COMP20010

Data Structures and Algorithms I
05 - Algorithm Analysis I

Dr. Aonghus Lawlor
aonghus.lawlor@ucd.ie

Formalities

Assessment

- Continuous Assessment 30%
  - 2 Assignments (2x12%=24%)
  - End of module code repository (6%)

- End of Semester Exam: 70%

Course Timetable COMP20010

Week	Topic	Assignment
7	Intro	
	Arrays, ADT's, Generics	
8	Linked Lists	
	Doubly Linked, Circularly Linked	A1
9	Analysis of Algorithms	
10	Recursion	
	Timing, Performance	A2
11	Stacks, Queues, Deques, Priority Queues	
12	Maps, hash functions	A3
	Review	

Algorithm Analysis

Outline

- Algorithm Analysis
- Big-Oh notation
- Relatives of Big-Oh
- Some examples

Overview

Experimental approaches to evaluating algorithms have several limitations:

- Algorithm must be implemented
- Limited Set of Inputs
- Data Set Selection Non-trivial
- Must use same hardware and software environments.
- No guarantee that the algorithm will work in the same way when "live".

In some application domains, this is not enough.

- For example, routing algorithms for Air Traffic Control Systems.

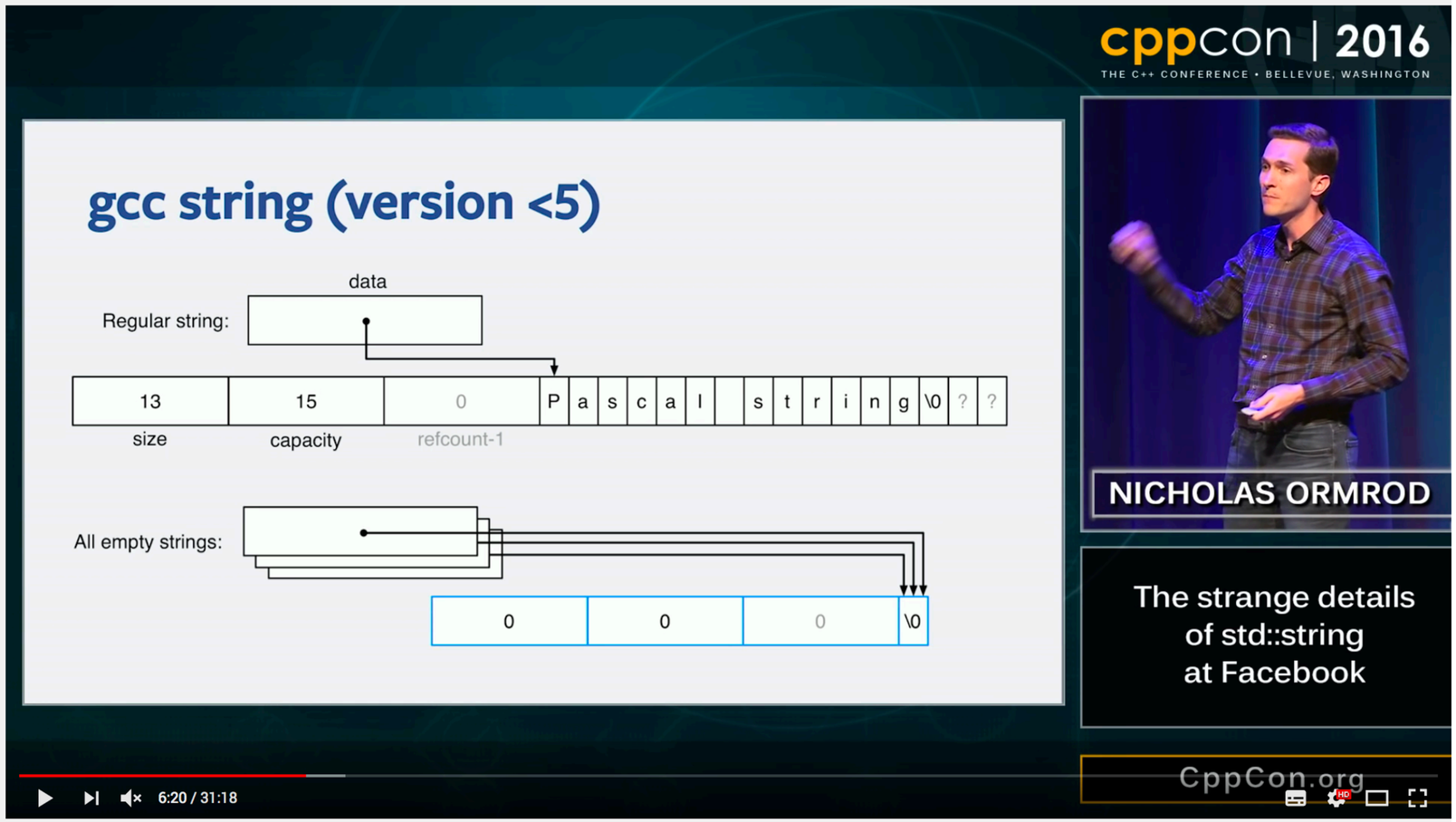
We need guarantees of what will happen in the worse case...

