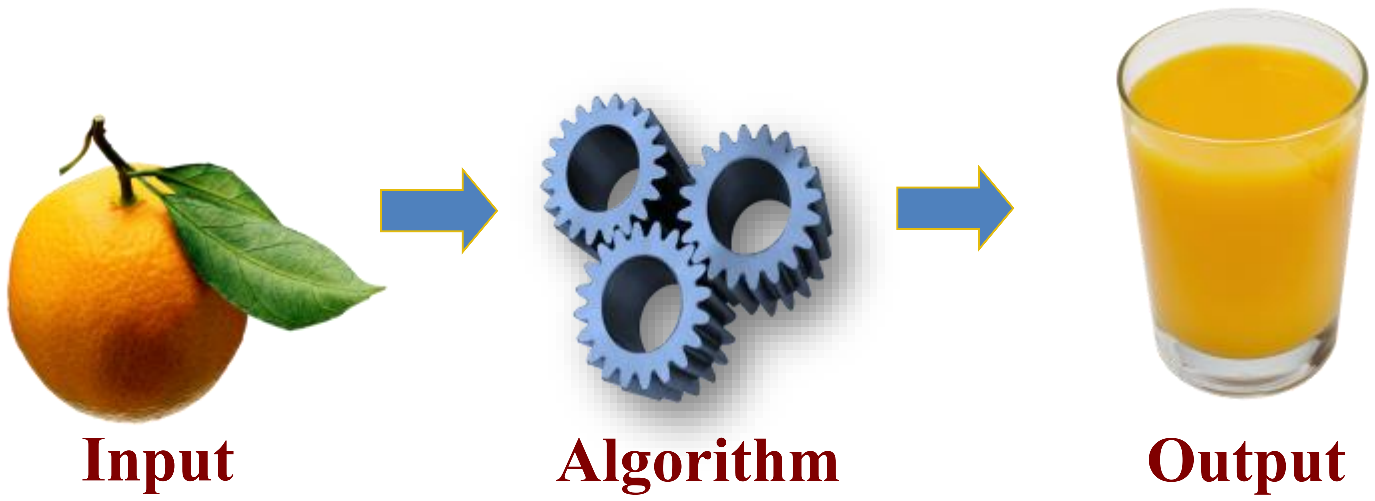
Analysis of Algorithms



Instructor: Andy Mirzaian

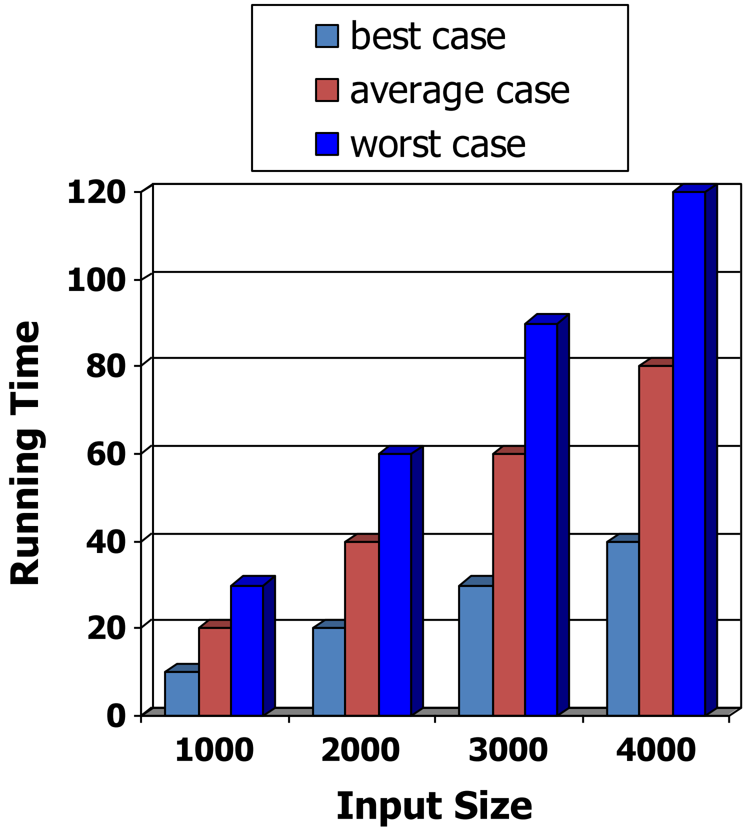
Edsger Wybe Dijkstra [Turing Award Winner, 1972]

"Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. The only effective way to raise confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand."

Part 1:
Running Time

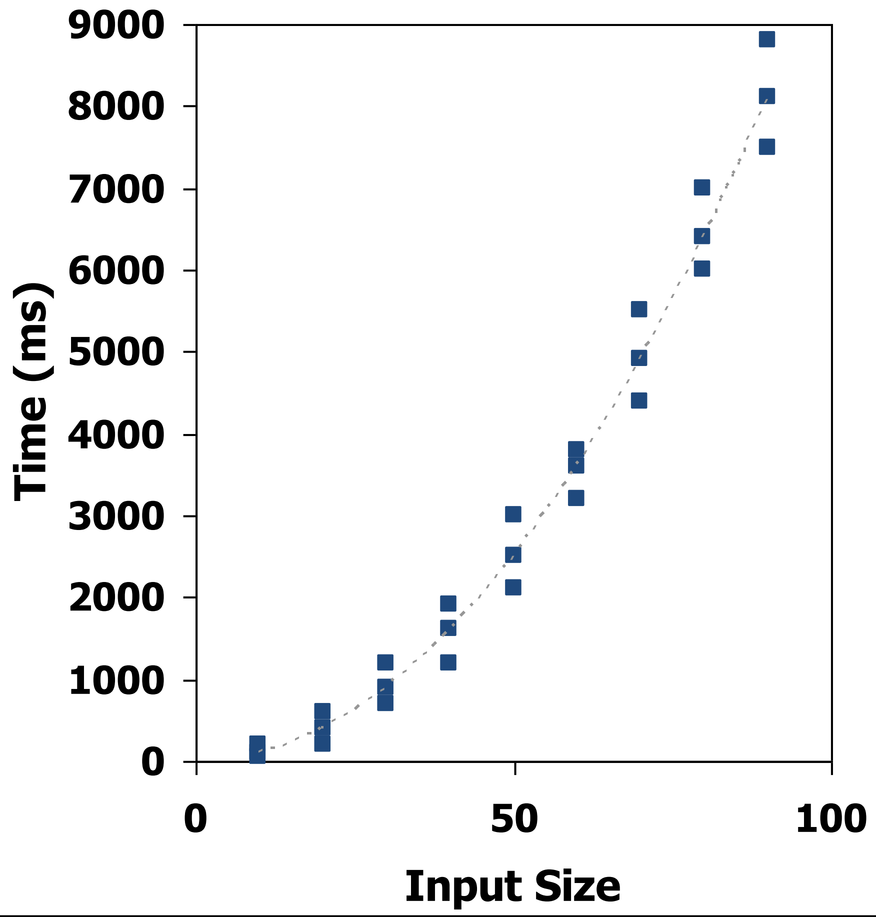
Running Time

- 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



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



```
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
```

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult, time consuming or costly.
- 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.

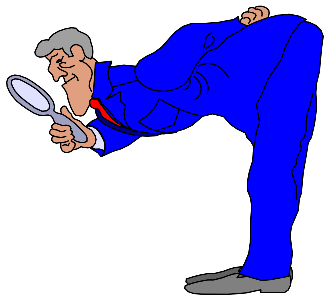
Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

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

Pseudocode Details



- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

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

  Input …

  Output . . .

- Method call

  method (***arg*** [, ***arg***…])

- Return value

  return ***expression***

- Expressions:

