Module #7:
Algorithmic Complexity

Rosen 5th ed., §2.3

~21 slides, ~1 lecture

What is *complexity*?

- The word *complexity* has a variety of technical meanings in different fields.
- There is a field of *complex systems*, which studies complicated, difficult-to-analyze *non-linear* and *chaotic* natural & artificial systems.
- Another concept: *Informational complexity*: the amount of *information* needed to completely describe an object. (An active research field.)
- We will study *algorithmic complexity*.

§2.2: Algorithmic Complexity

- The *algorithmic complexity* of a computation is some measure of how *difficult* it is to perform the computation.
- Measures some aspect of *cost* of computation (in a general sense of cost).
- Common complexity measures:
  - "Time" complexity: # of ops or steps required
  - "Space" complexity: # of memory bits req'd

Our focus

An aside...

- Another, increasingly important measure of complexity for computing is *energy complexity* - How much total energy is used in performing the computation.
- Motivations: Battery life, electricity cost...
- I develop *reversible* circuits & algorithms that recycle energy, trading off energy complexity for spacetime complexity.

Complexity Depends on Input

- Most algorithms have different complexities for inputs of different sizes. (*E.g.* searching a long list takes more time than searching a short one.)
- Therefore, complexity is usually expressed as a *function* of input length.
- This function usually gives the complexity for the *worst-case* input of any given length.

Complexity & Orders of Growth

- Suppose algorithm A has worst-case time complexity (w.c.t.c., or just *time*) $f(n)$ for inputs of length n, while algorithm B (for the same task) takes time $g(n)$.
- Suppose that $f \in \omega(g)$, also written $f \succ g$.
- Which algorithm will be *fastest* on all sufficiently-large, worst-case inputs?

B

Example 1: Max algorithm

- Problem: Find the *simplest form* of the *exact* order of growth (Θ) of the *worst-case* time complexity (w.c.t.c.) of the *max* algorithm, assuming that each line of code takes some constant time every time it is executed (with possibly different times for different lines of code).

Complexity analysis of *max*

procedure *max*($a_1, a_2, \ldots, a_n$: integers)

$v := a_1$	t_1	Times for *each* execution of each line.
for $i := 2$ **to** n	t_2	
if $a_i > v$ **then** $v := a_i$	t_3	
return v	t_4	

What's an expression for the *exact* total worst-case time? (Not its order of growth.)

Complexity analysis, cont.

procedure *max*($a_1, a_2, \ldots, a_n$: integers)

$v := a_1$ — t_1

for $i := 2$ **to** n — t_2

if $a_i > v$ **then** $v := a_i$ — t_3

return v — t_4

Times for *each* execution of each line.

w.c.t.c.:

$$t(n) = t_1 + \left(\sum_{i=2}^{n} (t_2 + t_3) \right) + t_4$$

Complexity analysis, *cont.*

Now, what is the simplest form of the exact (Θ) order of growth of $t(n)$?

$$t(n) = t_1 + \left(\sum_{i=2}^{n} (t_2 + t_3) \right) + t_4$$

$$= \Theta(1) + \left(\sum_{i=2}^{n} \Theta(1) \right) + \Theta(1) = \Theta(1) + (n-1)\Theta(1)$$

$$= \Theta(1) + \Theta(n)\Theta(1) = \Theta(1) + \Theta(n) = \Theta(n)$$

Example 2: Linear Search

procedure *linear search* (x: integer, a_1, a_2, …, a_n: distinct integers)

$i := 1$	t_1
while $(i \leq n \wedge x \neq a_i)$	t_2
$i := i + 1$	t_3
if $i \leq n$ **then** *location* $:= i$	t_4
else *location* $:= 0$	t_5
return *location*	t_6

Linear search analysis

- Worst case time complexity order:

$$t(n) = t_1 + \left(\sum_{i=1}^{n} (t_2 + t_3) \right) + t_4 + t_5 + t_6 = \Theta(n)$$

- Best case:

$$t(n) = t_1 + t_2 + t_4 + t_6 = \Theta(1)$$

- Average case, if item is present:

$$t(n) = t_1 + \left(\sum_{i=1}^{n/2} (t_2 + t_3) \right) + t_4 + t_5 + t_6 = \Theta(n)$$

Review §2.2: Complexity

- Algorithmic complexity = *cost* of computation.
- Focus on *time* complexity (space & energy are also important.)
- Characterize complexity as a function of input size: Worst-case, best-case, average-case.
- Use orders of growth notation to concisely summarize growth properties of complexity fns.