Overview

- For nontrivial problems to be solved (i.e. implemented) by computers, the efficiency of the solution is a crucial property
- One cannot simply develop a program (to solve a problem) disregarding its efficiency, and then hope the power of the hardware will do the rest
- on embedded systems, dedicated processors are not all that powerful, either
- Even trivial problems must be implemented carefully
- If a simple operation is executed 1e9 times, then a reduction in the execution time by half can result in enormous improvements.
- We need a way to “measure” the efficiency of a program, one that allows us to draw conclusions such as “program A is more efficient that program B” (provided programs A and B actually “do the same thing”)

Facebook C++ string optimisation

saving even 1% in production is a massive benefit at Facebook scale



CppCon 2016: Nicholas Ormrod “The strange details of std::string at Facebook”

CppCon Subscribe 12,308 views

Download Add to Share More 250 6

https://youtu.be/kPR8h4-qZdk

Overview

What language do we use to write programs?

- C, C++, Java, Python, Javascript, ...

Does the language really matter for complexity analysis?

- they say C/C++ is "faster" than Java, so is my program more efficient if I write it in C/C++ instead of Java?

Does the hardware on which I run the program matter?

- suppose I run the same program first on a Corei7, and then on an Xeon, and I notice that the first run takes 1000 times more than the second, does it mean that the efficiency of the program depends on the hardware on which I run it?

Overview

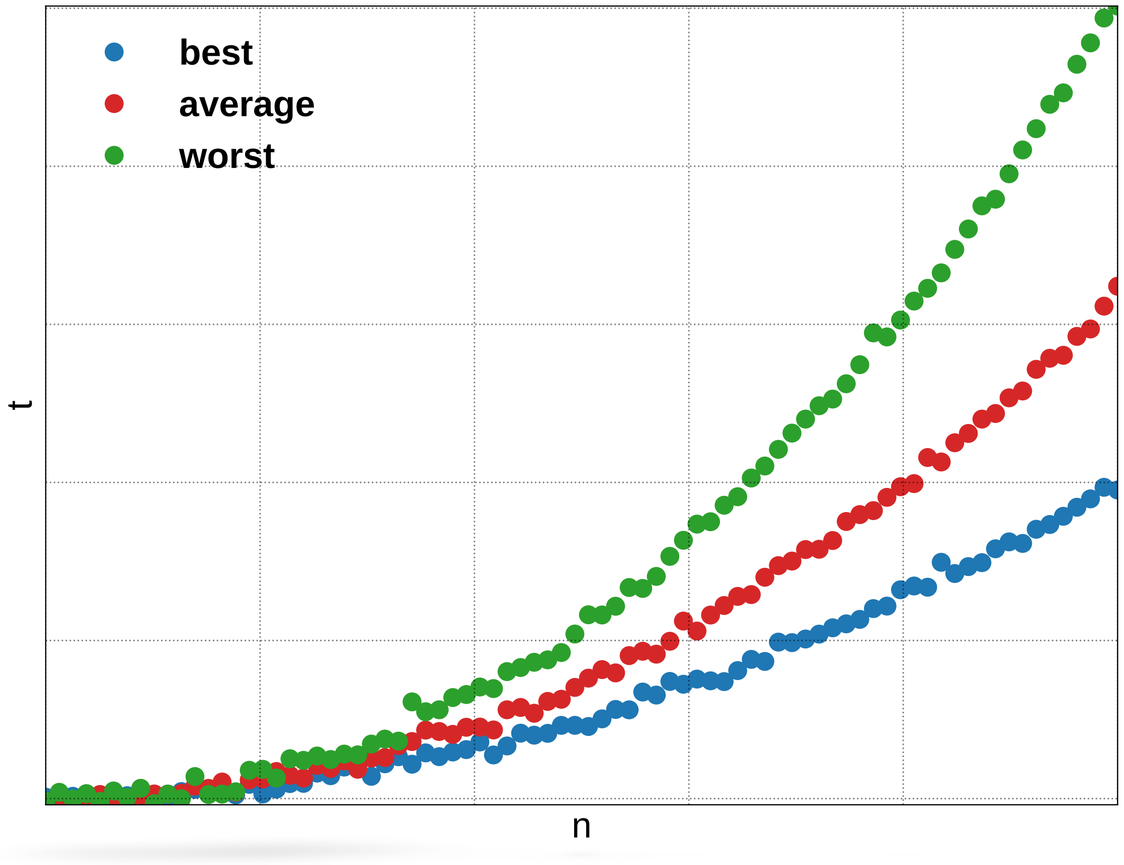
- Algorithms will be expressed in pseudocode