  $\leftarrow$ Assignment

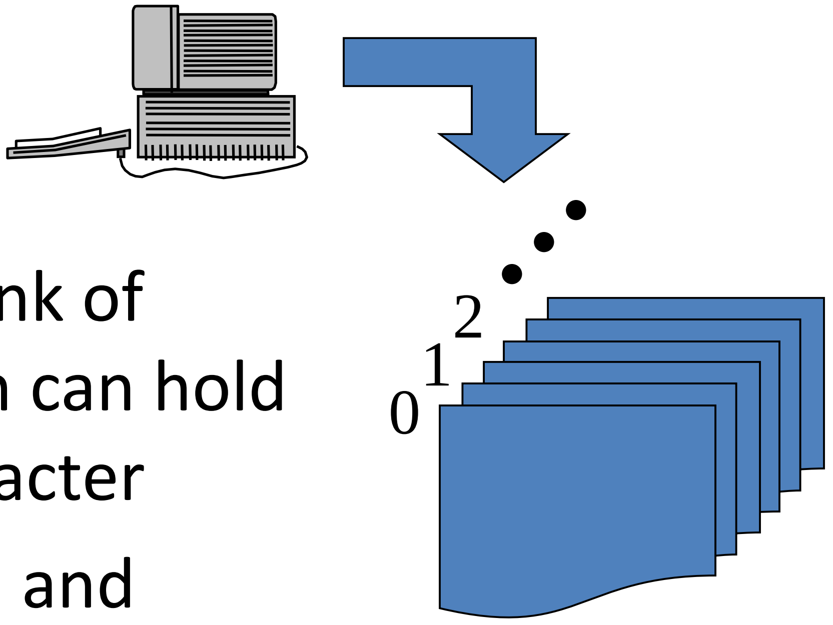
  $=$ Equality testing

  n^2 Superscripts and other mathematical formatting allowed

The Random Access Machine (RAM) Model

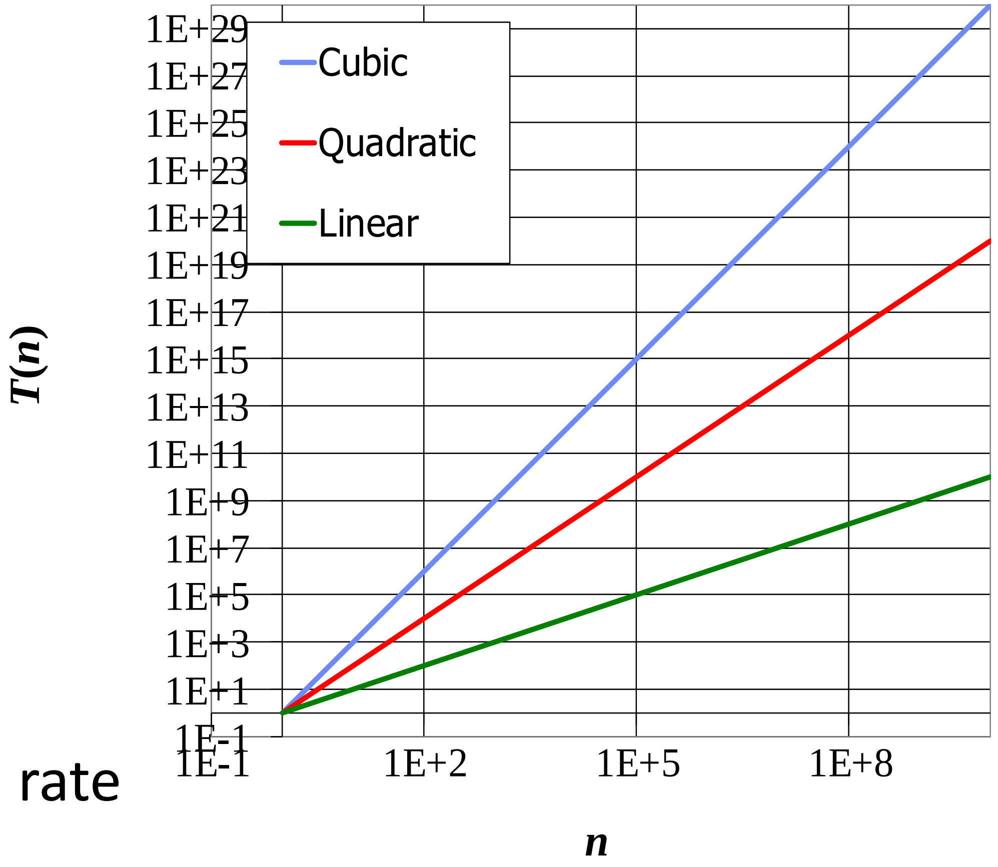
A **RAM** consists of

- A **CPU**
- A 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



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



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:

- Evaluating an expression
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method

