CMPUT204: Introduction to Algorithms

Agenda:

- Course Information
- Algorithms concepts
- Recursion and Induction
- Pseudo-code, RAM model
- Getting Started: InsertionSort

Course Information

- Lectures (A1): MWF, 10:00-10:50 (CSC B2)
- Seminars (starting from the second week of the term)
  - Mondays 17:00 - 17:50 (CSC B2)
  - Tuesdays 12:30 - 12:20 (CSC B2)
  - Wednesdays 11:00-11:50 (V 103)
- Lectures follow textbook: Introduction to Algorithms, by Cormen, Leiserson, Rivest and Stein (3rd Edition)
- Prerequisites:
  CMPUT 115 or 175; CMPUT 272; MATH 113, 114, or 117 or SCI 100.

Teaching Staff

- Prof Jia You (B2) (CCid: jyou), office hours: ATH 3-54, Mondays and Fridays after class, 11:00-12:00pm and by appointment. Email headline begin with: [CMPUT204]

- Details on TAs' offices and consulting hours will be posted in eClass.

Course Work and Evaluation

- 6 quizzes 4.5% each in Seminars (6 X 4.5 $=27\%$)
  - Exercises posted on a Friday or before
  - Problems randomly selected for a quiz after the following Friday
  - A quiz takes 30 minutes
  - Dates of seminars having a quiz are posted in eClass
- 2 term tests 16% and 17% in class, Oct 16 and Nov 18
- Final 40% (3 hrs.); Current date: Dec 17, 2019
- All quizzes, tests and exam are closed book
- Grades: Based on a combination of absolute achievement and relative performance in the class. Generally, you need at least 50% to pass and at least 90% for A+, ...

Theory Courses @ UofA

- 204 **Algorithm I**
  - Introduction to algorithms
  - Basic algorithm design and analysis principles
- 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

The Study of Algorithms

- This course separates "programmers" from "algorithm designer":
- A programmer knows how to write code
- An algorithm designer knows how to reason about code
  - Argue about correctness and resources *mathematically*
  - Has wide-range of "tools" in her belt for a wide-variety of problems
- Leaving this course you should have learned:
  - Algorithms: sorting, graphs, math-operations, problem solving, ...
  - Design Paradigms: greedy, divide-and-conquer, dynamic programming
  - Analysis: model assumptions, correctness, worst/average/best case, asymptotic

What's an Algorithm?

- Problem: Given an input X satisfying... output Y satisfying...
- Instance: A specific input for a problem is an *instance*
- Algorithm: A well-defined step-by-step procedure for $X \to Y$
- Examples (that you already know!):
  - Given a natural number X, output Y is $X!$
  - Given a sequence of numbers (a1, ..., an), output a sorted list.
- What do we need to argue about an algorithm?
  - Correctness
  - Amount of resources: time and space
  - Can we do any better?
- For *any* instance? For a *good* instance? For an *average* instance?

Correctness for Recursive Codes

- The factorial function: $n! = \prod_{i=1}^{n} i$ (with $0! = 1$)
- Recursive implementation:

```
procedure factorial(n)
if (n = 0) then
    return 1                        ** Base case
else
    return n × factorial(n − 1)     ** Recursive call
```

- How do we prove the correctness of recursive code?
- It is simple to argue that `Factorial(0)` returns the correct answer.
- Based on that, we can argue that `Factorial(1)` returns the right answer.
- Based on that, we can argue that `Factorial(2)` returns the right answer.

  ...
- We need a proof that starts at a simple (base) case, and progresses from there until it covers all integers...
- I.e., we prove correctness of recursions using induction!

Induction

- Claim: For every natural number n, $\texttt{factorial}(n) = n!$.
- Proof: By induction.
  - (Base case) We show the claim holds for some initial value.
    - For $n = 0$ we have that $0! = 1$ and $\texttt{factorial}(0) = 1$.
  - (Induction step) Fix some k. Assuming the claim holds for k, we show the claim also holds for $k + 1$.
    - Assuming $\texttt{factorial}(k) = k!$, we have that

$$\begin{aligned}
\texttt{factorial}(k+1) &= (k+1) \times \texttt{factorial}(k) && **\text{ since } k+1 > 0 \\
&= (k+1) \times k! && **\text{by induction hypothesis} \\
&= (k+1) \times \prod_{i=1}^{k} i \\
&= \prod_{i=1}^{k+1} i = (k+1)!
\end{aligned}$$

$\blacksquare$

Induction

- Induction proof structure
  - **Base case:** We show the claim holds for some initial value. (Not necessarily 0)
  - **Induction step:** Fix some natural number k. Assuming the claim holds for k, we show that the claim also holds for $k+1$.
- Alternative: (Full/Complete/Strong induction)
  - **Induction step:** Fix some natural number k. Assuming the claim holds for all natural numbers $0 \leq i \leq k$ we show it also holds for $k+1$.