- Our analysis will attempt to abstract away the implementation related issues (such as the hardware on which the program runs)

Running time

input $\Rightarrow$ algorithm $\Rightarrow$ output

- 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.



Size of Input

The "size of the input data" can be different things for different problems

- it can be different things also for the same problem!
- in this case we might obtain different complexities depending on the quantity taken as “size of the input”

For sorting

- size = number of elements to sort

For sum, multiplication, etc.

- size = number of bits to encode the input numbers
- but we could also use as "size" the value of the numbers!

For matrix operations,

- size = number of elements in the matrix

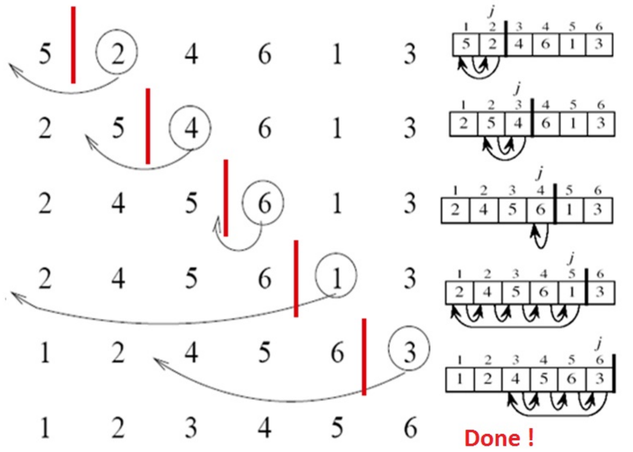
For graph operations,

- size = number of nodes and number of edges
- two dimensions, not only one!

Exercise

- take insertion_sort algorithm
- run the program for varying sizes
- use a built in method for determining the time to run the algorithm
- plot the results (execution time against array size)

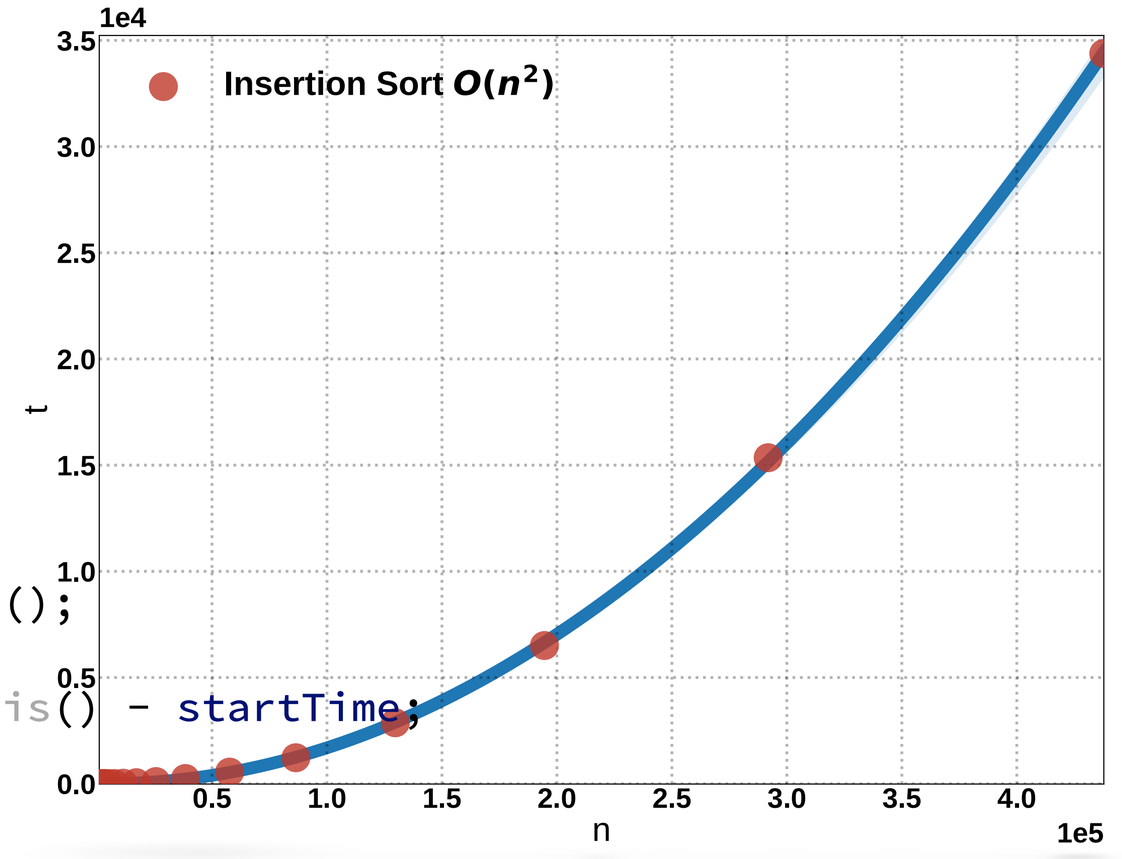
```
public void sort(int[] items) {
  for (int i = 1; i < items.length; i++) {
    int n = items[i];
    int pos = i;
    while (pos > 0 && items[pos - 1] > n) {
      items[pos] = items[--pos];
    }
    items[pos] = n;
  }
}
```