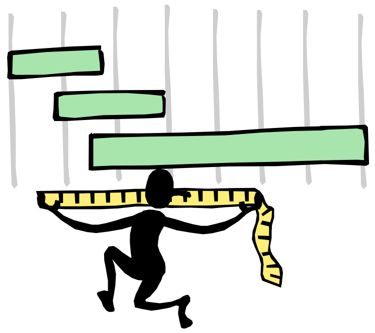
Counting Primitive Operations

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

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  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;
9  }
```

STEP	3	4	5	6	7	8	TOTAL
# ops	2	2	2n	2n-2	0 to 2n-2	1	4n+3 to 6n+1

Estimating Running Time



- Algorithm arrayMax executes $6\boldsymbol{n} + 1$ primitive operations in the worst case, $4\boldsymbol{n} + 3$ in the best case. Define:

  a = Time taken by the fastest primitive operation

  b = Time taken by the slowest primitive operation

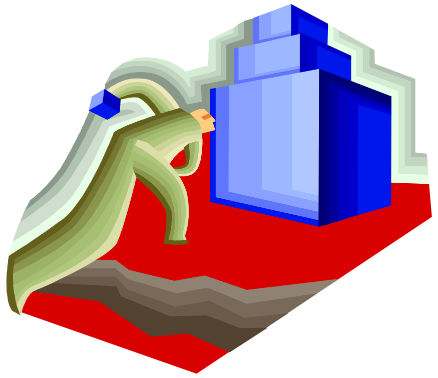
- Let $\boldsymbol{T}(\boldsymbol{n})$ be worst-case time of arrayMax. Then

$$\boldsymbol{a}\,(4\boldsymbol{n} + 3) \le \boldsymbol{T}(\boldsymbol{n}) \le \boldsymbol{b}(6\boldsymbol{n} + 1)$$

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

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

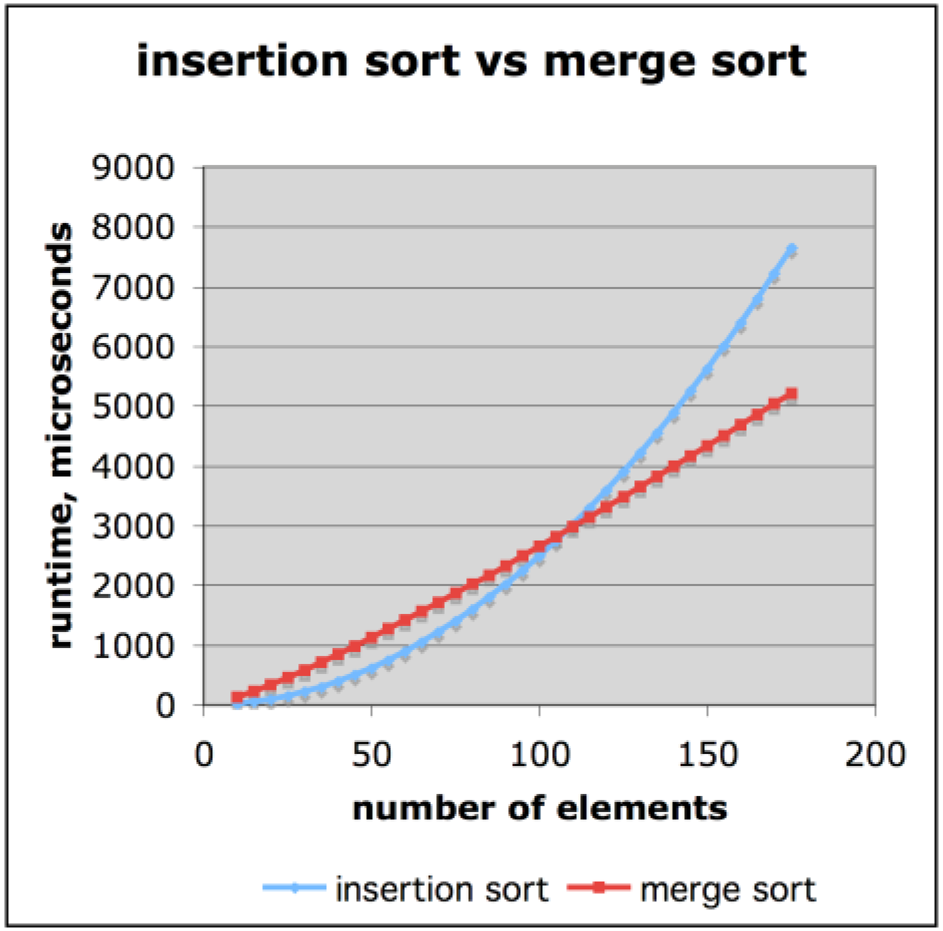


Why Growth Rate Matters

if runtime is...	time for n + 1	time for 2 n	time for 4 n
$c \lg n$	$c \lg (n + 1)$	$c (\lg n + 1)$	$c(\lg n + 2)$
$c\, n$	$c\,(n + 1)$	$2c\, n$	$4c\, n$
$c\, n \lg n$	$\sim c\, n \lg n + c\, n$	$2c\, n \lg n + 2cn$	$4c\, n \lg n + 4cn$
$c\, n^2$	$\sim c\, n^2 + 2c\, n$	**$4c\, n^2$**	$16c\, n^2$
$c\, n^3$	$\sim c\, n^3 + 3c\, n^2$	$8c\, n^3$	$64c\, n^3$
$c\, 2^n$	$c\, 2^{n+1}$	$c\, 2^{2n}$	$c\, 2^{4n}$

runtime quadruples when problem size doubles

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

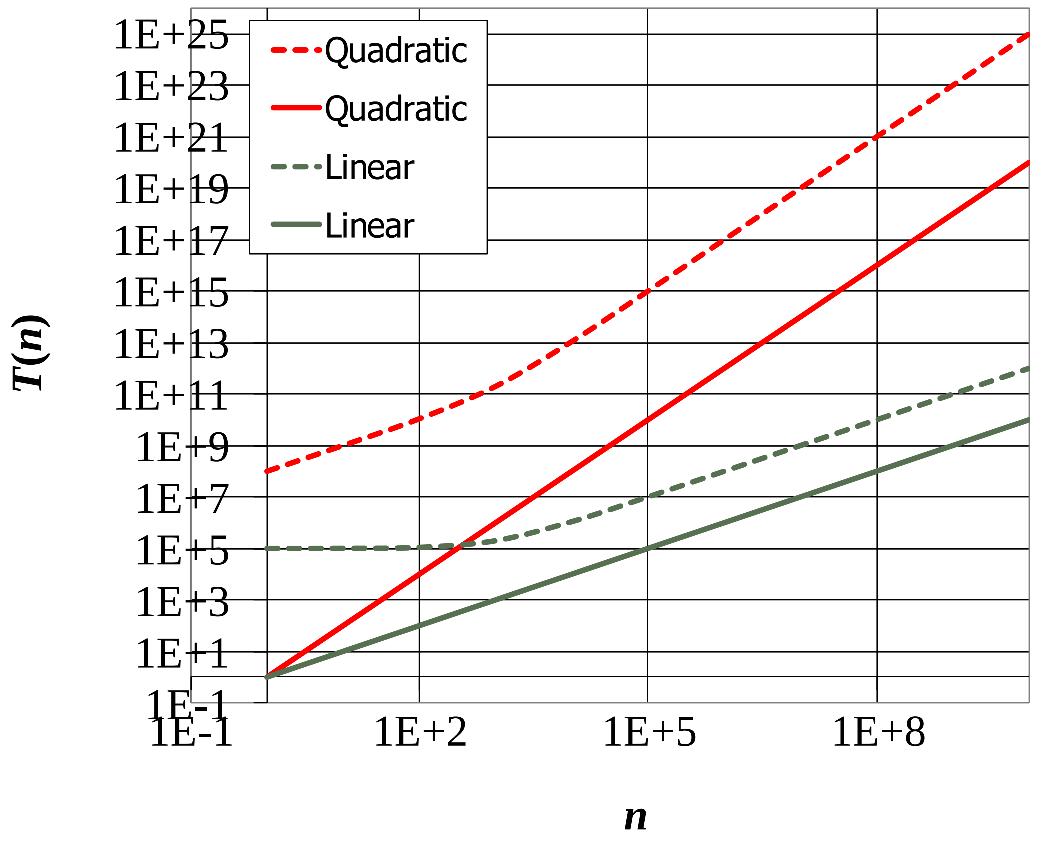
while

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**
  - $10^2 \boldsymbol{n} + 10^5$ is a linear function
  - $10^5 \boldsymbol{n}^2 + 10^8 \boldsymbol{n}$ is a quadratic function

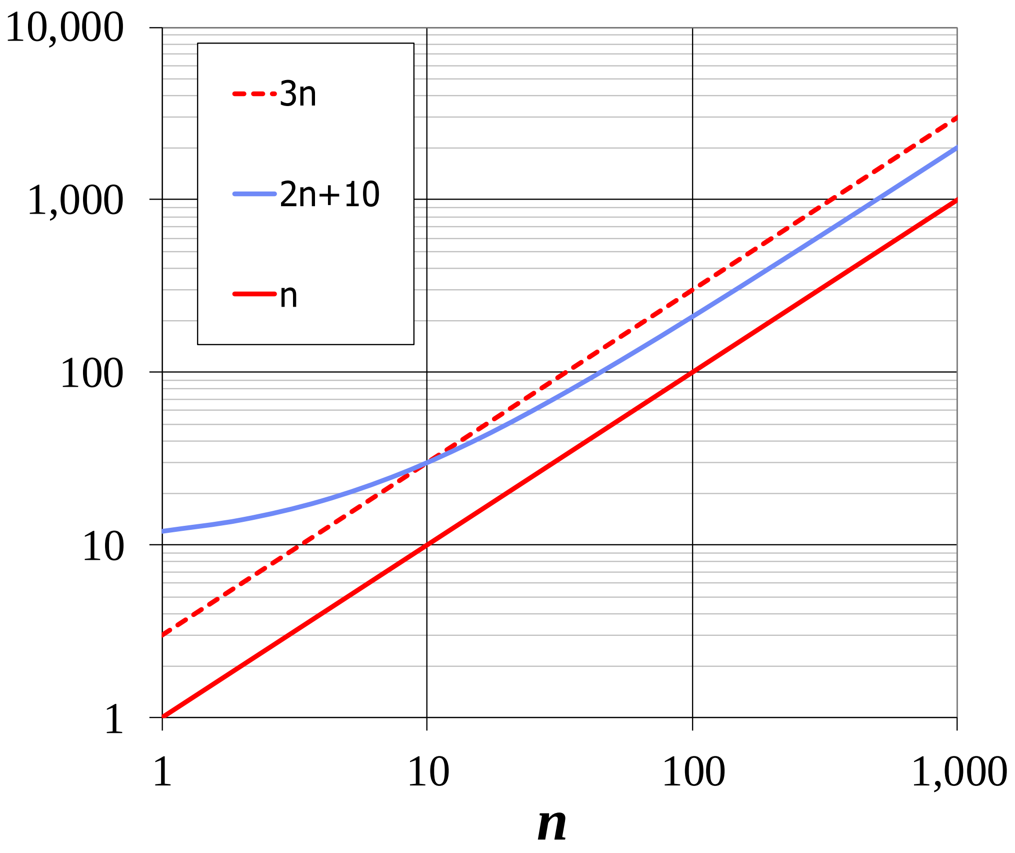


Big-Oh Notation

- 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

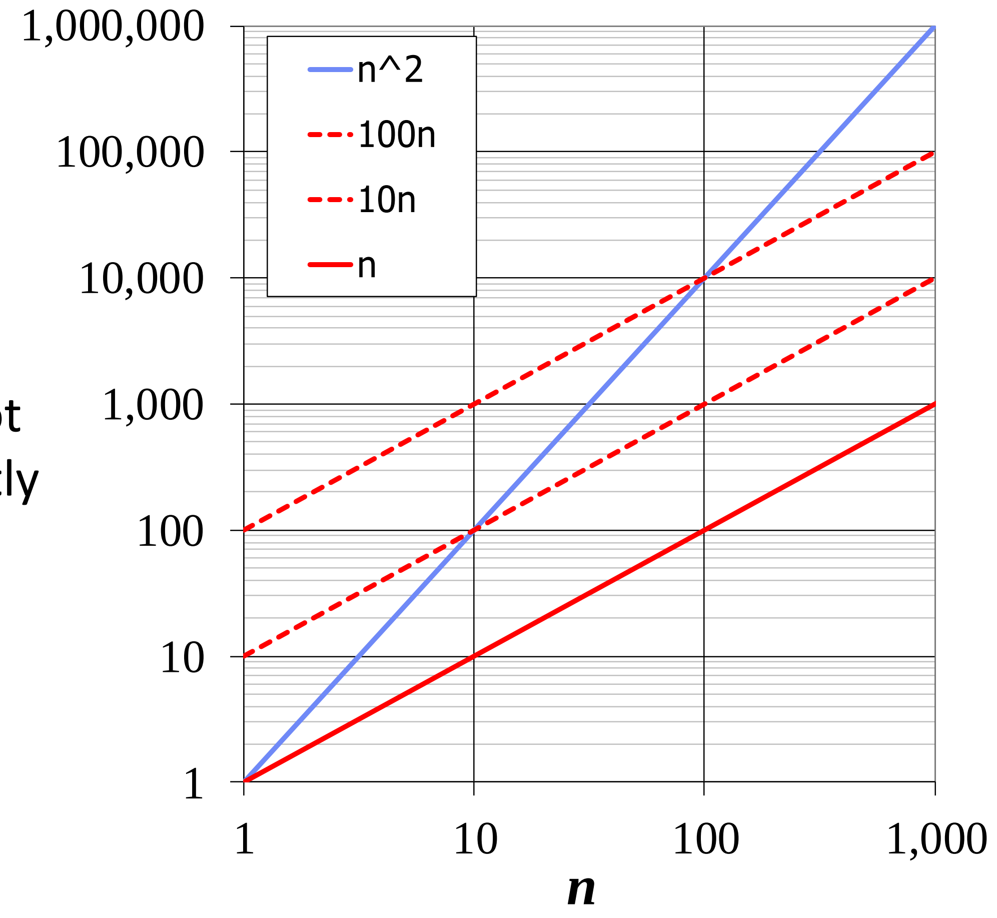
  $f(n) \leq cg(n)$ for $n \geq n_0$

- **Example:** $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



Big-Oh Example

- **Example:**
  the function $\boldsymbol{n}^2$ is not $\boldsymbol{O}(\boldsymbol{n})$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied for all sufficiently large n, since ***c*** must be a constant



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 $n_0 = 2$

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

Big-Oh Rules

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  - drop lower-order terms
  - 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 $O(3n)$"

Asymptotic Algorithm Analysis

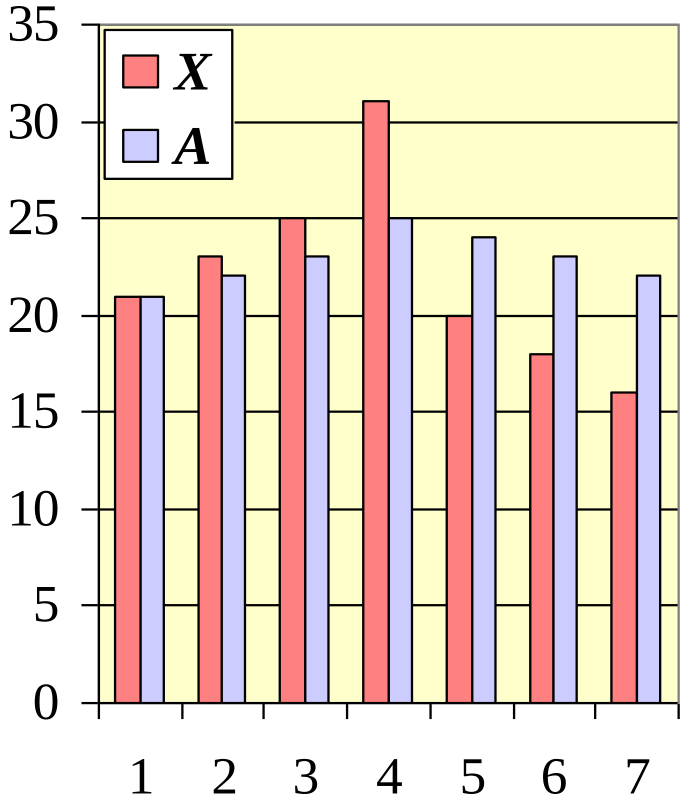
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
    - We find the worst-case number of primitive operations executed as a function of the input size
    - We express this function with big-Oh notation
- **Example:**
    - We 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

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



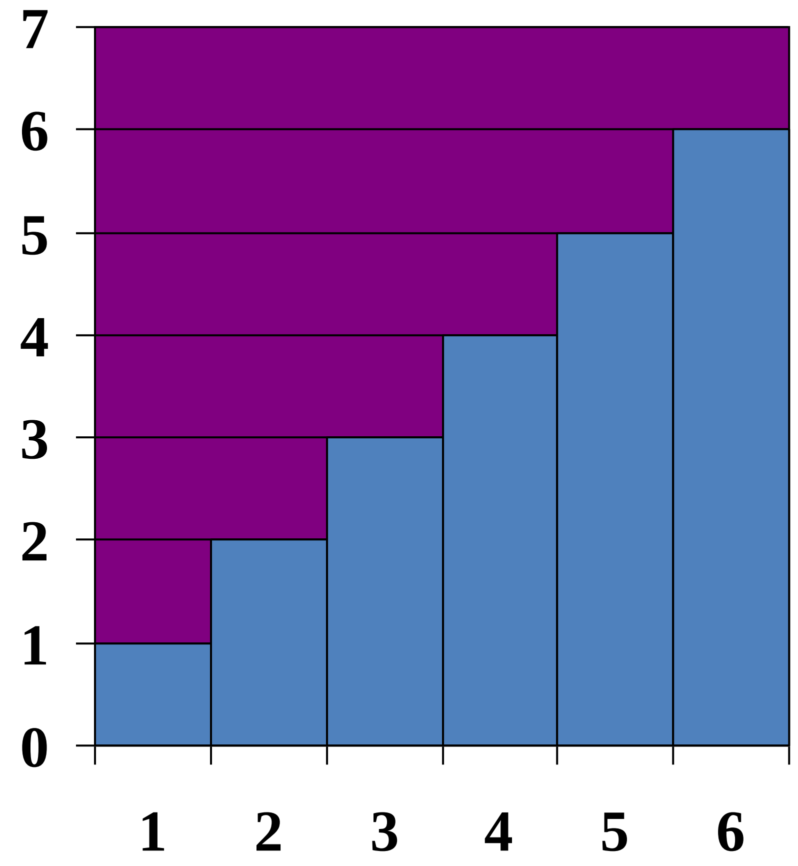
Prefix Averages (Quadratic)

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;
}
```