- Induction is a powerful and a commonly used tool.
- Must-haves in induction proofs:
  - A *clearly defined* base case
  - A *well-defined* assumption for any instance of a fixed size
  - A correct induction step — that indeed works for any k.
- Without these (and they are sometimes subtle) inductions can go horribly wrong.

Motivation: Fibonacci (Efficiency Issues)

- Consider the sequence of Fibonacci numbers defined recursively:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9	10	...
$F(n)$	0	1	1	2	3	5	8	13	21	34	55	...

- Problem: Given n, output $F(n)$.

Direct and easy recursive implementation:

```
procedure fib1(n)
if n < 2 then
   return n
else
   return fib1(n − 1) + fib1(n − 2)
```

Trace of recursive calls:

```
                fib1(4)
               /       \
         fib1(3)     fib1(2)
         /     \     /      \
    fib1(2)  fib1(1) fib1(1)  fib1(0)
   /     \
fib1(1)  fib1(0)
```

Some back-of-the-envelop calculations:

Let $T_1(n)$ denote the number of recursive calls in fib1(n).

- fib1$(n-1)$ — invoked 1 time
- fib1$(n-2)$ — invoked 2 times
- fib1$(n-3)$ — invoked 3 times
- fib1$(n-4)$ — invoked 5 times
- fib1$(n-5)$ — invoked 8 times
- Claim: fib1$(n-i)$ is invoked $F(i+1)$ times for any $1 \leq i \leq n-1$.
  Proof: induction!
- It follows $T_1(n) \geq \sum_{i=2}^{n} F(i)$ which is exponential in n (it is known that $T_1(n)$ is about $\left(\frac{1+\sqrt{5}}{2}\right)^n$, see pages 59-60 of Text).

- **Non-recursive implementation:**

```
procedure fib2(n)
F[1] = 0
F[2] = 1
for j = 3 to n + 1 do
    F[j] = F[j − 1] + F[j − 2]
return F[n + 1]
```

- **Yet another non-recursive implementation:**

```
procedure fib3(n)
if n = 0
    return 0
x = 0
y = 1
for j = 2 to n do
    newy = x + y
    x = y
    y = newy
return y
```

More calculations:

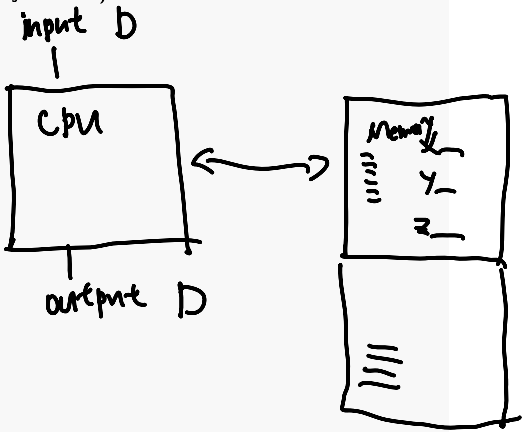
- Let $T_2(n)$ and $T_3(n)$ denote the number of times we invoke the loop in `fib2` and `fib3` respectively
$$T_2(n) = T_3(n) = n - 1, \text{ for all } n \geq 1$$
- Thus, $T_1(n)$ - exponential, $T_2(n), T_3(n)$ - linear
- What about space, measured by the number of "integers stored in memory"
  - `fib2` - linear; all Fibonacci numbers are stored in an array
  - `fib3` - small constant, x and y, $newy$, and a loop counter
- Conclusion: `fib3` is the best.