Example 3: Binary Search

procedure *binary search* (x:integer, $a_1, a_2, \ldots, a_n$: distinct integers)

$i := 1$
$j := n$
} $\Theta(1)$

while $i<j$ **begin**

$m := \lfloor (i+j)/2 \rfloor$
if $x>a_m$ **then** $i := m+1$ **else** $j := m$
} $\Theta(1)$

end

if $x = a_i$ **then** *location* $:= i$ **else** *location* $:= 0$
return *location*
} $\Theta(1)$

Key question:
How many loop iterations?

Binary search analysis

- Suppose $n=2^k$.
- Original range from $i=1$ to $j=n$ contains n elems.
- Each iteration: Size $j-i+1$ of range is cut in half.
- Loop terminates when size of range is $1=2^0$ ($i=j$).
- Therefore, number of iterations is $k = \log_2 n$ $= \Theta(\log_2 n) = \Theta(\log n)$
- Even for $n \neq 2^k$ (not an integral power of 2), time complexity is still $\Theta(\log_2 n) = \Theta(\log n)$.

Names for some orders of growth

• $\Theta(1)$	Constant	
• $\Theta(\log_c n)$	Logarithmic (same order $\forall c$)	
• $\Theta(\log^c n)$	Polylogarithmic	(With c a constant.)
• $\Theta(n)$	Linear	
• $\Theta(n^c)$	Polynomial	
• $\Theta(c^n)$, $c>1$	Exponential	
• $\Theta(n!)$	Factorial	

Problem Complexity

- The complexity of a computational *problem* or *task* is (the order of growth of) the complexity of the algorithm with the lowest order of growth of complexity for solving that problem or performing that task.
- *E.g.* the problem of searching an ordered list has *at most logarithmic* time complexity. (Complexity is O(log *n*).)

Tractable *vs.* intractable

- A problem or algorithm with at most polynomial time complexity is considered *tractable* (or *feasible*). **P** is the set of all tractable problems.
- A problem or algorithm that has more than polynomial complexity is considered *intractable* (or *infeasible*).
- Note that $n^{1,000,000}$ is *technically* tractable, but really impossible. $n^{\log \log \log n}$ is *technically* intractable, but easy. Such cases are rare though.

Unsolvable problems

- Turing discovered in the 1930's that there are problems unsolvable by *any* algorithm.
  - Or equivalently, there are undecidable yes/no questions, and uncomputable functions.
- Example: the *halting problem*.
  - Given an arbitrary algorithm and its input, will that algorithm eventually halt, or will it continue forever in an "*infinite loop*?"

P *vs*. **NP**

- **NP** is the set of problems for which there exists a tractable algorithm for *checking solutions* to see if they are correct.
- We know **P**⊆**NP**, but the most famous unproven conjecture in computer science is that this inclusion is *proper* (*i.e.*, that **P**⊂**NP** rather than **P**=**NP**).
- Whoever first proves it will be famous!

Computer Time Examples

#ops(n)	n=10 (1.25 bytes)	n=10^6 (125 kB)
$\log_2 n$	3.3 ns	19.9 ns
n	10 ns	1 ms
$n \log_2 n$	33 ns	19.9 ms
n^2	100 ns	16 m 40 s
2^n	1.024 μs	$10^{301,004.5}$ Gyr
n!	3.63 ms	Ouch!

Assume time = 1 ns (10^{-9} second) per op, problem size = n bits, #ops a function of n as shown.

Things to Know

- Definitions of algorithmic complexity, time complexity, worst-case complexity; names of orders of growth of complexity.
- How to analyze the worst case, best case, or average case order of growth of time complexity for simple algorithms.