Algorithm Analysis 1

Counting

Algorithm Analysis

- Measuring algorithm efficiency – 1st presentation
- Cost functions
  - Searching – 2nd presentation
  - Sorting
    - Simple sorting – 3rd presentation
    - Quicksort – 4th presentation
- O Notation – 5th presentation

Algorithm Analysis

- Algorithms can be described in terms of time and space efficiency

- Time
  - How long, in *ms*, does my algorithm take to solve a particular problem?
  - How much does the time increase as the problem size increases?
  - How does my algorithm compare to other algorithms?
- Space
  - How much memory space does my algorithm require to solve the problem?

Usability

- Choosing an appropriate algorithm can make a significant difference in the usability of a system
  - Government and corporate databases with many millions of records, which are accessed frequently
  - Online search engines and media platforms
  - Big data
  - Real time systems where near instantaneous response is required
    - From air traffic control systems to computer games

Comparing Algorithms

- There are often many ways to solve a problem
  - Different algorithms that produce the same results
    - e.g. there are numerous *sorting* algorithms
- We are usually interested in how an algorithm performs when its input is large
  - In practice, with today's hardware, *most* algorithms will perform well with small input
  - There are exceptions to this, such as the *Traveling Salesman Problem*
    - Or the recursive Fibonacci algorithm presented previously …

Measuring Algorithms

- It is possible to *count* the number of operations that an algorithm performs
  - By a careful visual walkthrough of the algorithm or by
  - Inserting code in the algorithm to count and print the number of times that each line executes profiling
- It is also possible to *time* algorithms
  - Compare system time before and after running an algorithm
    - More sophisticated timer classes exist
  - Simply timing an algorithm may ignore a variety of issues

Timing Algorithms

- It may be useful to time how long an algorithm takes to run
  - In some cases it may be *essential* to know how long an algorithm takes on a particular system
    - e.g. air traffic control systems
    - Running time may be a strict requirement for an application
- But is this a good *general* comparison method?
  - Running time is affected by a number of factors other than algorithm efficiency

Running Time is Affected By

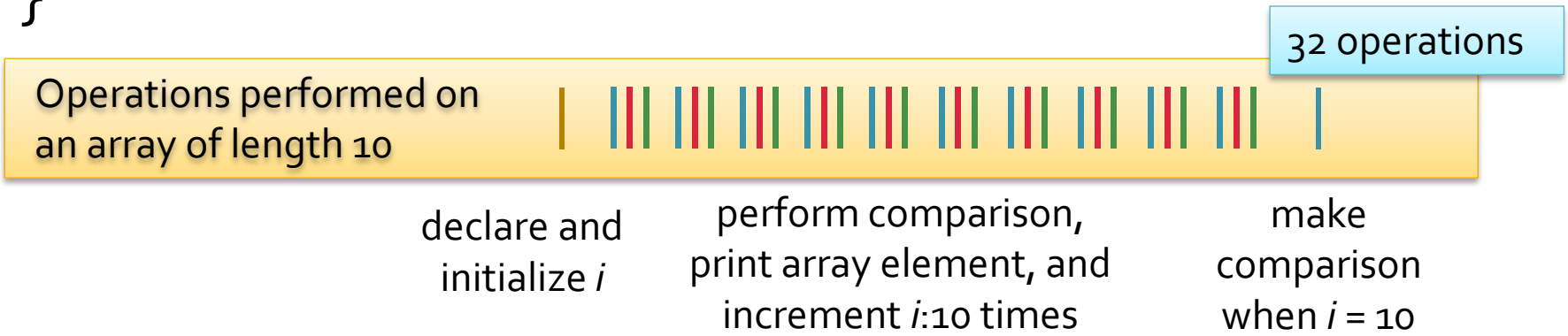
- CPU speed
- Amount of main memory
- Specialized hardware (e.g. graphics card)
- Operating system
- System configuration (e.g. virtual memory)
- Programming language
- Algorithm implementation
- Other programs
- System tasks (e.g. memory management)
- …

Counting

- Instead of *timing* an algorithm, *count* the number of instructions that it performs
- The number of instructions performed may vary based on
  - The size of the input
  - The organization of the input
- The number of instructions can be written as a cost function on the input size

A Simple Example

```
void printArray(int arr[], int n){
  for (int i = 0; i < n; ++i){
    cout << arr[i] << endl;
  }
}
```



Cost Functions

- Instead of choosing a particular input size we will express a cost function for input of size n
  - We assume that the running time, t, of an algorithm is proportional to the number of operations
- Express t as a function of n
  - Where t is the time required to process the data using some algorithm A
  - Denote a cost function as $t_A(n)$
    - i.e. the running time of algorithm A, with input size n

A Simple Example

```
void printArray(int arr[], int n){
  for (int i = 0; i < n; ++i){
    cout << arr[i] << endl;
  }
}
```

$t = 3n + 2$

Operations performed on an array of length **n**	1	3n	1
	declare and initialize *i*	perform comparison, print array element, and increment *i*: *n* times	make comparison when *i* = *n*

What's an Operation?

- In the example we assumed two things
  - Neither of which are strictly true …
- Any C++ statement counts as a single operation
  - Unless it is a function call
- That all operations take the same amount of time
  - Some fundamental operations are faster than others
  - What is a fundamental operation in a high level language is multiple operations in assembly
- These are both simplifying assumptions

Input Varies

- The number of operations often varies based on the size of the input
  - Though not always – consider array lookup
- In addition algorithm performance may vary based on the *organization* of the input
  - For example consider searching a large array
  - If the target is the first item in the array the search will be very fast

Best, Average and Worst Case

- Algorithm efficiency is often calculated for three broad cases of input
  - Best case
  - Average (or "usual") case
  - Worst case
- This analysis considers how performance varies for *different* inputs of the *same* size

Analyzing Algorithms

- It can be difficult to determine the exact number of operations performed by an algorithm
  - Though it is often still useful to do so
- An alternative to counting all instructions is to focus on an algorithm's *barometer instruction*
  - The barometer instruction is the instruction that is executed the most number of times in an algorithm
  - The number of times that the barometer instruction is executed is usually proportional to its running time