Exercise

```
public static void main(String args[]) {
  Random random = new Random(20010);

  InsertionSort ob = new InsertionSort();
  for(int n = 1000; n < 10000000; ) {
    int[] arr = new int[n];
    for(int i = 0; i < n; ++i) {
      arr[i] = random.nextInt();
    }
    final long startTime = System.currentTimeMillis();
    ob.sort(arr);
    final long elapsedTime = System.currentTimeMillis() - startTime;
    System.out.println("" + n + "," + elapsedTime);
    n = (int) (n * 1.5);
  }
}
```




```
InsertionSort [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_122.jdk/Content
1000,3
1500,3
2250,4
3375,12
5062,7
7593,9
11389,23
```

Limitations of run-time analysis

- 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

Theoretical Analysis

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

Use 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

```
1:  function ArrayMax(A, n)
2:      Input: array A of n integers
3:      Output: maximum element of A
4:      currentMax ← A[0]
5:      for i ← 1 to n do
6:          if A[i] > currentMax then
7:              currentMax ← A[i]
8:          end if
9:      end for
10: return currentMax
11: end function
```

Pseudocode - Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important
- Assumed to take a constant amount of time

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

Pseudocode - Analysis Rules

1. for loops

The running time of a for loop is at most the running time of the statements inside the for loop times the number of statements

```
for(int i = 0; i < n; ++i) {
  foo();
}
```

O(n)

Pseudocode - Analysis Rules

2. nested loops

Do the analysis inside out. The total running time of a statement inside a group of nested loops is the running time of the statement times the product of the sizes of all the loops:

```
for(int i = 0; i < n; ++i) {
  for(int j = 0; j < n; ++j) {
    foo();
  }
}
```

$O(n^2)$

Pseudocode - Analysis Rules

3. consecutive statements

these simple add- the maximum is the one that counts.

```
for(int i = 0; i < n; ++i) {
  a[i] += 1;
}
```

$O(n)$

```
for(int i = 0; i < n; ++i) {
  for(int j = 0; j < n; ++j) {
    a[i] += a[j];
  }
}
```

$O(n^2)$

total = $O(n) + O(n^2) = O(n^2)$

Pseudocode - Analysis Rules

4. if/else

the running time is never more than the running time of the test in addition to the larger of the branches

```
if(cond) {
  f();
} else {
  g();
}
```

total = O(cond) + max(O(f), O(g))

Using Pseudocode

```
1:  function ArrayMax(A, n)                          ▷ # operations
2:      Input: array A of n integers
3:      Output: maximum element of A
4:      currentMax ← A[0]                            ▷ 2          array access & assignment
5:      for i ← 1 to n do                            ▷ n − 1      loop handling
6:          if A[i] > currentMax then                ▷ 2(n − 1)   array access & comparison
7:              currentMax ← A[i]                    ▷ 2(n − 1)   assignment & array access
8:          end if
9:      end for
10: return currentMax                                ▷ 1          return
11: end function                                     ▷ total: 5n − 1
```

This function is O(n). The statements in the loop take a constant time each and are performed n times.