Arithmetic Progression

- The running time of prefixAverage1 is ***O***(1 + 2 + …+ ***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



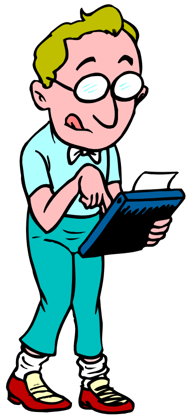
Prefix Averages 2 (Linear)

The following algorithm uses a running sum to improve 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
  - Induction
  - . . .
- Basic probability

- Properties of powers:

$$a^{b+c} = a^b * a^c$$
$$a^{b-c} = a^b / a^c$$
$$a^{b*c} = \left(a^b\right)^c$$
$$a^b = a^{b \log_c c} = c^{b \log_c a}$$

- Properties of logarithms:

$$\log_b(x * y) = \log_b x + \log_b y$$
$$\log_b(x/y) = \log_b x - \log_b y$$
$$\log_b x^a = a \log_b x$$
$$x^{\log y} = y^{\log x}$$
$$\log_b x = (\log_c x)/(\log_c b)$$

Relatives of Big-Oh

big-Omega

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

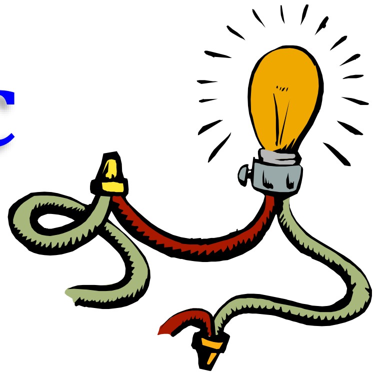
$$f(n) \geq c\, g(n) \qquad \text{for } n \geq n_0$$

big-Theta

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

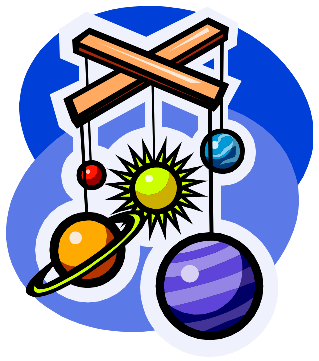
$$c'\, g(n) \leq f(n) \leq c''\, g(n) \qquad \text{for } n \geq n_0$$

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)

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 $\mathrm{f}(n) \geq c\, g(n)$ 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 $\mathrm{f}(n) \leq c\, g(n)$ for $n \geq n_0$.
Let $c = 5$ and $n_0 = 1$.

Part 1: Summary

- Analyzing running time of algorithms
  - Experimentation & its limitations
  - Theoretical analysis
- Pseudo-code
- RAM: Random Access Machine
- 7 important functions
- Asymptotic notations: $O()$, $\Omega()$, $\Theta()$
- Asymptotic running time analysis of algorithms

Part 2: Correctness

Outline

- Iterative Algorithms:
  Assertions and Proofs of Correctness

- Binary Search: A Case Study

Assertions

- An **assertion** is a statement about the state of the data at a specified point in your algorithm.

- An assertion is not a task for the algorithm to perform.

- You may think of it as a comment that is added for the benefit of the reader.

Loop Invariants

- Binary search can be implemented as an **iterative algorithm** (it could also be done recursively).

- **Loop Invariant:** An **assertion** about the current state useful for designing, analyzing and proving the correctness of iterative algorithms.

Other Examples of Assertions

- **Pre-conditions:** Any assumptions that must be true about the input instance.
- **Post-conditions:** The statement of what must be true when the algorithm/program returns.
- **Exit-condition:** The statement of what must be true to exit a loop.

Iterative Algorithms

Take one step at a time

towards the final destination

```
loop {
      if (done) { exit loop }
      take step
}
```

An Example

Problem: **Input:** natural number n.

Output: 2^n.

Restriction: No arithmetic ops other than "+".

```
Algorithm exp2( n )



i ← 0 ,  j ← 1
loop {
        if ( $i \geq n$ ) { exit loop }
        i ← i + 1 ,  j ← j + j
}
return j
end
```

An Example

Problem: **Input:** natural number n.

Output: 2^n.

Restriction: No arithmetic ops other than "+".

