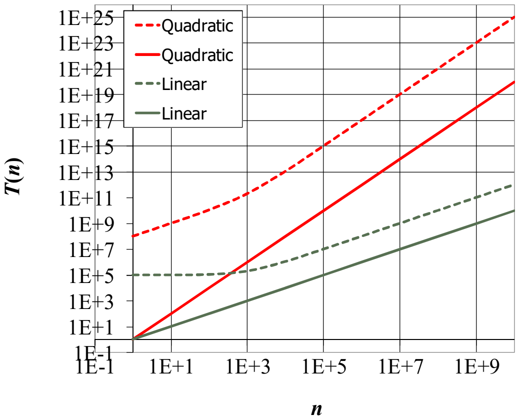
while

merge sort takes roughly 40 seconds

This is a slow machine, but if 100 x as fast then it's 40 minutes versus less than 0.5 seconds

Constant Factors

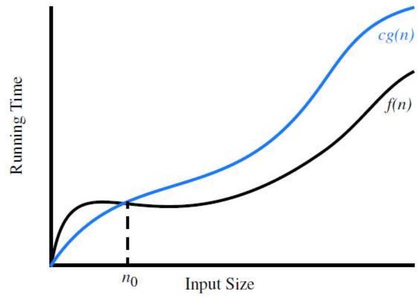
- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function



Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

  $f(n) \le cg(n)$ for $n \ge n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \le cn$
  - $(c - 2)\, n \ge 10$
  - $n \ge 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$

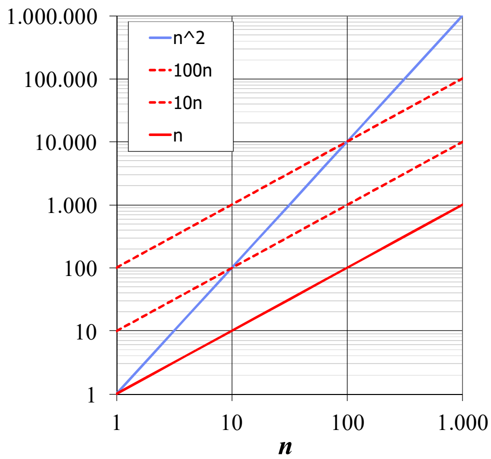


Illustrating the "big-Oh" notation. The function *f (n) is O(g(n)), since f (n) ≤ c ·g(n) when n ≥ n0.*

23

Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \le cn$
  - $n \le c$
  - The above inequality cannot be satisfied since c must be a constant



24

More Big-Oh Examples

- $7n - 2$

  $7n-2$ is $O(n)$

  need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq cn$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

- $3 \log n + 5$

  $3 \log n + 5$ is $O(\log n)$

  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \log n$ for $n \geq n_0$

  this is true for $c = 8$ and $n_0 = 2$

25

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

26

Big-Oh Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

27

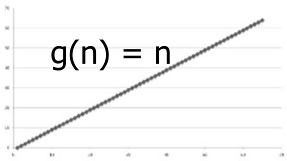
Ex:

- What is the complexity/growth rate of the following java function?

```java
public static void printAll(double[ ] x) {
    int n = x.length;
    for (int j=0; j < n; j++) {
        System.out.print(x[j]);
    }

}
```

$O(n)$



28

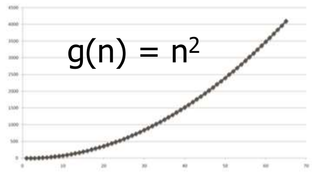
Ex:

- What is the complexity/growth rate of the following java function?

```
public static void printAll(double[ ] x) {
    int n = x.length;
    for (int j=0; j < n; j++) {
     for (int k=0; k < n; k++) {

        System.out.print(x[j] + x[k]);
      }
    }
}
```

$O(n^2)$



29

Relatives of Big-Oh

Big-Omega Ω

- f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c\, g(n)$ for $n \geq n_0$

Big-Theta Θ

- f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c'g(n) \leq f(n) \leq c''g(n)$ for $n \geq n_0$

30

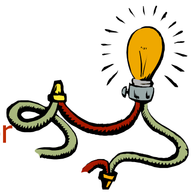
Intuition for Asymptotic Notation

big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically less than or equal to g(n)

big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically greater than or equal to g(n)



big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically equal to g(n)

31

Example Uses of the Relatives of Big-Oh

- **$5n^2$ is $\Omega(n^2)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c\, g(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

- **$5n^2$ is $\Omega(n)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c\, g(n)$ for $n \geq n_0$

  let $c = 1$ and $n_0 = 1$

- **$5n^2$ is $\Theta(n^2)$**

  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c\, g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

32

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We say that algorithm arrayMax "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

33

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

- Computing the array A of prefix averages of another array X has applications to financial analysis

34

Prefix Averages (Quadratic)