Estimating running time

Algorithm ***arrayMax*** executes 5***n***-1 primitive operations in the worst 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

$$a\,(5n - 1) \leq T(n) \leq b(5n - 1)$$

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

Growth Rate

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***

Growth Factors

The growth rate is not affected by constant factors or lower-order terms

Examples:

$$10^2 n + 43 \text{ (linear equation)}$$

$$10n^2 + 12n + 6 \text{ (quadratic equation)}$$

“Big Oh” Notation

Given functions f(n) and g(n), we say that

“ f(n) is big-Oh of g(n)”

“ f(n) is order of g(n)”

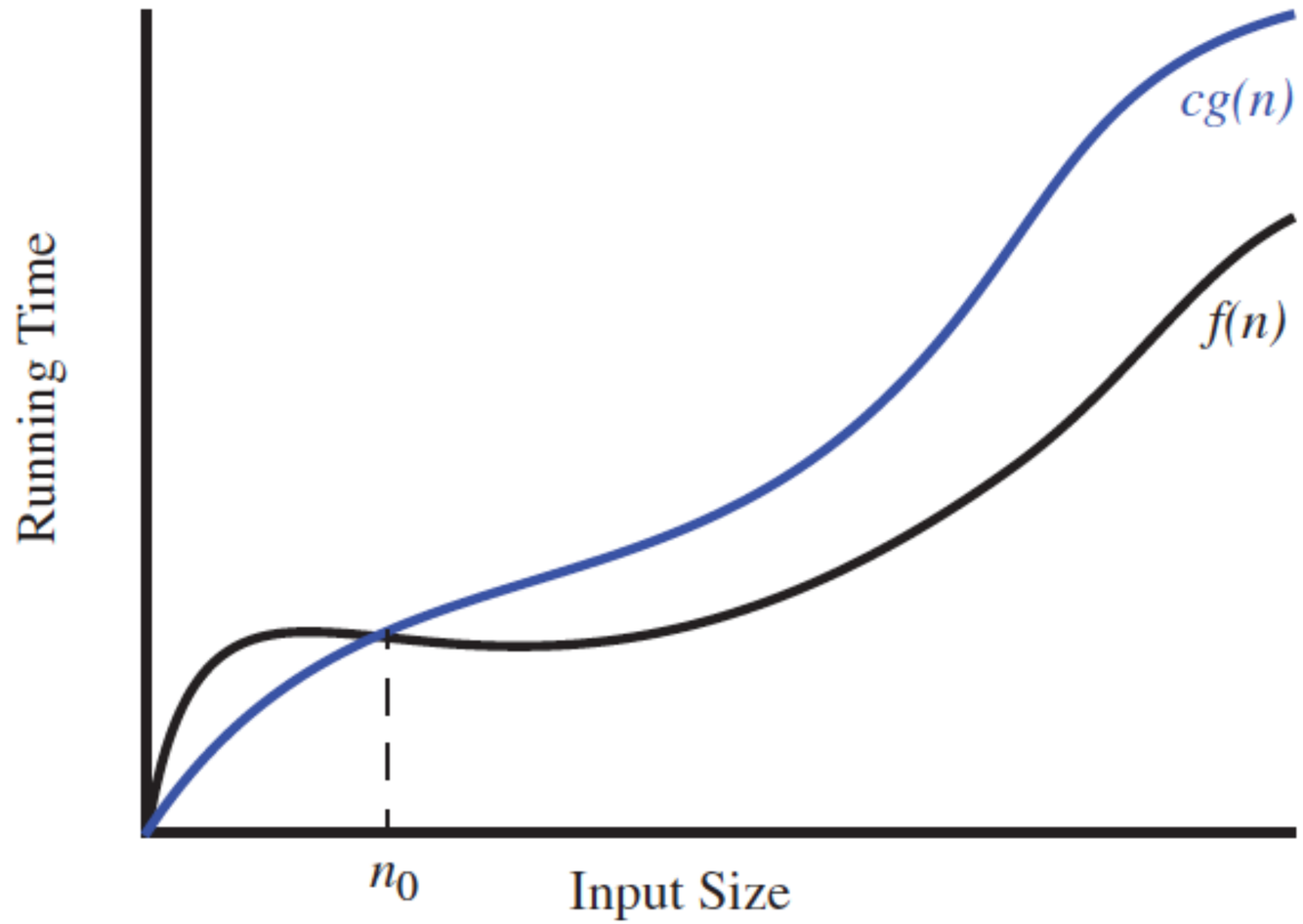
$$f(n) \; is \; O(g(n))$$

if and only if there are positive constants c and n_0 such that

$$f(n) \leq cg(n) \;\; \forall \;\; n \geq n_0$$

Big Oh Notation

$$f(n)\ is\ O(g(n))$$



“Big Oh” Example

Consider an algorithm with running time 11n + 5.

f(n) = 11n + 5

If we can show that g(n) = n satisfies the constraint:

$f(n) \leq c * g(n)$ for all $n \geq n_0$

Then, we can prove that 11n + 5 is O(n):

“Big Oh” Example

start with our simple example- we want to show 11n + 5 is O(n)

$$11n + 5$$

“Big Oh” Example

So, lets consider c = 12:

$$11n + 5 \leq 12n$$

“Big Oh” Example

rearrange

$$11n + 5 \leq 12n$$
$$5 \leq 12n - 11n$$

“Big Oh” Example

$$
\begin{aligned}
11n + 5 &\leq 12n \\
5 &\leq 12n - 11n \\
5 &\leq n
\end{aligned}
$$

$$f(n) \leq 12g(n) \quad \forall \; n \geq 5$$

“Big Oh” Example

- Indeed, this is one of infinitely many choices available because any real number greater than or equal to 12 works for c, and any integer greater than or equal to 5 works for n_0.

$$f(n) \leq 12g(n) \quad \forall \ n \geq 5$$

Big Oh Rules

when we choose g(n), we can choose any function that we want.

- If is $f(n)$ 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

Eg: we determine that algorithm ***arrayMax*** executes at most 5***n*** - 1 primitive operations- 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

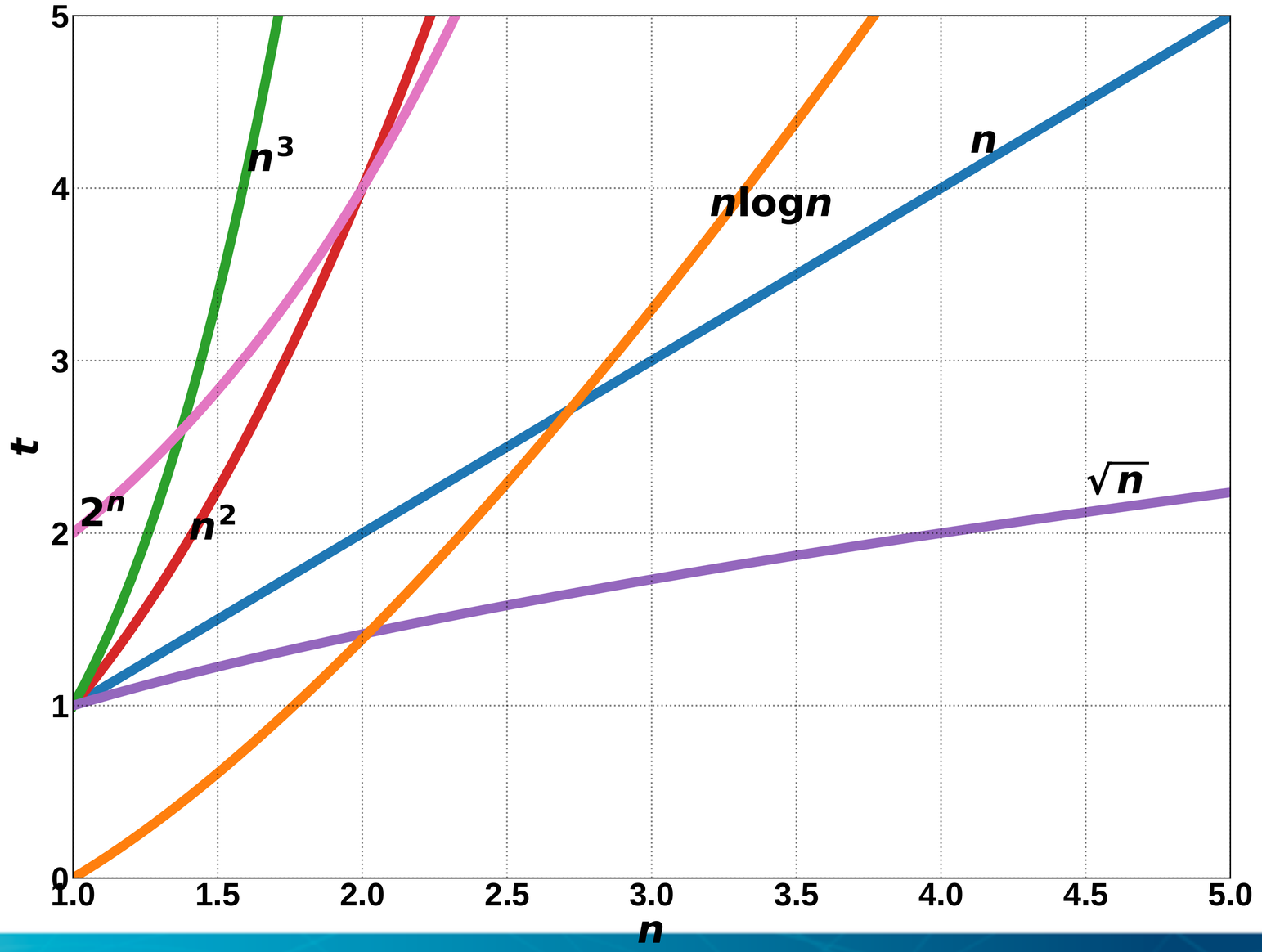
Common functions

An approximation is good if it is similar to the actual function, but does not include lower order terms.

