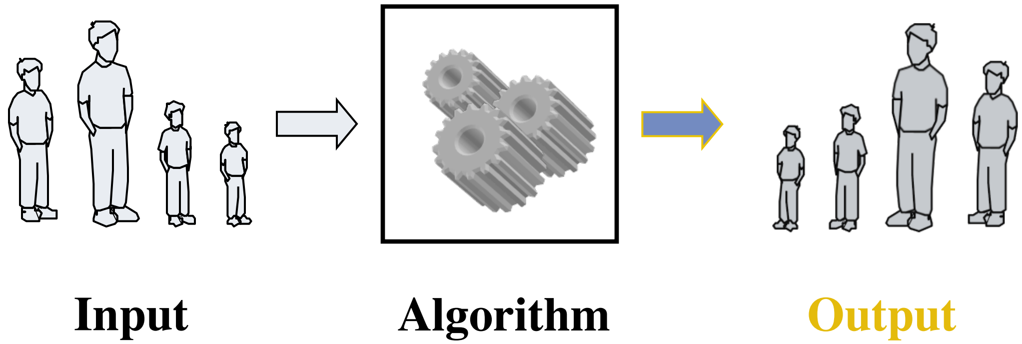
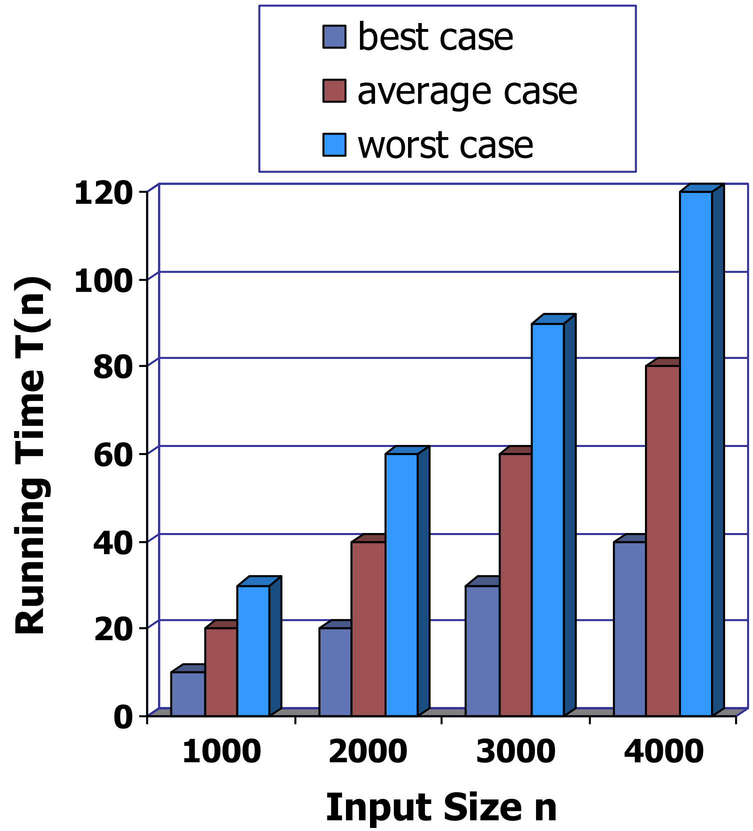
Analysis of Algorithms



An **algorithm** is a step-by-step procedure of unambiguous instructions for solving a problem in a finite amount of time.

Algorithm Running Time

- Most algorithms transform input (data) into output (data).
- The **running time** of an algorithm typically grows with the input size.
- **Average case** time is often difficult to determine.
  - Need extra information about the input (easy/hard to process etc.)
- We focus on the **worst case** running time.
  - Easier to analyze
  - Pays to be pessimistic
  - Identify any bottlenecks