Methodologies for Analyzing Algorithms

- How do we measure an algorithm's running time (RT)?
- RT depends on hardware, software, implementation language, ...
- How about measuring RT in terms of input size?
- We cannot run against all possible inputs
- Even inputs of the same size may have different RTs.
- We need an analytic way of measuring RT independent of environment factors (CPU speed, compiler, implementation, ...).
- Idea/Solution:
  - Abstract away — select theoretical computer model
  - Try to identify "key operations": If each operation costs me $\$1$ dollar — how much will I end up paying

Model of computation: RAM

- RAM: random access machine (▸ You may watch this video)
- 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
  - variables are local (unless stated otherwise)
  - parameters passed by value
- primitive operations:
  - assign a value to a variable
  - calling a method/function/procedure
  - an arithmetic operation
  - comparing two basic variables
  - returning a value from a method



Model of computation: RAM (Cont'd)

- We will do our best to abstract away from all of those!
- Given an algorithm, we can measure:
  - Time — number of primitive (basic) instructions executed
  - Space — number of memory locations used

  In this course we will often look at Time, seldom Space.

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`
  - Procedure/Function: $name(pam1, pam2, \ldots)$
  - ** or ▷ comment
  - comparison: boolean "short circuit" evaluation
    e.g. $j > 0$ `AND` $A[j] < key$
  - array indexing: $A[i]$ for ith cell of array A.

A[i ..... n]

2 8 4 7 3 5

2 4 8 7 3 5

2 4 7 8 3 5

2 3 4 7 8 5

2 3 4 5 7 8

An example: Insertion Sort

- Input: n elements $(a_1, a_2, \ldots, a_n)$ where *each* pair is comparable (e.g.: numbers, cards, chess players, GDPs)
- Output: an ordered permutation $(a'_1, a'_2, \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- Our first solution: Insertion sort
  - Idea: repeatedly insert $A[j]$ into sorted sublist $A[1..j-1]$
  - How to insert? One by one, move elements in sorted sublist $A[1..j-1]$ which are bigger than $key (= A[j])$ to the right
- Pseudocode

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

`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`
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$

Array index

A: 1 2 3 4 5 6

2 4 7 8 3 5

2 4 7 8 8 5

2 4 7 7 8 5

2 3 4 7 8 5

item=3

An example: Insertion Sort Trace

- Input: : $(53, 21, 47, 62, 14, 38)$

53	21	47	62	14	38	** $j \leftarrow 2$, $key = 21$
53	53	47	62	14	38	
21	53	47	62	14	38	** end of this iteration
21	53	47	62	14	38	** $j \leftarrow 3$, $key = 47$
21	53	53	62	14	38	
21	47	53	62	14	38	
21	47	53	62	14	38	** $j \leftarrow 4$, $key = 62$
21	47	53	62	14	38	
21	47	53	62	14	38	** $j \leftarrow 5$, $key = 14$
21	47	53	62	62	38	
21	47	53	53	62	38	
21	47	47	53	62	38	
21	21	47	53	62	38	
14	21	47	53	62	38	
14	21	47	53	62	38	** $j \leftarrow 6$, $key = 38$
14	21	47	53	62	62	
14	21	47	53	53	62	
14	21	47	47	53	62	
14	21	38	47	53	62	** output permutation

Depends on input (eg. already sort)

depends on input size

~ want upper time

~ guarantee to the user

Analysis of running time

- Model of computation: RAM
- Problem: run time varies with input

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 distribution** (all inputs of size n equally likely)
- Best case
  - The least running time for any instance; useful only for lower bound.

Analysis of Insertion Sort

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

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

$\sum_{j=2}^{n} \theta(j) = \theta(n^2)$

$$T(n) = n + 2(n-1) + \left[3\sum_{j=2}^{n} t_j - 2(n-1)\right] + n - 1 = 2n - 1 + 3\sum_{j=2}^{n} t_j$$

Running time

- Best case: list is already sorted (for any $2 \leq j \leq n$ we have $t_j = 1$)
$$T(n) = 2n - 1 + 3(n-1) = 5n - 4$$
- Worst case: list is reverse sorted (for any $2 \leq j \leq n$ we have $t_j = j$)
$$T(n) = 2n - 1 + 3\left[\frac{n(n+1)}{2} - 1\right] = 1.5n^2 + 3.5n - 4$$
- Average case: Suppoe we randomly choose n numbers and apply insertionSort. On average, half of the elements in $A[1..j]$ are less than $A[j]$ and half are greater. So t_j is about $j/2$ and we end up with a quatratic function, as bad as the worst case.

Correctness of insertion sort

Theorem: $\texttt{InsertionSort}(A)$ returns A in a sorted order.

To prove the theorem we will prove the following claim.

- Claim: For any $2 \leq j \leq n$, at the start of the j-iteration $(2 \leq j \leq n)$ of the `for` loop, the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, and in sorted order.

This Claim is an example of a *Loop Invariant*, which we will study next.