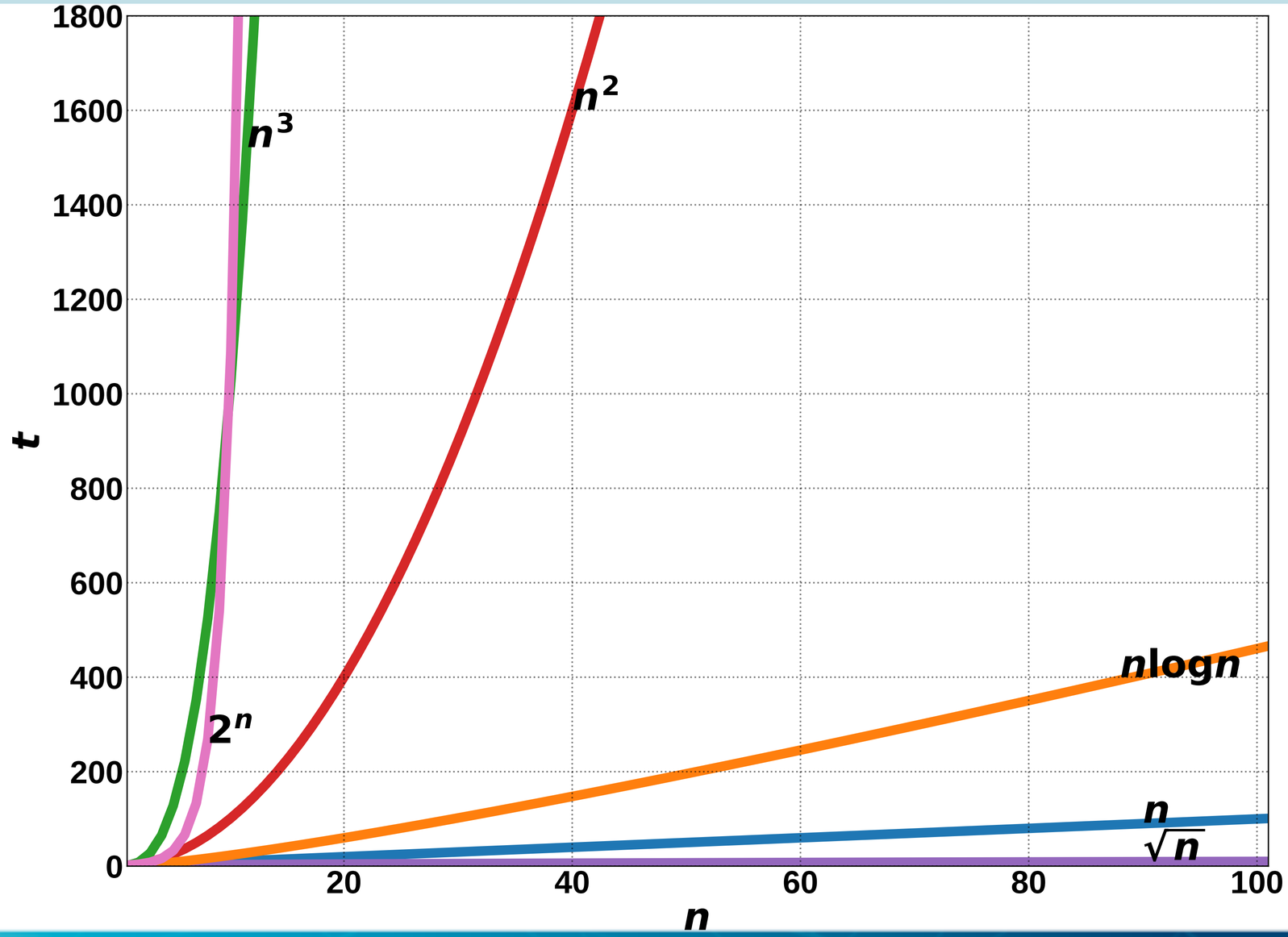
The hierarchy allows us to classify algorithms:

fixed:	$O(1)$
logarithmic:	$O(\log n)$
linear:	$O(n)$
n log n:	$O(n \log n)$
quadratic:	$O(n^2)$
polynomial:	$O(n^k), k \geq 1$
exponential:	$O(a^n), n > 1$
factorial:	$O(n!)$