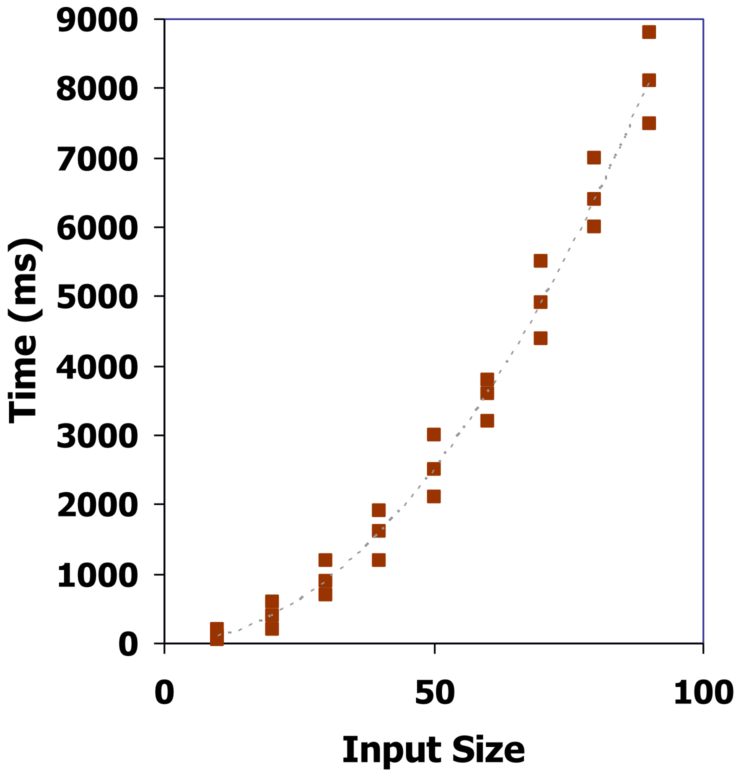


Running Time - Dependencies

- The running time (clock) of an algorithm depends on…
  - Computer – hardware (memory, CPU), bandwidth
  - Programmer skill – design and implementation
  - Compiler – optimization
  - Input values - varying composition and size
  - Algorithm -
    - Algorithms, just as humans, have strengths and weaknesses.
    - May perform well on a particular input, but equally poor on a different input.

Experimental Studies

- Write a program implementing the algorithm.
- Run the program with inputs of varying *size* and *composition*.
- Measure the runtime using *clock-time, cpu-time*
- Plot the results…



Limitations of Experimental Studies

- It is necessary to implement the algorithm first, which may in fact end up being a poor performing algorithm.
- Difficult to obtain a *good (large and varied)* range of inputs?
  - Real world data – costly to acquire & time consuming
  - Synthetically created data – makes claims suspect
- In order to compare two algorithms, the **same** hardware and software environments must be used.
- Difficult to be exhaustive, or use enough sample inputs to be able to make reliable claims about the algorithm.
  - There can be some input that completely brings an algorithm to its knees that never gets tested.

Theoretical Analysis

- Uses a high-level description (pseudocode) of the algorithm instead of an actual implementation.
- Characterizes running time as a function of the input size, n, e.g., $T(n)$ *or* $f(n)$.
  - We care about very large input sizes, Large n
- Takes into account all possible inputs.
- Evaluates algorithm independent of hardware, implementation, input set, etc.
- Metric – we count number of operations not actual clock time

Limitations of Theoretical Analysis

- Donald Knuth quotes
  - *"If you optimize everything, you will always be unhappy."*
  - *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil"* – often attributed to Tony Hoare
- Usually costly to hire a PhD in Theoretical Computer Science to analyze/design algorithms.
- Math….. ☺ or ☹
- Companies such as Google of Microsoft might run diagnostic tests on software, identify the bottle-necks and then hire CS theoreticians to design better algorithms to address these.

Theoretical Analysis - Benefits

- Determine how efficient an algorithm is, across all machines, programming languages, etc.
- Allows us to compare different algorithms for the same problem, e.g., *sorting*.
- Allows us to identify problems which cannot be solved with a computer.
- Helps us identify sections of algorithm with high cost where we:
  - can improve
  - cannot improve, i.e., ***lower-bound***
- Asymptotic Analysis – Compare running time as a function of the input size in the limit, *i.e.*, as n approaches infinity.

Pseudocode

- high-level description of an algorithm
- more structured and less ambiguous than English
- less detailed than a programming language
- preferred notation for describing algorithms
- hides implementation details

Example: find max element of an array

Algorithm *arrayMax*(A, n)
  Input array A of n integers
  Output maximum element of A

  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $A[i] > currentMax$ **then**
      $currentMax \leftarrow A[i]$
  return $currentMax$

Pseudocode v. C++ : side-by-side comparison

Algorithm ***arrayMax***(A,***n***)

Input: *An array A storing n ≥ 1 integers.*

Output: *The maximum element of A.*

currentMax ← *A[0]*

for i ← n **to** n – 1 **do**

  if ***currentMax*** < *A[i]* **then**

    currentMax ← *A[i]*

return ***currentMax***

```
int arrayMax(int A[], int n){

int currentMax = A[0];
   for(int i = 1; i < n; i++){
        if(currentMax < A[i])
           currentMax = A[i];
    }
    return currentMax;
}
```

Analysis - Counting Primitive Operations

- Basic computations performed by an algorithm
  - Assigning a value to a variable, x = 5, x = y
  - Function call, max(5,7)
  - Performing an arithmetic operation, e.g., 5+7
  - Comparison, e.g., x < 5
  - Indexing into an array, a[5]
  - Evaluating an expression (4+n)*5
  - Returning from a function, return
- Above are Primitive Operations ("Atomic" in book)
  - a low level instruction whose execution time depends on environment's hardware and software .
  - for analysis purposes, constant time instruction, O(1) – "Big-Oh of 1" or "constant time"

Counting Primitive Operations

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

Algorithm ***printArray***(A, n)

$i \leftarrow 0$	1 assignment
while $i < n$ **do**	n + 1 comparisons
cout << A[i] << endl	n outputs
i++	n increments

