ITS64304 Theory of Computation

School of Computer Science
Taylor's University Lakeside Campus

Lecture 8: Complexity

Dr Raja..

Learning outcomes

At the end of this topic students should be able to:

- Explain the term algorithmic complexity
- Recognise complexity classes in terms of big-O notation
- Predict computation times based on complexity class and measurements
- *Classify problems as tractable or intractable based on complexity class**

* Module Learning Outcome 3

Complexity

- Complexity - **Cost of Computation.**

- How can we measure the complexity of a program?
  - Running time
  - Memory

Problem 1:

Design a computer program that for any input word , outputs 1 if the word is of the length $4n$ where n = 1, 2, 3, ... and outputs 0, otherwise.

- Problem 1 computationally solvable in principle
  - may not be solvable in practice
    - Large amount of time or resources.

Complexity

- Complexity - Cost of Computation.
- Complexity usually measured in terms of how the amount of a **resource** (time or space) required to run a program increases as the size of the program input increases.
- Most common measure of complexity is the amount of time the program takes to run – Time Complexity!

Tools of complexity analysis:

- should be independent of implementation (operating system, programming language, and processor speed)
- should not limit available memory or time
- must allow for all possible algorithms

Time Complexity

- What statistic should we use to measure time usage?
  - Minimum or Best case?
    - Best case may not always exist
  - Average?
    - An average (or even minimum) performance measurement may sometimes be more useful, but in general it makes the analysis significantly more difficult. (what is average? how often do "bad" cases occur? etc.)
  - Worst case?
    - Worst case analysis (i.e. we work out the maximum number of operations that could occur):
      - guarantees termination within a certain number of steps
      - provides an upper bound on the resources needed

Time Complexity

- Consider a (sorted) electronic phone directory.

  Aristotle

  Archimedes

  DaVinci

  Einstein

  Gallileo

  Newton

  Plato

  Russell

  Tesla

  Turing

Time Complexity

- Consider the following "pseudo-code" to find a supplied name:

```
i := 1;
while (name <> book[i] and i < n) do
        i := i + 1;
```

- This has complexity *n*
  - worst case it will take *n* comparisons to find an entry
    - i.e. the last one in the book

Time Complexity

- A better method is to use a binary search

```
left := 1; right := n;
found := false;
while (not found and left <= right) do
      i := (left + right)/2;
 if (name = book[i]) then found := true
 else if (name > book[i]) then
            left := i+1
      else right := i-1
```

- This has complexity $\log_2 n$

Time Complexity

Rates of Growth

- We measure the time complexity of a program by a function.

- The *rate of growth* of the function is usually what is most important.

n	0	10	100	1000
$20n + 500$	500	700	2,500	20,500
n^2	0	100	10,000	1000,000
$n^2 + 2n + 5$	5	125	10,205	1,002,005

- Note that the lower order terms have progressively less influence as *n* gets large.

Time Complexity

- A function f is of *order* g, written $f = O(g)$ if g is an approximation of the rate of growth of f for large n.

- For example:

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

1	-	1	+	2	+	5
20	-	400	+	40	+	5
40	-	1600	+	80	+	5
60	-	3600	+	120	+	5
80	-	6400	+	160	+	5
100	-	10000	+	200	+	5