Asymptotics (small values)



Asymptotics (large values)



Big Oh In Practice

Simple Rule: Drop lower order terms and constant factors:

$11n + 5$ is $O(n)$

$8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

Simpler Rule: Only examine loops / recursion

A loop over a fixed range [i-n] is typically $O(n)$

A loop within a loop is typically $O(n^2)$

When its not obvious, use induction

Caution! Beware of very large constant factors.

An algorithm with running-time $1{,}000{,}000n$ is still $O(n)$ but might be less efficient on your data set than one with running-time $2n^2$, which is $O(n^2)$.

Remember "Big-Oh" only works for values of n greater than or equal to some constant value.

Examples

$$5n^2 + 3n\log n + 2n + 5$$

$$5n^2 + 3n\log n + 2n + 5 \leq (5+2+2+5)n^2$$

$$= cn^2$$

$$= O(n^2)$$

for $c = 15$ when $n \geq n_0 = 2$

(since $n\log n = 0$ for $n = 1$)

We could also say the function is $O(n^2 + n\log n + n)$ but we try to find the simplest representation

Examples

$$
\begin{aligned}
& 2^{n+2} \\
& 2^{n+2} \leq 2^n 2^2 \\
& \qquad = 4 \times 2^n \\
& \qquad = O(2^n) \qquad\qquad \text{for } c = 4 \text{ when } n \geq n_0 = 1
\end{aligned}
$$

Examples

Let us assume that `f()` has complexity $O(1)$

```
for (int i = 0; i < n; ++i) f();
```
$\longrightarrow O(n)$

```
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j) f();
```
$\longrightarrow O(n^2)$

```
for (int i = 0; i < n; ++i)
  for (int j = 0; j <= i; ++j) f();
```
$\longrightarrow O(n^2)$

```
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
    for (int k = 0; k < n; ++k) f();
```
$\longrightarrow O(n^3)$

```
for (int i = 0; i < m; ++i)
  for (int j = 0; j < n; ++j)
    for (int k = 0; k < p; ++k) f();
```
$\longrightarrow O(mnp)$

Relatives of Big-Oh

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)

little-oh: f(n) is o(g(n)) if f(n) is strictly less than g(n)

little-omega: f(n) is ω(g(n)) if f(n) is strictly greater than g(n)

Functions we will use

- The Constant Function
- The Logarithm Function
- The Linear Function
- The N-log-N Function
- The Quadratic Function
- The Cubic Function and Other Polynomials
- The Exponential Function

Big Oh Cheats

http://bigocheatsheet.com/

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Excellent Good Fair Bad Horrible

Big Oh Cheats

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

Excellent Good Fair Bad Horrible

Typical Timescales

Operation	Time 1/1,000,000,000 sec = 1 ns	1ns -> 1second
execute typical instruction	1 nanosec	heartbeat
fetch from L1 cache memory	0.5 nanosec	
branch misprediction	5 nanosec	yawn
fetch from L2 cache memory	7 nanosec	long yawn
Mutex lock/unlock	25 nanosec	make a coffee
fetch from main memory	100 nanosec	brushing teeth

https://gist.github.com/hellerbarde/2843375

Execution Times

consider an operation that take 1nanosecond (1e-9) to execute

	Time		
Function	$(n = 10^3)$	$(n = 10^4)$	$(n = 10^5)$
$\log_2 n$	10 ns	13.3 ns	16.6 ns
$\sqrt{n}$	31.6 ns	100 ns	316 ns
n	1 μs	10 μs	100 μs
$n \log_2 n$	10 μs	133 μs	1.7 ms
n^2	1 ms	100 ms	10 s
n^3	1 s	16.7 min	11.6 days
n^4	16.7 min	116 days	3171 yr
2^n	$3.4 \cdot 10^{284}$ yr	$6.3 \cdot 10^{2993}$ yr	$3.2 \cdot 10^{30086}$ yr

C++ Container Complexity

	Add	Remove	Get	Contains	Data Structure
ArrayList	O(1)	O(n)	O(1)	O(n)	Array
LinkedList	O(1)	O(1)	O(n)	O(n)	Linked List
CopyonWriteArrayList	O(n)	O(n)	O(1)	O(n)	Array

What’s next

- More on Algorithm Analysis
- Linked lists in Tutorial Session