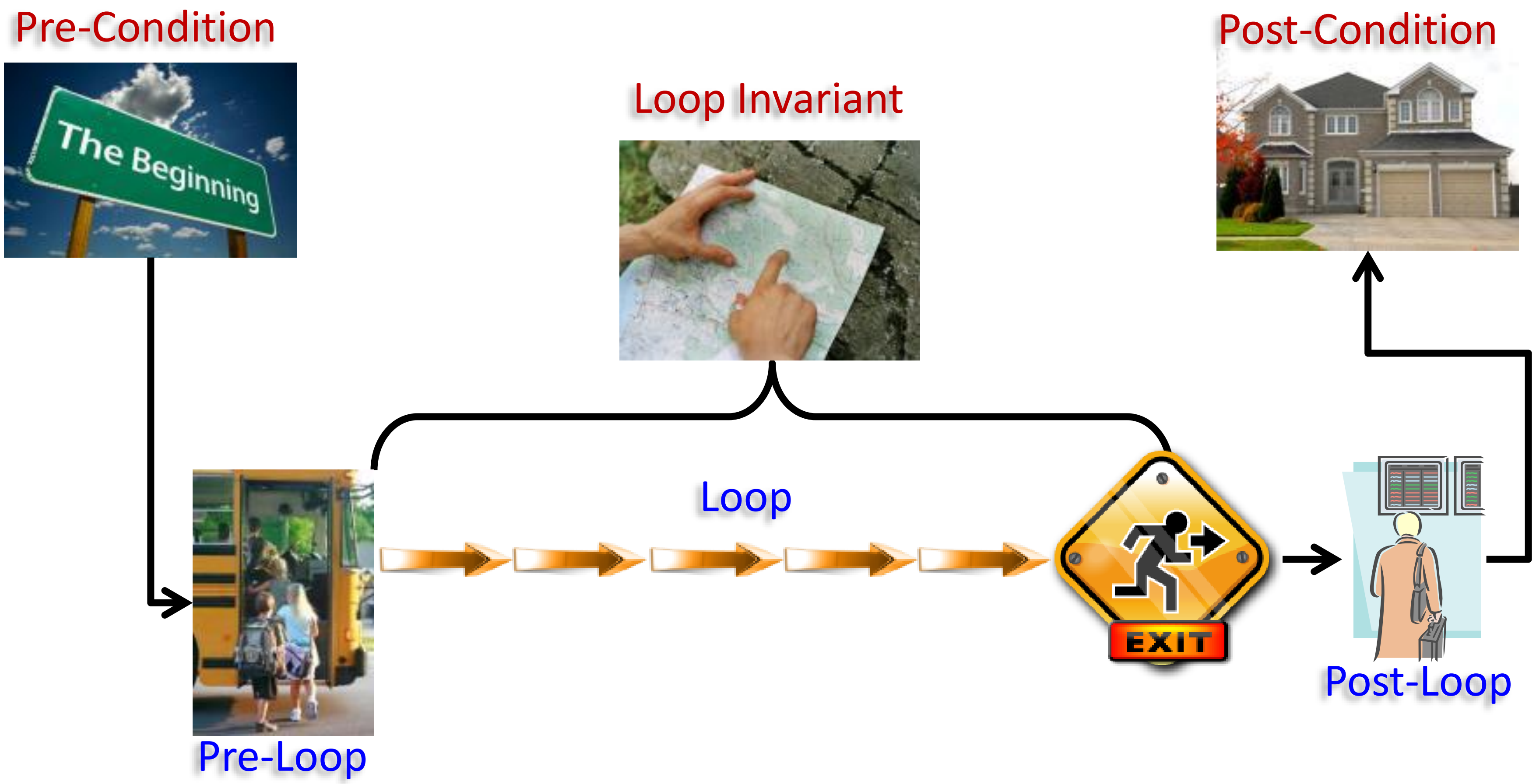
```
Algorithm  exp2( n )
Pre-Condition: input n is a natural number.
Post-Condition: outputs $2^n$
i ← 0 ,  j ← 1     // Pre-Loop  code
loop {    // Loop-Invariant: $j = 2^i$ $and$ $i \le n$
          if ( $i \ge n$ ) { exit loop }
          i ← i + 1 ,  j ← j + j    // Loop body
}
return j          // Post-Loop code
end
```

establish LI

maintain LI
& make progress

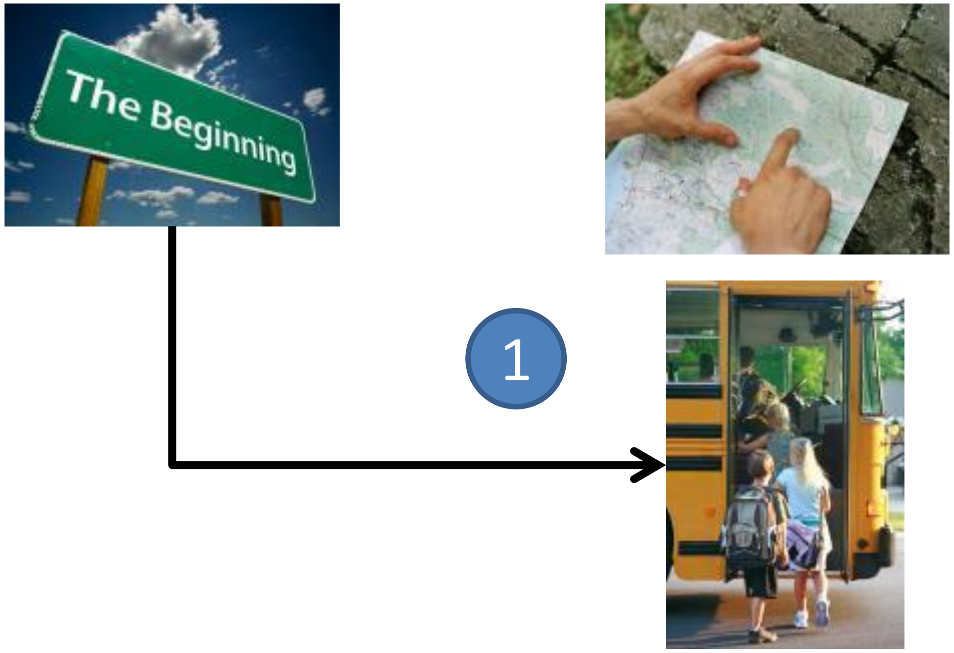
use LI & exit-cond
to establish
post-condition

From Pre-Condition to Post-Condition

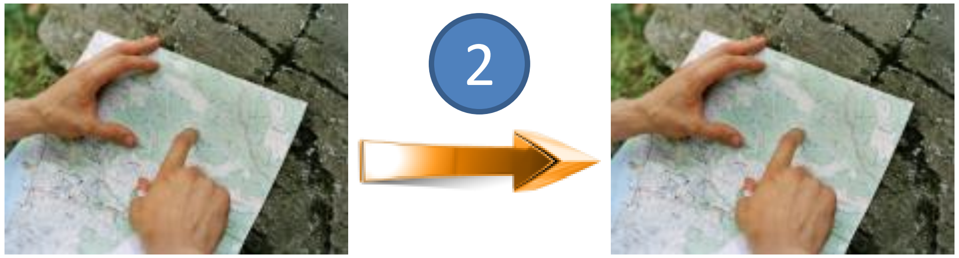


Establishing Loop Invariant

1. from the pre-condition on the input instance, establish the loop invariant.



Maintain Loop Invariant



- Suppose that
  1) We start in a safe location (loop invariant just established from pre-condition)
  2) If we are in a safe location (loop invariant), we always step to another safe location (loop invariant)
- Can we be assured that the computation will always keep us in a safe location?
- By what principle?

Maintain Loop Invariant

By Induction the computation will always be in a safe location.

$S(i) =$ loop invariant
at the end of iteration i
(beginning of iteration $i+1$)

1 $S(0)$

and

2 $\forall i, \quad S(i) \Longrightarrow S(i+1)$

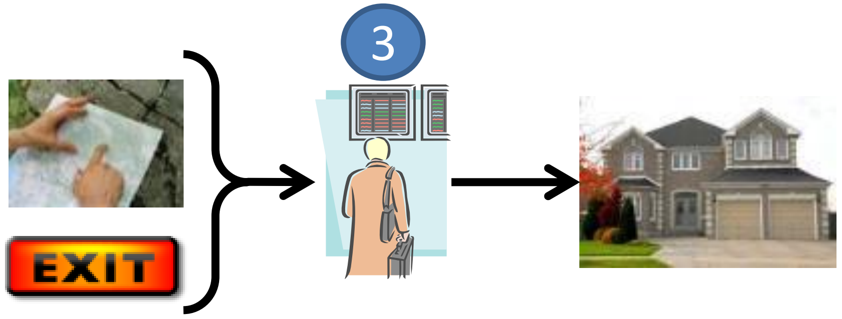
$$\left.\begin{array}{c} S(0) \\ and \\ \forall i, \quad S(i) \Longrightarrow S(i+1) \end{array}\right\} \Longrightarrow \forall i, \quad S(i)$$