The following algorithm computes prefix averages in quadratic time by applying the definition

```
/** Returns an array a such that, for all j, a[j] equals the average of x[0],
..., x[j]. */
public static double[ ] prefixAverage1(double[ ] x) {
    int n = x.length;
    double[ ] a = new double[n]; // filled with zeros by default
    for (int j=0; j < n; j++) {
        double total = 0; // begin computing x[0] + ... + x[j]
        for (int i=0; i <= j; i++)
            total += x[i];
        a[j] = total / (j+1); // record the average
    }
    return a;
}
```

35

Arithmetic Progression

- The running time of prefixAverage1 is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm prefixAverage1 runs in $O(n^2)$ time

36

Prefix Averages 2 (Linear)

The following algorithm uses a running summation to improve the efficiency

```
/** Returns an array a such that, for all j, a[j] equals the average of
x[0], ..., x[j]. */
public static double[ ] prefixAverage2(double[] x) {
int n = x.length;
double[ ] a = new double[n]; // filled with zeros by default
double total = 0; // compute prefix sum as x[0] + x[1] + ...
for (int j=0; j < n; j++) {
    total += x[j]; // update prefix sum to include x[j]
    a[j] = total / (j+1); // compute average based on current sum
}
return a;
}
```

Algorithm prefixAverage2 runs in $O(n)$ time!

Math you need to Review

- Summations
- Powers
- Logarithms
- Proof techniques
- Basic probability

- Properties of powers:

  $a^{(b+c)} = a^b a^c$

  $a^{bc} = (a^b)^c$

  $a^b / a^c = a^{(b-c)}$

  $b = a^{\log_a b}$

  $b^c = a^{c*\log_a b}$

- Properties of logarithms:

  $\log_b(xy) = \log_b x + \log_b y$

  $\log_b (x/y) = \log_b x - \log_b y$

  $\log_b xa = a\log_b x$

  $\log_b a = \log_x a/\log_x b$

Justification Techniques (By Example)

- Some claims are of the **generic form**, "There is an element x *in a set S that has* property *P*." *To justify such a claim, we only need to produce a particular x in S* that has property *P. Likewise, some hard-to-believe claims are of the generic form,* "Every element x *in a set S has property P.*" *To justify that such a claim is false, we* only need to produce a particular x ***from S that does not have property P***. *Such an* instance is called a ***counterexample*****.**
- Example: Professor Among us claims that every number of the form $2^i - 1$ is a prime, when i *is an integer greater than 1. Professor Amongus is wrong.*
- Justification: To prove Professor Amongus is wrong, we find a counterexample. Fortunately, we need not look too far, for $2^4 - 1 = 15 = 3 \cdot 5$.

39

The "Contra" Attack

- Another set of justification techniques involves the use of the negative. The two primary such methods are the use of the ***contrapositive and the contradiction.*** *To* justify the statement "if *p is true, then q is true,*" *we establish that "if q is not true,* then *p is not true" instead. Logically, these two statements are the same, but the* latter, which is called the ***contrapositive of the first, may be easier to think about.***
- Example 4.18: Let *a and b be integers. If ab is even, then a is even or b is even.*
- Justification: To justify this claim, consider the contrapositive, "If *a is odd and b is odd, then ab is odd.*" *So, suppose* $a = 2j+1$ *and* $b = 2k+1$, *for some integers j and k. Then* $ab = 4jk+2j+2k+1 = 2(2jk+j+k)+1$; *hence, ab is odd.*

40

Contradiction

- **Justification by contradiction technique**, we establish that a statement *q is true by first supposing that q is false and then showing that this assumption leads to a contradiction (such as* $2 \neq 2$ or $1 > 3$). By reaching such a contradiction, we show that no consistent situation exists with *q being false, so q must be true. Of course, in order to reach this* conclusion, we must be sure our situation is consistent before we assume *q is false*.
- **Example:** Let *a and b be integers. If ab is odd, then a is odd and b is odd.*
- **Justification:** Let *ab be odd.We wish to show that a is odd and b is odd. So,* with the hope of leading to a contradiction, let us assume the opposite, namely, suppose *a is even or b is even. In fact, without loss of generality, we can assume* that *a is even (since the case for b is symmetric).Then* $a = 2j$ *for some integer j. Hence,* $ab = (2j)b = 2(jb)$*, that is, ab is even. But this is a contradiction:* ***ab* cannot simultaneously be odd and even**. Therefore, *a is odd and b is odd*.

Induction and Loop Invariants

- Most of the claims we make about a running time or a space bound involve an integer parameter *n (usually denoting an intuitive notion of the "size" of the problem)*. Moreover, most of these claims are equivalent to saying some statement *q(n) is true* "for all $n \geq 1$."*Since this is making a claim about an infinite set of numbers, we* cannot justify this exhaustively in a direct fashion.