$20n + 500$ is $O(?)$

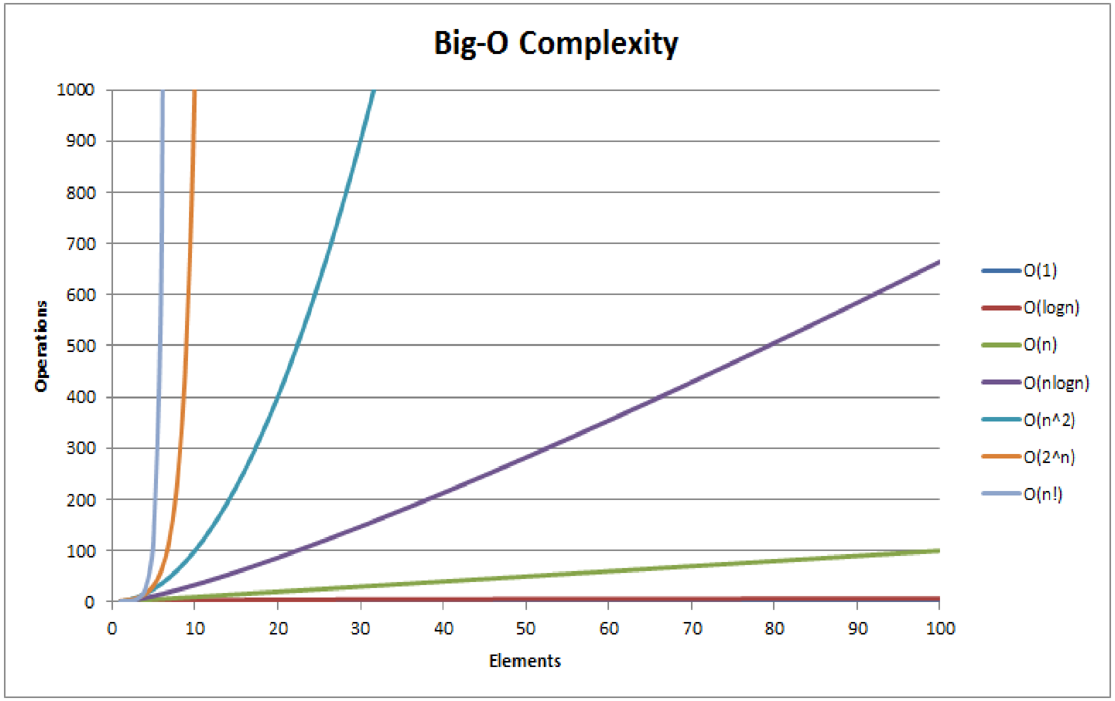
$n^3 + 500n^2 + 6$ is $O(?)$

$2^n + 30n^6$ is $O(?)$

Time Complexity

- To determine the order we always take the fastest growing term.

$O(1)$	= constant
$O(\log_a n)$	= logarithmic
$O(n)$	= linear
$O(n \log_a n)$	= "n log n"
$O(n^2)$	= quadratic
$O(n^3)$	= cubic
$O(n^r)$	= polynomial
$O(a^n)$	= exponential
$O(n!)$	= factorial



- Often anything larger than exponential is referred to simply as exponential

Time Complexity

n	*log n*	n^2	2^n	*n!*
5	2	25	32	120
10	3	100	1,024	3,628,800
20	4	400	1,048,576	$\sim 10^{18}$
30	4	900	$\sim 10^{9}$	$\sim 10^{32}$
40	5	1,600	$\sim 10^{12}$	$\sim 10^{48}$
50	5	2,500	$\sim 10^{15}$	$\sim 10^{64}$
100	6	10,000	$\sim 10^{30}$	$\sim 10^{161}$
200	7	40,000	$\sim 10^{60}$	$\sim 10^{374}$

Big-O time of programs

- To calculate the Big-O time requirements of a program...

1. First, assume the following can be done in O(1) time:
   - arithmetic operations (+ * / - % etc)
   - logical operations (&& etc)
   - comparison operations (<= etc)
   - structure-accessing operations (a[i] etc)
   - simple assignment (i = j etc)
   - calls to library functions (printf etc)

Big-O time of programs

2. Loop run time
   - taken as the number of times we go around the loop multiplied by the big-O upper bound of the loop body

3. Trivial O(1) times for initializing loop variables, and comparisons can be omitted (except where the loop body is empty)

Big-O time of programs

Example:

```
(1)      for(i = 0; i < n; i++)
(2)               for(k = 0; k < n; k++)
(3)                        A[i,k] = 0;
```

Line (3) takes $O(1)$ time

So lines (2)->(3) takes $O(n)$ time

So lines (1)->(3) takes $O(n^2)$ time

TEST YOURSELF

- What is the worst-case complexity of the each of the following code fragments?
  - Two loops in a row:

```
for (i = 0; i < N; i++)
{
   sequence of statements
}
for (j = 0; j < M; j++)
{
   sequence of statements
}
```

- How would the complexity change if the second loop went to N instead of M?

continued….

What is the worst-case complexity of the each of the following code fragments?

a) A nested loop followed by a non-nested loop:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        sequence of statements
    }
}
for (k = 0; k < N; k++) {
    sequence of statements
}
```

b) A nested loop in which the number of times the inner loop executes depends on the value of the outer loop index:

```
for (i = 0; i < N; i++) {
    for (j = i; j < N; j++) {
        sequence of statements
    }
}
```

Travelling Salesman problem

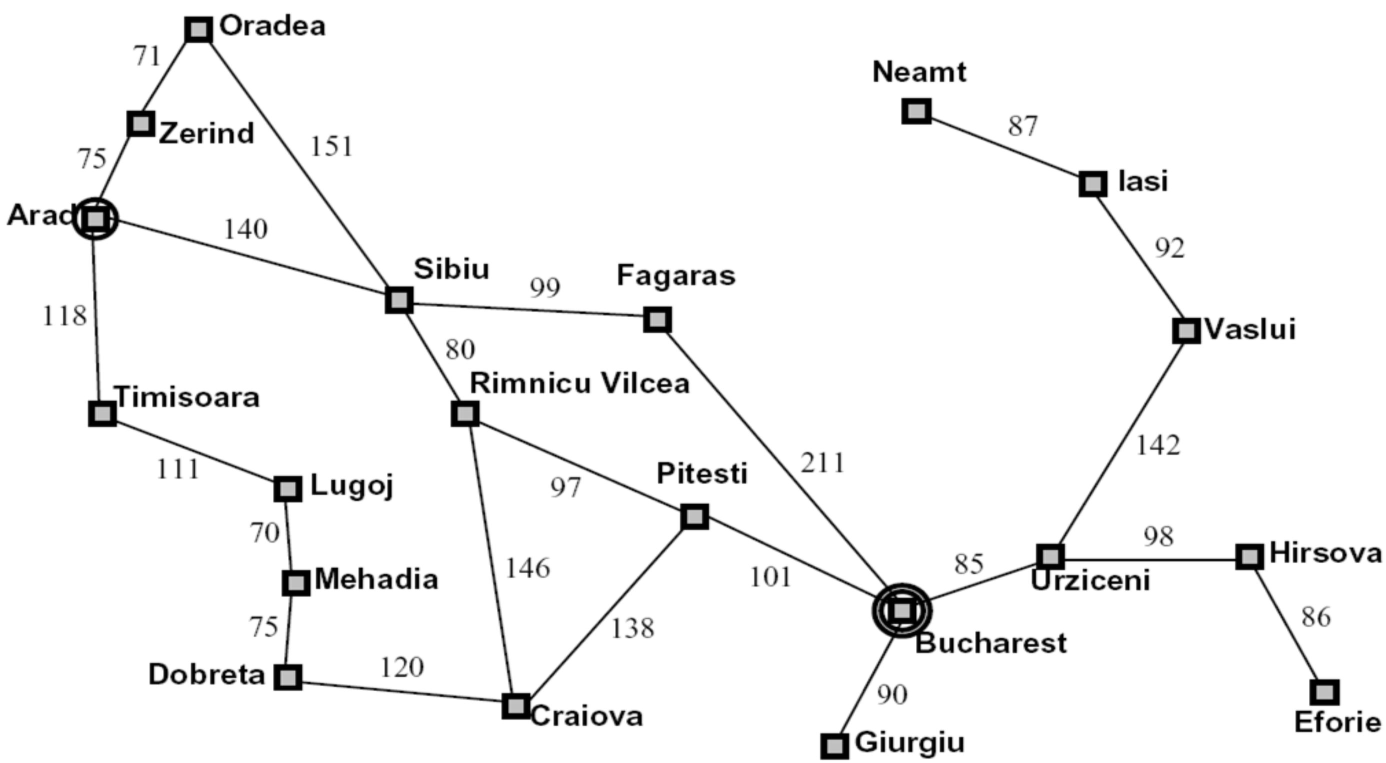
- Imagine a map showing a number of towns.
- A sales representative needs to find the shortest closed tour through each town.
  - i.e travels through each town exactly once
  - returns to starting point
  - entire distance traveled is the minimum possible

- Problem is an example of node traversal in a network of nodes
  - applications: telephone networks, integrated circuit design, planning etc

Example

In deciding to travel in Romania, say from Arad to Bucharest, we would look at different routes on a map before choosing one.

(How many possible routes are there?)



Travelling salesman problem

- One 'solution' algorithm
  - search every possible tour - report the one with minimum length
- With n nodes, there are $n!$ tours
- Running time $O(n!)$
- Recall that $n!$ grows faster than k^n for any constant k

Complexity Theory

- Complexity theory problems can be classified as follows.
  1. Unsolvable
  2. Solvable

     (a) Tractable - solvable in sensible time

     (b) Intractable - solvable, but requires so much time that practically they are too difficult to solve

- The complexity of a problem is measured by the <u>most efficient</u> solution to it.

Polynomial Time - Deterministic algorithms

- Problem solved within polynomial time $O(n^r)$ *for some r*
  - The problem is decidable in polynomial time
  - The class of all such problems is denoted as $\wp$.

- $\wp$ is the set of tractable decision problems.
- Generally, it is considered that any problem outside $\wp$ is intractable.
- Note: a decision problem is one whose return values are either YES or NO (or true or false, or 0 or 1).

Read for next week: Cryptography