Ending The Algorithm

- Define **Exit Condition**

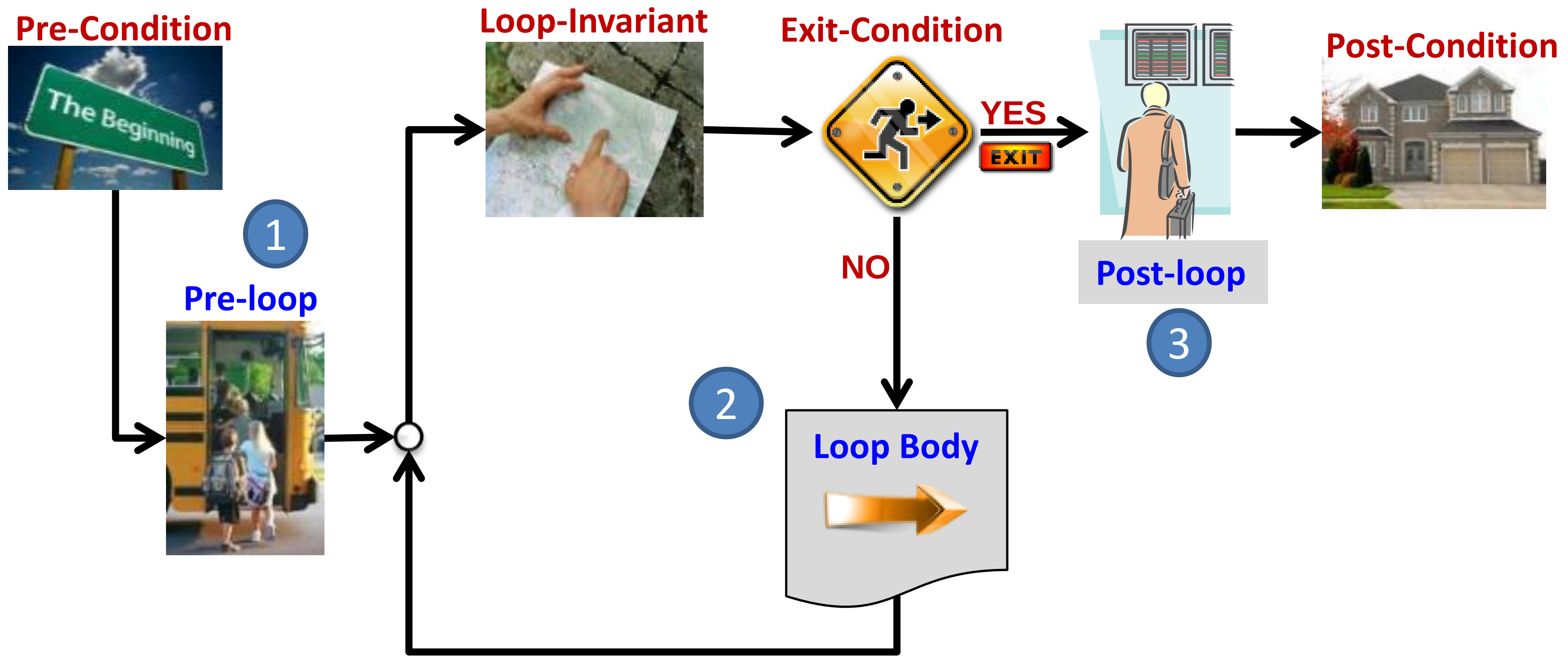
- **Termination:** With sufficient progress,
  the exit condition will be met.

- When we exit, we know
  - loop invariant is true
  - exit condition is true



3) from these we must establish the **post-condition**.

Iterative Algorithm



Definition of Correctness

<PreCond> & <code> ⇒ <PostCond>

If the input meets the pre-conditions,
then the output must meet the post-conditions.

If the input does not meet the preconditions, then nothing is required.

Binary Search: a case study

Define Problem: Binary Search

- PreConditions
  - Key 25
  - Sorted indexed List A

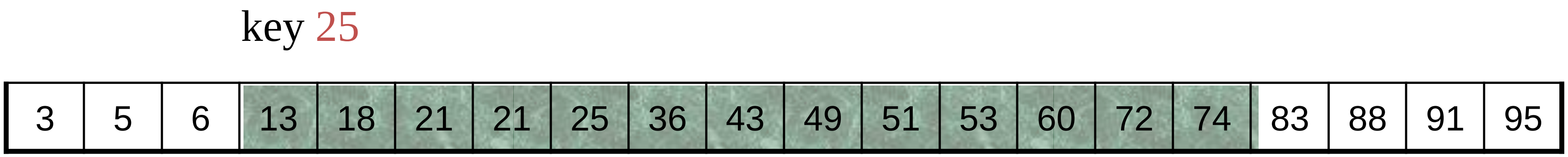
3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

- PostConditions
  - Find key in list (if there).

3	5	6	13	18	21	21	$\boxed{25}$	36	43	49	51	53	60	72	74	83	88	91	95

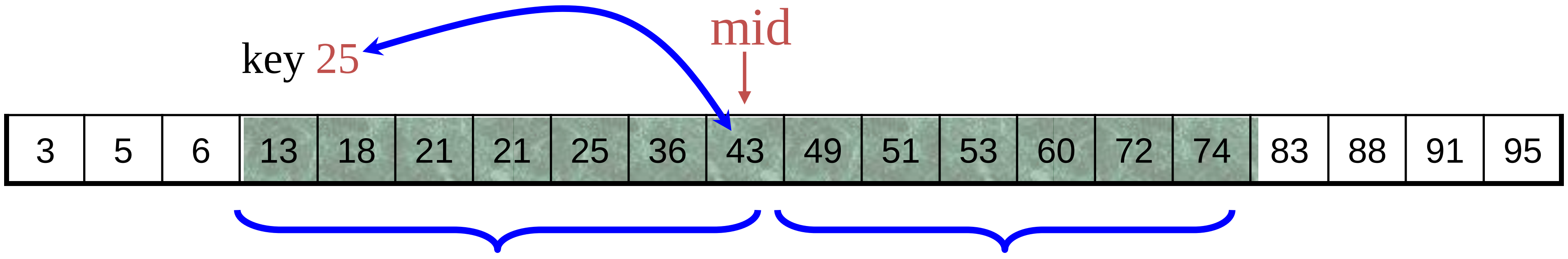
Define Loop Invariant

- Maintain a sub-list.
- LI: If the key is contained in the original list, then the key is contained in the sub-list.



Define Step

- Cut sub-list in half.
- Determine which half the key would be in.
- Keep that half.

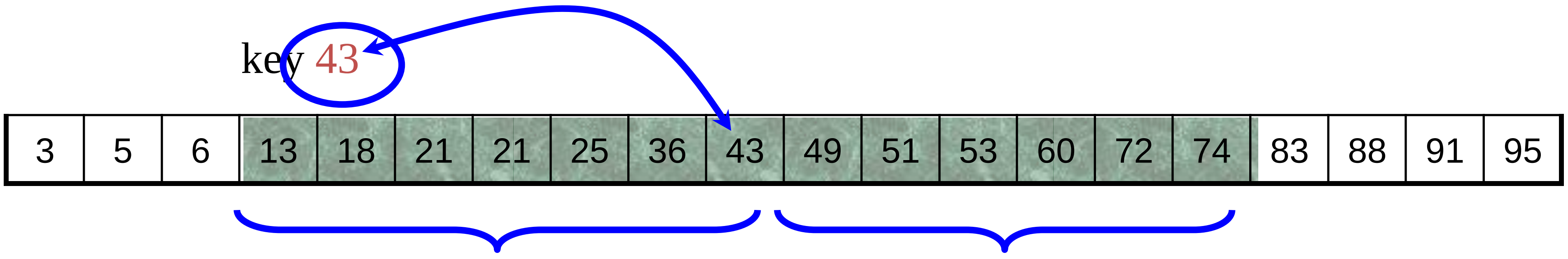


If key ≤ A[mid],
then key is in
left half.

If key > A[mid],
then key is in
right half.

Define Step

- It is faster not to check if the middle element is the key.
- Simply continue.

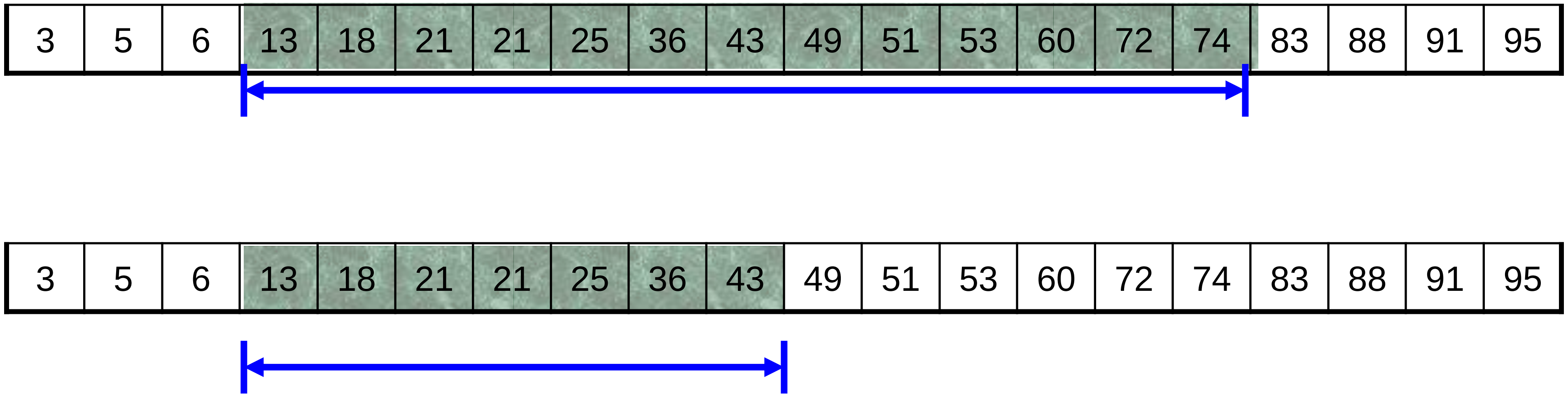


If key ≤ A[mid], then key is in left half.

