Week 1: Introduction, Basic Concepts

Agenda:

- Course information
- Algorithms concepts
- Pseudo-code, RAM model

Textbook pages 4-26

Official course information

- Course web page:
  http://ugweb.cs.ualberta.ca/ c204/F07/
- Mark distribution:
  - 5 assignments 35% (7% each)
  - 2 term tests at 15% each (in class, 50 minutes)
  - Final 35%
- Lectures follow textbook: Introduction to Algorithms,
  by Cormen/Leiserson/Rivest/Stein (2nd Edition, 2001)
- Michael Buro (Section A1)
  - ATH337, mburo@cs.ualberta.ca
  - Office hours: Monday 13:30-14:00 and by appointment
- Mohammad Salavatipour (Section A2)
  - ATH303, mreza@cs.ualberta.ca
  - Office hours: Friday 14:00-15:00 and by appointment

Course overview

- 204 **Algorithm I**, Topics covered:
  - Introduction to algorithms
  - Algorithms: sorting, graphs,
  - Design: greedy, divide-and-conquer, dynamic programming
  - Analysis: model assumptions, worst/average/best case, asymptotic, correctness
- 304 **Algorithms II**
  - More advanced algorithms, and their design and analysis, complexity, notion of reduction, NP and NP-completeness
- 474 **Formal Languages, Automata and Computability**
  - More formal approach to models, complexity, and computability
- No seminars first week
- Assignment #1 available (from course web page)
- Attend your registered section and seminar
- Some of the materials covered in seminars

Motivation: Fibonacci

- Consider the sequence of Fibonacci numbers defined recursively:

  $$F(n) = \begin{cases} 1 & \text{for } n = 1 \\ 2 & \text{for } n = 2 \\ F(n-1) + F(n-2) & \text{if } n \geq 3 \end{cases}$$

n	1	2	3	4	5	6	7	8	9
$F(n)$	1	2	3	5	8	13	21	34	55

  Question: how do we compute $f(n)$?

- Direct and easy recursive implementation:

  procedure $fib1(n)$

  if $n < 3$ **then**
      return n
  else
      return $fib1(n-1) + fib1(n-2)$

- non-recursive implementation:

  procedure fib2(n)

  $F[1] \leftarrow 1$
  $F[2] \leftarrow 2$
  for $j \leftarrow 3$ **to** n **do**
      $F[j] \leftarrow F[j-1] + F[j-2]$
  return $F[n]$

Motivation: Fibonacci (cont'd)

- yet again another non-recursive implementation:

  procedure fib3(n)

  $x \leftarrow 1$
  $y \leftarrow 2$
  for $j \leftarrow 3$ **to** n **do**
  $\quad newy \leftarrow x + y$
  $\quad x \leftarrow y$
  $\quad y \leftarrow newy$
  return y

- Which of these is more efficient?

- Let $T1(n)$ be the number of recursive calls in $fib1(n)$.

- It follows $T1(n)$ is about $\left(\frac{1+\sqrt{5}}{2}\right)^n$

- For $n = 100$ this algorithm takes 10^8 days on a powerful desktop!!

- Both $fib2(n)$ and $fib3(n)$ take less than a second for n up to $100,000$.

- But $fib2$ needs a huge memory for huge values of n.

- So $fib3$ is both time and space efficient!

Basic concepts

- Problem: Given an input X satisfying certain properties, output Y which satisfies certain properties.

- Instance: A specific input for a problem is an instance

- Algorithm: A well-defined step-by-step procedure for solving a problem in a finite amount of time

- Issues for a given algorithm
  - Correctness — usually proved by "loop invariant" and induction
  - Analysis — measuring resource requirement
    * Running time
    * Space
  - Optimality

- Algorithm design concepts
  - Data structure: a systematic way to store and organize data for easy access and manipulation.
  - Some algorithm design techniques: greedy, Divide and conquer, Dynamic programming,
  - Some more (won't be covered): Exhaustive enumeration, Branch and bound, etc.

Methodologies for analyzing algorithms

- How do we measure if an algorithm is fast?
- Several factors involved: implementation language, compiler, operating system, the way it is implemented, test data, computer hardware (CPU, memory, disk, etc), and so on.
- The running time often increases as the input size increases
- So one way is to measure time in terms of input size
- Then run experiments on different input sizes
- Problem with experimental analysis:
  - We cannot run against all possible inputs
  - Even inputs of the same size may have different running time
  - Some factors (like CPU, memory, implementation, etc) can vary significantly; so test results are very dependent on them.
- So we need an analytic way of measuring the running time independent of environment factors (CPU speed, compiler, implementation, etc).
- Idea/Solution:
  - select theoretical computer model
  - estimate running time on model by the number of cycles