1 + (n+1) + n + n = $3n$ + 2 operations
Proportional to n, 3 times n + constant

Counting Primitive Operations

- (Stop here) Quick exercise -

Algorithm $foo(n)$

$x \leftarrow 0$, $y \leftarrow 0$
while $x < n$ **do**
    x++
    while $y < n$ **do**
        y++
    $y \leftarrow 0$

Remember that…

$$printArray = 3n + 2$$

Algorithm $printArray(A, n)$

$i \leftarrow 0$ — 1 assignment

while $i < \mathrm{n}$ **do** — n + 1 comparisons

  cout << A[i] << endl — n outputs

  i ++ — n increments

1 + (n+1) + n + n = $3n$ + 2 operations
Proportional to n, 3 times n + constant

Estimating Running Time

- Algorithm ***printArray*** executes $3\boldsymbol{n} + 2$ primitive operations.
- If we *define*:

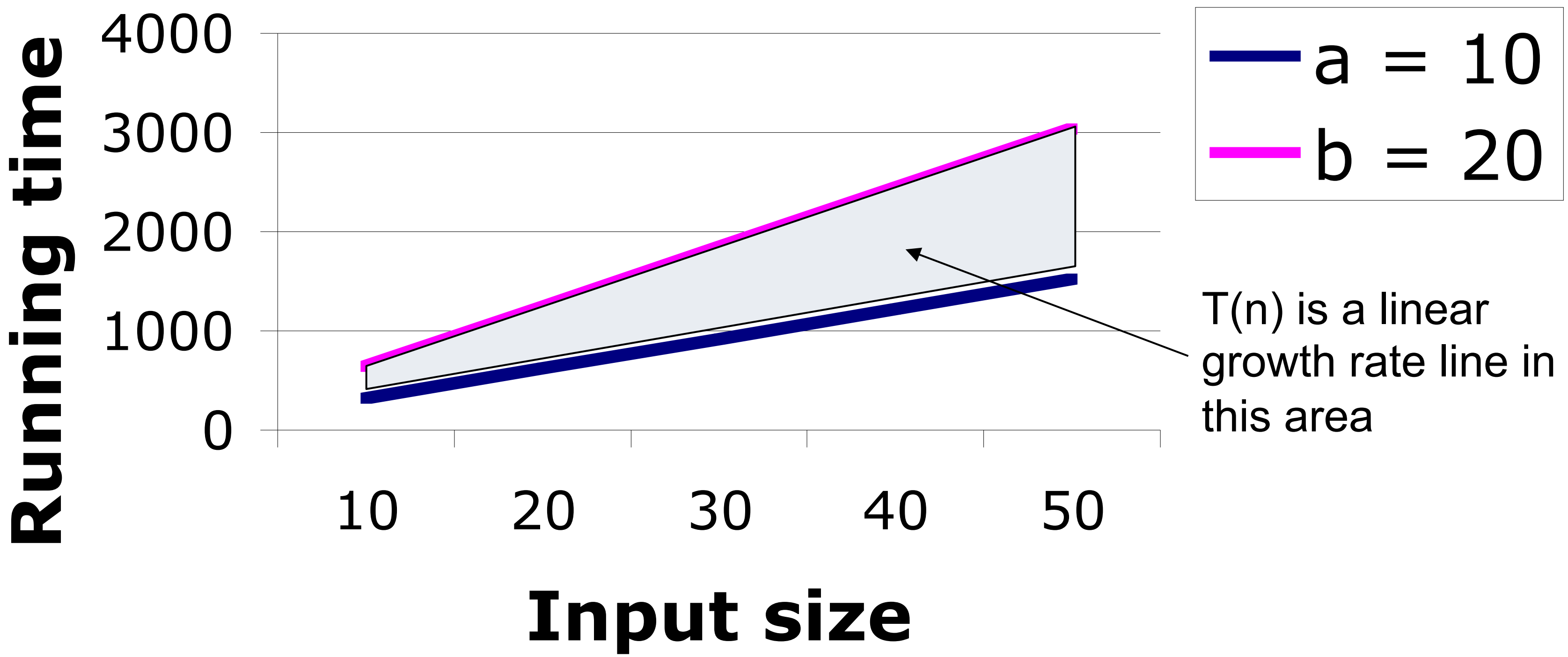
  $\boldsymbol{a}$ = Time taken by the fastest primitive operation

  $\boldsymbol{b}$ = Time taken by the slowest primitive operation
- Let $\boldsymbol{T}(\boldsymbol{n})$ be worst-case time of ***printArray.*** Then

$$\boldsymbol{a}\,(3\boldsymbol{n} + 2) \le \boldsymbol{T}(\boldsymbol{n}) \le \boldsymbol{b}(3\boldsymbol{n} + 2)$$

- Hence, the running time $\boldsymbol{T}(\boldsymbol{n})$ is bounded by two linear functions.

Growth Rate of Running Time



Growth Rates