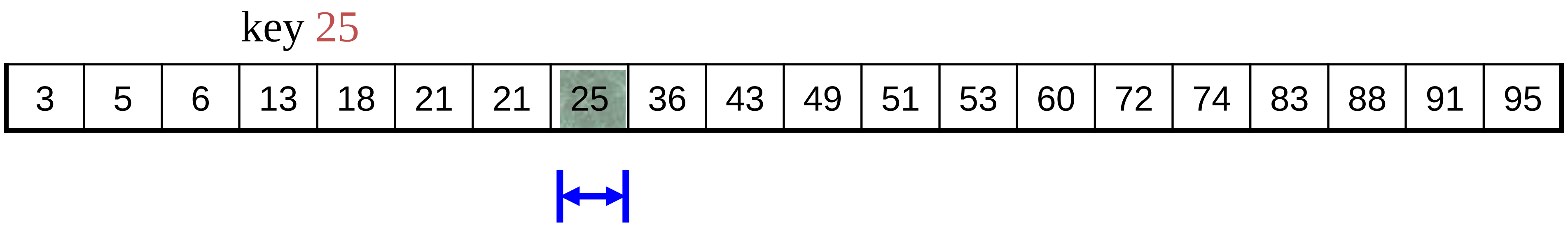
If key > A[mid], then key is in right half.

Make Progress

- The size of the list becomes smaller.



Exit Condition



3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

- LI: If the key is contained in the original list, then the key is contained in the sub-list.
- Sub-list contains one element.

EXIT

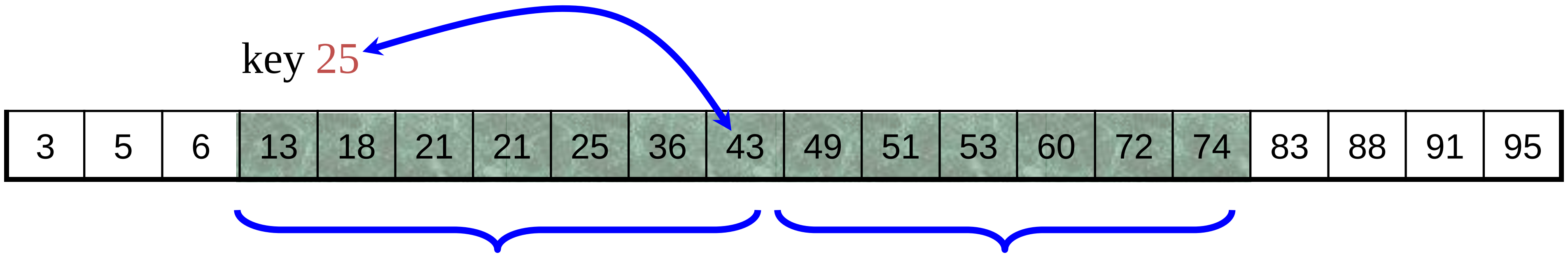
- If element = key, return associated entry.
- Otherwise return false.

Running Time

The sub-list is of size $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots, 1$

Each step O(1) time.

Total time = O(log n)



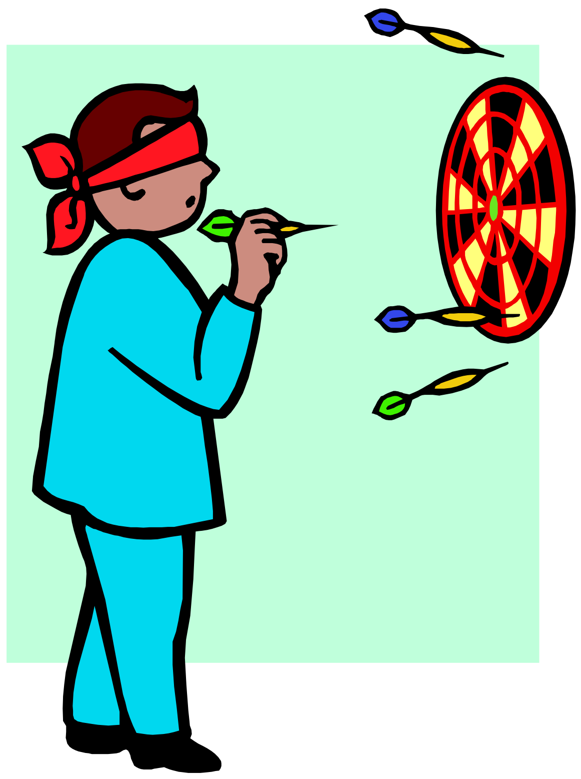
If key ≤ A[mid], then key is in left half.

If key > A[mid], then key is in right half.

```
Algorithm BinarySearch ( A[1..n] , key)
    <precondition>:   A[1..n] is sorted in non-decreasing order
    <postcondition>:  If key is in A[1..n], output is its location
    $p \leftarrow 1$ ,   $q \leftarrow n$
    while  $p < q$  do
        <Loop-invariant>:  if key is in A[1..n], then key is in A[p..q]
        $mid \leftarrow \left\lfloor \frac{p+q}{2} \right\rfloor$
        if  $key \le A[mid]$   then $q \leftarrow mid$
                               else $p \leftarrow mid + 1$
    end while
    if  $key = A[p]$
        then return (p)
        else return ("key not in list")
end algorithm
```

Simple, right?

- Although the concept is simple, binary search is notoriously easy to get wrong.
- Why is this?



Boundary Conditions

- The basic idea behind binary search is easy to grasp.
- It is then easy to write pseudo-code that works for a 'typical' case.
- Unfortunately, it is equally easy to write pseudo-code that fails on the *boundary conditions*.

Boundary Conditions

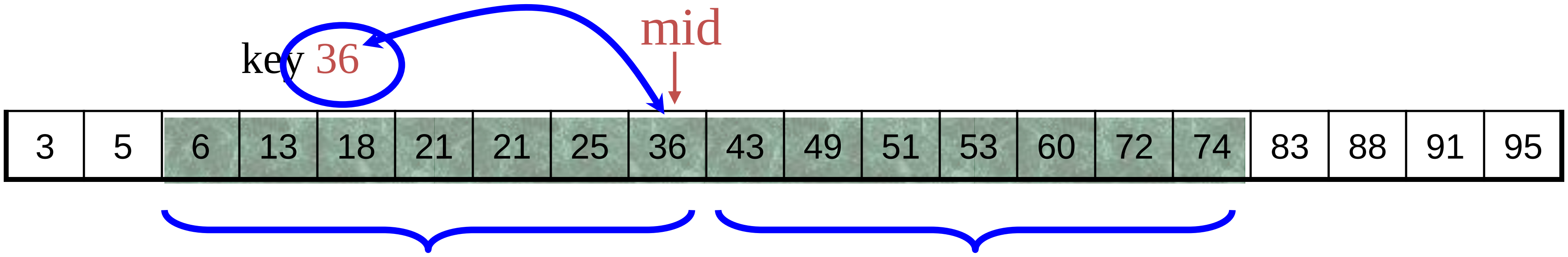
```
if  $key \le A[mid]$
      then  $q \leftarrow mid$
      else  $p \leftarrow mid + 1$
```

or

```
~~if  $key < A[mid]$~~
~~      then  $q \leftarrow mid$~~
~~      else  $p \leftarrow mid + 1$~~
```

What condition will break the loop invariant?

Boundary Conditions



Code: key ≥ A[mid] → select right half

Bug!!