Model of computation: RAM

- RAM: random access machine
- Components
  - Input device
  - Output device
  - CPU: computation unit (inc. program)
  - M: memory locations (each can store an integer) M[0], M[1], M[2], ...
  - Program: fixed user-defined instruction sequence
- Properties
  - CPU has access to any mem location directly (by index)
  - move data between memory
  - compare data and branch
  - binary arithmetic operation
  - read from Input to memory
  - write from memory to Output
- primitive operations:
  - assign a value to a variable
  - calling a method
  - an arithmetic operation
  - comparing two basic variables
  - returning a value from a method
- Time — number of primitive (basic) instructions executed
- Space — number of memory locations used

Model of describing an algorithm

- Describing algorithms: `pseudocode`
- Pseudocode example

```
input:  integers a, b
output:  a × b

sum ← 0
for j ← 1 to b do
    sum ← sum + a
return sum
```

- Pseudocode conventions
  - indentation indicates block structure
  - `while/for/repeat/if/then/else`
  - Method: $name(pam1, pam2, \ldots)$
  - ** or $\triangleright$ comment
  - variables are local (unless stated otherwise)
  - parameters passed by value
  - comparison: boolean "short circuit" evaluation
    e.g. $j > 0$ `AND` $A[j] < key$
  - while-loop, repeat-loop, for-loop, if-then-else
  - array indexing: $A[i]$ for ith cell of array A.

An example: Insertion sort

- Input: a sequence of n numbers $(a_1, a_2, \ldots, a_n)$
- Output: a permutation $(a'_1, a'_2, \ldots, a'_n)$ such that $a'_1 \le a'_2 \le \ldots \le a'_n$
- Insertion sort — one sorting algorithm (we'll see more later)
  - Idea: repeatedly insert $A[j]$ into sorted sublist $A[1..j-1]$
  - How to insert?
    - search for insert location in sequence, $A[j-1]$, $A[j-2]$, ...
    - move contents to the right during the search
- Pseudocode

```
InsertionSort(A) **sort A[1..n] in place

for j ← 2 to n do
    key ← A[j] **insert A[j] into sorted sublist A[1..j − 1]
    i ← j − 1
    while (i > 0 and A[i] > key) do
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← key
```

- Insertion sort trace: $(53, 21, 47, 62, 14, 38)$

53	$\underline{21}$	47	62	14	38	** $j \leftarrow 2$
21	$\overline{53}$	47	62	14	38	** end of this iteration
21	53	$\underline{47}$	62	14	38	** $j \leftarrow 3$
21	47	$\overline{53}$	62	14	38	
21	47	53	$\underline{62}$	14	38	
21	47	53	$\overline{62}$	$\underline{14}$	38	
14	21	38	47	$\overline{53}$	62	** output permutation

Analysis of running time

- Model of computation: RAM
- Problem: run time varies with input
- Idea/Solution:
  - estimate worst/average/best case performance
  - make estimate a *function* of input *size*
  - sorting: input size usually takes as the number of keys

Kinds of analysis

- Worst case
  - $T(n)$ — maximum time over all inputs of size n
- Average case
  - Must specify input distribution over which average computed
  - Most common: assume uniform (a.k.a. equiprobable) input distribution (all inputs of size n equally likely)
  - Useful but usually difficult
- Best case
  - Useful for lower bound
  - Not otherwise useful: any algorithm can be modified to have fast best case (by adding: `if input is particular case then return particular answer`)

Analysis of insertion sort

`InsertionSort`(A)	*times*
for $j \leftarrow 2$ **to** n **do**	n
$\quad key \leftarrow A[j]$	$n-1$
$\quad i \leftarrow j-1$	$n-1$
$\quad$ **while** $(i > 0$ **and** $A[i] > key)$ **do**	$\sum_{j=2}^{n} t_j$
$\qquad A[i+1] \leftarrow A[i]$	$\sum_{j=2}^{n}(t_j - 1)$
$\qquad i \leftarrow i-1$	$\sum_{j=2}^{n}(t_j - 1)$
$\quad A[i+1] \leftarrow key$	$n-1$

t_j — number of times the `while` loop test is executed for j.

$$T(n) = n+2*(n-1)+3\sum_{j=2}^{n} t_j - 2*(n-1)*(-1)+n-1 = 2n-1+3\sum_{j=2}^{n} t_j$$

Running time

- Best case: list is already sorted ($t_j = 1$)

$$T(n) = 5n - 4$$

- Worst case: list is reverse sorted ($t_j = j$)

$$T(n) = 2n - 1 + 3\left[\frac{n(n+1)}{2} - 1\right] = 1.5n^2 + 3.5n - 4$$

  Why worst case analysis?
  - upper bound — guarantee
  - occurs fairly often
  - average case roughly as bad as worst case