Induction

- We can often justify claims such as those above as true, however, by using the technique of ***induction. This technique amounts to showing that, for any particular*** *$n \geq 1$, there is a finite sequence of implications that starts with something known* to be true and ultimately leads to showing that *$q(n)$ is true. Specifically, we begin a* justification by induction by showing that *$q(n)$ is true for $n = 1$ (and possibly some* other values *$n = 2, 3, \ldots, k$, for some constant k). Then we justify that the inductive* "step" is true for *$n > k$, namely, we show "if $q(j)$ is true for all $j < n$, then $q(n)$ is* true." The combination of these two pieces completes the justification by induction.

43

Induction

- Proposition 4.20: Consider the Fibonacci function *$F(n)$, which is defined such* that *$F(1) = 1$, $F(2) = 2$, and $F(n) = F(n-2)+F(n-1)$ for $n > 2$. (See Section* 2.2.3.) We claim that $\boldsymbol{F(n)} < 2^n$.
- Justification: We will show our claim is correct by induction.
- ***Base cases: ($n \leq 2$). $F(1) = 1 < 2 = 2^1$ and $F(2) = 2 < 4 = 2^2$.***
- ***Induction step: ($n > 2$). Suppose our claim is true for all $j<n$. Since both $n-2$*** and *$n-1$ are less than n, we can apply the inductive assumption (sometimes called* the "inductive hypothesis") to imply that
- $\boldsymbol{F(n) = F(n-2)+F(n-1)} < 2^{n-2}+2^{n-1}$.
- Since
- $2^{n-2} + 2^{n-1} < 2^{n-1} + 2^{n-1} = 2 \cdot 2^{n-1} = 2n$,
- we have that $\boldsymbol{F(n)} < 2^n$, *thus showing the inductive hypothesis for n.*

44

Loop Invariants

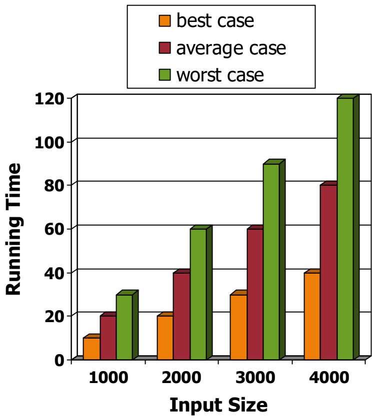
- The final justification technique we discuss in this section is the ***loop invariant.*** *To* prove some statement L about a loop is correct, define L in terms of a series of smaller statements $L_0, L_1, \ldots, L_k$, *where:*
- 1. The ***initial claim, L_0, is true before the loop begins.***
- 2. If L_{j-1} *is true before iteration j, then* L_j *will be true after iteration j.*
- 3. The final statement, L_k, *implies the desired statement L to be true.*
- Let us give a simple example of using a loop-invariant argument to justify the correctness of an algorithm. In particular, we use a loop invariant to justify that the method arrayFind (see Code Fragment 4.11) finds the smallest index at which element val occurs in array *A*.


```
/** Returns index j such that data[j] == val, or −1 if no such element. */
public static int arrayFind(int[ ] data, int val) {
    int n = data.length;
    int j = 0;
    while (j < n) { // val is not equal to any of the first j elements of data
        if (data[j] == val)
            return j; // a match was found at index j
        j++; // continue to next index
        // val is not equal to any of the first j elements of data
    }
    return −1; // if we reach this, no match found
}
```

Code Fragment 4.11: Algorithm arrayFind for finding the first index at which a given element occurs in an array.
To show that arrayFind is correct, we inductively define a series of statements, L*j, that lead to the correctness of our algorithm.*
Specifically, we claim the following is true at the beginning of iteration *j of the **while loop:***
L*j*: *val is not equal to any of the first j elements of data.* This claim is true at the beginning of the first iteration of the loop, because *j is* 0 and there are no elements among the first 0 in data (this kind of a trivially true claim is said to hold ***vacuously). In iteration j, we compare element val to element* data[*j*]**; *if these two elements are equivalent, we return the index j, which is clearly* correct since no earlier elements equal *val. If the two elements val and data[j] are* not equal, then we have found one more element not equal to *val and we increment* the index *j. Thus, the claim Lj will be true for this new value of j; hence, it is* true at the beginning of the next iteration. If the while loop terminates without ever returning an index in data, then we have *j = n. That is, Ln is true—there are no* elements of data equal to *val. Therefore, the algorithm correctly returns −1 to* indicate that *val is not in data.*

Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics

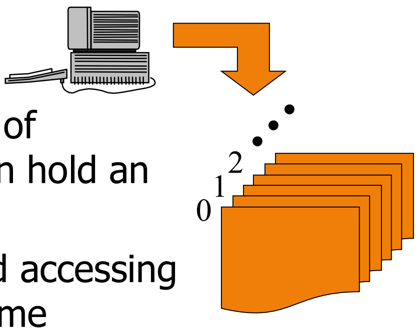


47

The Random Access Machine (RAM) Model

A RAM consists of

- A CPU
- An potentially unbounded bank of memory cells, each of which can hold an arbitrary number or character
- Memory cells are numbered and accessing any cell in memory takes unit time



48