Boundary Conditions

```
if $key \le A[mid]$
    then $q \leftarrow mid$
    else $p \leftarrow mid + 1$
```

OK

```
if $key < A[mid]$
    then $q \leftarrow mid - 1$
    else $p \leftarrow mid$
```

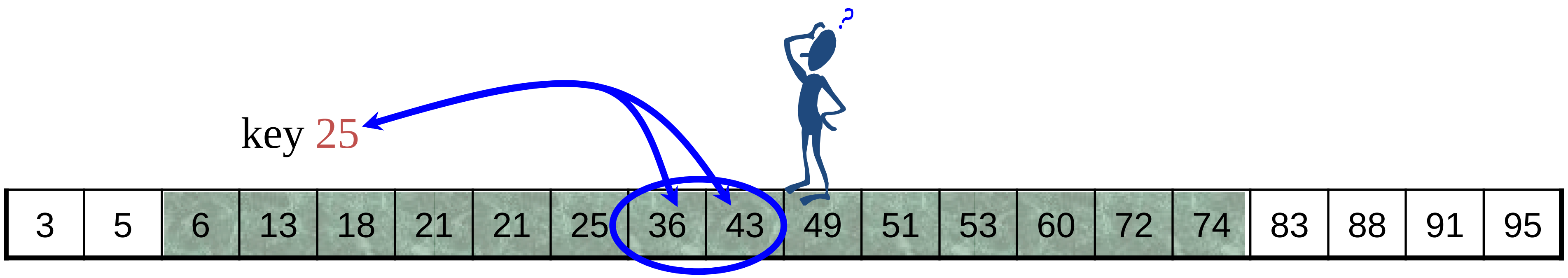
OK

```
~~if $key < A[mid]$~~
    ~~then $q \leftarrow mid$~~
    ~~else $p \leftarrow mid + 1$~~
```

Not OK !!

Boundary Conditions

$$\text{mid} = \left\lfloor \frac{p+q}{2} \right\rfloor \quad \text{or} \quad \text{mid} = \left\lceil \frac{p+q}{2} \right\rceil$$

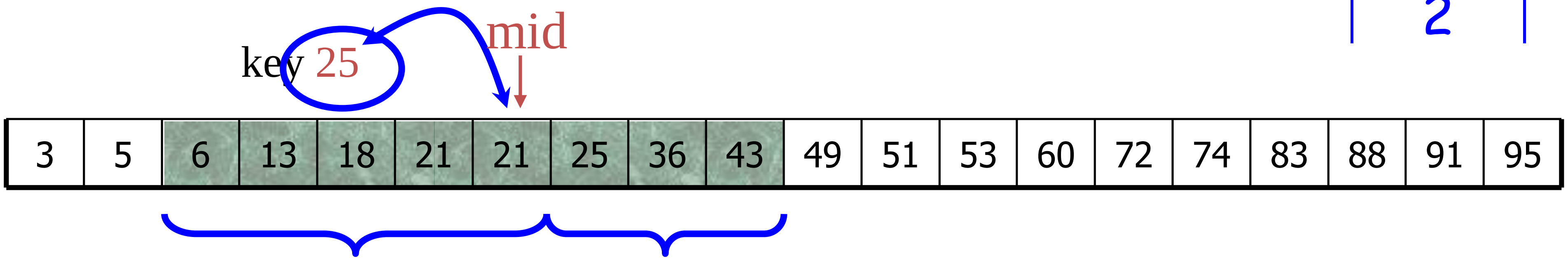


Shouldn't matter, right?

Select $\text{mid} = \left\lceil \frac{p+q}{2} \right\rceil$

Boundary Conditions

Select $\text{mid} = \left\lceil \frac{p+q}{2} \right\rceil$

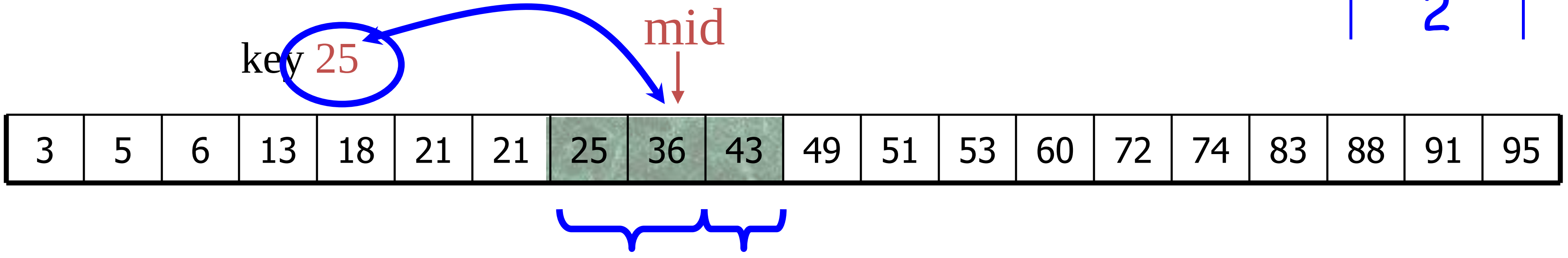


If key ≤ A[mid], then key is in left half.

If key > A[mid], then key is in right half.

Boundary Conditions

Select $\text{mid} = \left\lceil \frac{p+q}{2} \right\rceil$



If key ≤ A[mid], then key is in left half.

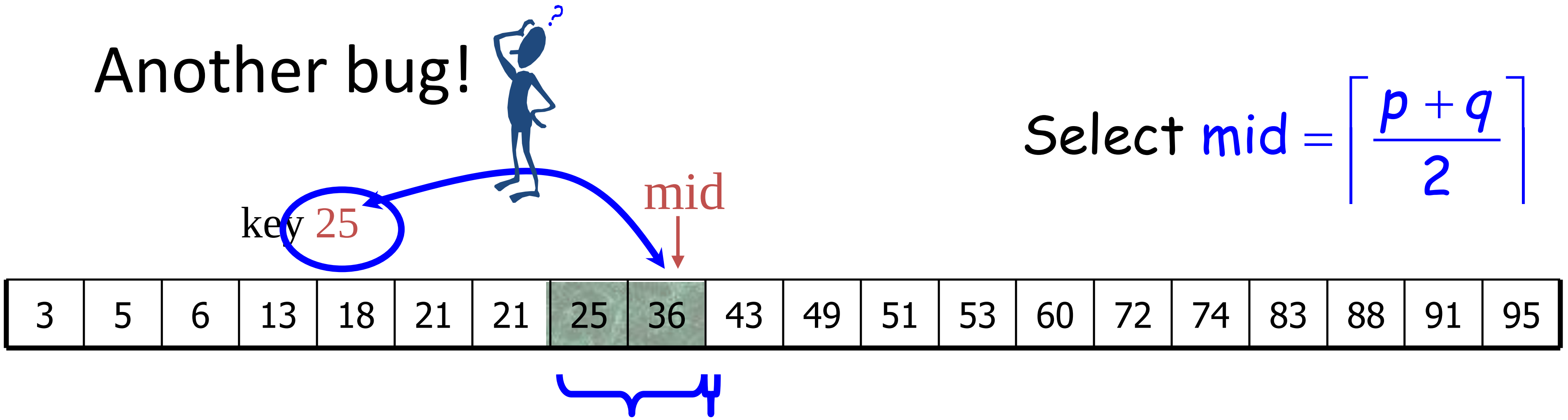
If key > A[mid], then key is in right half.

Boundary Conditions

No progress toward goal:
Loops Forever!

Another bug!

Select $\text{mid} = \left\lceil \frac{p+q}{2} \right\rceil$



3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

If key ≤ A[mid], then key is in left half.

If key > A[mid], then key is in right half.

Boundary Conditions

```
mid ← $\left\lfloor \frac{p+q}{2} \right\rfloor$
if key ≤ A[mid]
    then q ← mid
    else p ← mid + 1
```

OK

```
mid ← $\left\lceil \frac{p+q}{2} \right\rceil$
if key < A[mid]
    then q ← mid - 1
    else p ← mid
```

OK

```
~~mid ← $\left\lceil \frac{p+q}{2} \right\rceil$~~
~~if key ≤ A[mid]~~
~~    then q ← mid~~
~~    else p ← mid + 1~~
```

Not OK !!

Getting it Right

- How many possible algorithms?
- How many **correct** algorithms?

```
mid ← $\left\lfloor \frac{p+q}{2} \right\rfloor$
if key ≤ A[mid]
    then q ← mid
    else p ← mid + 1
```

or
```
mid ← $\left\lceil \frac{p+q}{2} \right\rceil$
```
?

or
```
if key < A[mid]
```
?

or
```
then q ← mid - 1
else p ← mid
```
?

Alternative Algorithm: Less Efficient but More Clear

```
Algorithm BinarySearch ( A[1..n] , key)
        <precondition>: A[1..n] is sorted in non-decreasing order
        <postcondition>: If key is in A[1..n], output is its location
        p ← 1 , q ← n
        while p < q do
            <Loop-invariant>: if key is in A[1..n], then key is in A[p..q]
            mid ← $\left\lfloor \frac{p+q}{2} \right\rfloor$
            if key < A[mid]
                then q ← mid - 1
                else if key > A[mid]
                    then p ← mid + 1
                    else return (mid)
        end while
        return ("key not in list")
end algorithm
```

Still O(log n), but with slightly larger constant.

Any bugs ?

Part 2: Summary

From Part 2, you should be able to:

- Use the loop invariant method to think about iterative algorithms.
- Prove that the loop invariant is established.
- Prove that the loop invariant is maintained in the ‘typical’ case.
- Prove that the loop invariant is maintained at all boundary conditions.
- Prove that progress is made in the ‘typical’ case
- Prove that progress is guaranteed even near termination, so that the exit condition is always reached.
- Prove that the loop invariant, when combined with the exit condition, produces the post-condition.