Toolbox

Before we begin we need some tools. This section will mostly be refreshing concepts taught in CMPUT 174/5, 201, 204, 274/5, and 272, which hopefully intersect with courses you took already. If something looks unfamiliar we encourage you to read the “refresher” documents linked from the course web page. They go into more detail. Starting next week we will assume that your toolbox is fully equipped. The required tools are:

- Pseudo code
- Generating executables from pseudo code
- Math notation, big-O notation
- Induction proofs
- Fundamental data structures
- Program verification with loop invariants
- Basic program runtime analysis
- Graphs and common graph data structures

Pseudo Code ("soo-doh-kohd")

"a notation resembling a simplified programming language, used in program design." [Google dictionary]

```
// input:  integers x and y >= 0 (this is a comment)
// output: x+y
function Add(x, y)
  sum ← x              // assignment
  while y > 0 do
    decrement y        // same as y ← y-1
    increment sum      // same as sum ← sum+1
  done
  return sum
end
```

- Algorithm description independent of particular language
- Can be very compact and easily readable
- Relaxed syntax requirements (e.g., missing ;'s and most types)
- Must contain enough details to allow creating executable code

Generating Executables from Pseudo Code (1)

Once pseudo code is deemed correct, translating it into a programming language of choice step by step should be easy.

E.g., compiling the following C++ file Add.C on Linux with `g++ Add.C` creates executable `a.out`, which can be started by issuing `./a.out`

```
// testing Add function
#include <iostream>
using namespace std;

// input:  integers x and y >= 0
// output: x+y
int Add(int x, int y)
{
  int sum = x;
  while (y > 0) {
    y = y-1;
    sum = sum+1;
  }
  return sum;
}
```

```
int main()
{
  // call Add and
  // print results
  cout << Add(10, 20)
       << Add(5, 0)
       << endl;

  return 0;
}
```

Generating Executables from Pseudo Code (2)

Some assignments will ask you to implement algorithms for which pseudo code was presented in the lectures.

In CMPUT 396 we will accept submissions in C++, Java, and Python-3

If you are not familiar with C++, we suggest to learn it while working on assignments for this course. The reason is that search based methods need to be fast. C++ code is, but Python isn't. Java code is somewhere in between.

Good C++ tutorials are freely available (also see my "refreshers")

You can develop programs on your own computers, but all submissions have to work on the Linux lab machines in CSC 153/159.

Detailed submission procedures will be published alongside the assignments.

Math Notation (1)

Throughout the course we'll use the following math notation:

- Set: collection of unique items
  E.g., $S = \{1, 2, 3, 4, 5\}$ contains five small integers, their order doesn't matter, elements appear exactly once
- Set cardinality: number of elements in set, denoted $|S|$
- Set membership: $x \in S$ is true if and only if x is an element of S.
  $x \notin S$ is true if and only if $x \in S$ is false
- Set operations: $S \cup T$ (union), $S \cap T$ (intersection)
- Set inclusion: $S \subseteq T$ (all elements in S are also in T)
- Tuple/list: sequence of items
  E.g., $L = (1, 2, 3, 3, 5)$, contains five small integers, their order matters, elements can appear multiple times
- Cross Product: $S \times T = \{(s, t) \mid s \in S \text{ and } t \in T\}$ (all pairs)

Math Notation (2)

Special sets:

- $\emptyset = \{\}$: empty set (no elements)
- $\mathbb{N} = \{0, 1, 2, 3, ...\}$: natural numbers
- $\mathbb{Z} = \{0, 1, -1, 2, -2, ...\}$: integers
- $\mathbb{Q} = \{\frac{p}{q} \mid p, q \in \mathbb{Z} \text{ and } q > 0\}$: rational numbers (e.g., $3.5 \in \mathbb{Q}$, but $3.5 \notin \mathbb{Z}$)
- $\mathbb{R}$: real numbers (e.g., $\pi \in \mathbb{R}$, but $\pi \notin \mathbb{Q}$)

Function:

- Assigns at most one range element to each domain element
- $f : X \to Y$ maps elements of X (domain) to elements of Y (range)
- $f(x) = y$ ("f of x equals y"): f maps $x \in X$ to $y \in Y$
- E.g., $g : \mathbb{N} \to \mathbb{N}$, defined by $g(n) = 2n$ for all $n \in \mathbb{N}$
  I.e., $g(0) = 0, g(1) = 2, g(10) = 20$, etc.

Math Notation (3)

Relation: set of tuples $(x_1, x_2, ..., x_n)$ of related items

Example 1

$R_= = \{(x, y) \mid x, y \in \mathbb{N}, x \text{ equals } y\} = \{(0, 0), (1, 1), (2, 2), ...\}$

We write $x = y$ as a shorthand for $(x, y) \in R_=$

Thus, $x = y$ is either true or false, depending on whether x equals y

$x \neq y, x < y, x > y, x \geq y, ...$ can be defined analogously

Example 2

$R_F = \{(a, b) \mid a \text{ is a friend of } b\} = \{(\text{Mike,Kim}), (\text{Bill,Joe}), ...\}$

Math Notation (4)

Boolean Values and Operators:

- Boolean constants: 0 : "false, 1 : "true"
- Boolean variables: $x, y, ... \in \{0, 1\}$
- $\neg x, \bar{x}$: negation ($\neg 0 = \bar{0} = 1, \neg 1 = \bar{1} = 0$)
- $x \wedge y$: and (result 1 if $x = 1$ and $y = 1$, 0 otherwise)
- $x \vee y$: or (result 0 if $x = 0$ and $y = 0$, 1 otherwise)
- $x \Leftrightarrow y$: if-and-only-if (result 1 if $x = y$, 0 otherwise)
- $x \Rightarrow y$: implication (result 0 if $x = 1$ and $y = 0$, 1 otherwise)

Boolean formulas are built from these primitives and ( ) like regular math formulas involving numbers

E.g., $(x_1 \wedge \bar{x}_2) \vee (x_1 \Leftrightarrow 1)$

What is the value of this formula for $x_1 = 0$ and $x_2 = 1$?

Big-O Notation

When comparing runtime functions we are only interested in their growth rate which is the asymptotic behaviour for large input sizes n.

To describe the asymptotic growth of functions we will use "big-O" notation, which allows us to concentrate on the leading term because lower order terms become insignificant for large n, and also skip irrelevant coefficients.

Example:

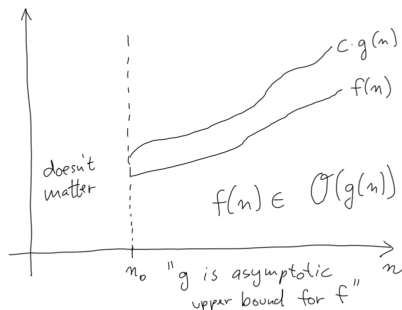
Suppose for input size n our algorithm takes $t(n) = 3n^3 + 5n^2 + 5$ computation steps in the worst case.
We can then use the following expressions to describe the asymptotic growth rate of t:

- $t(n) \in O(n^3)$: n^3 is an asymptotic upper bound for $t(n)$
- $t(n) \in \Omega(n^3)$: n^3 is an asymptotic lower bound for $t(n)$
- $t(n) \in \Theta(n^3)$: n^3 is the asymptotic growth rate of $t(n)$

Details: $O(g(n))$

Read as "big O of g of n"

- roughly: set of functions $f(n)$ which, as n gets large, grow no faster than a constant times $g(n)$
- precisely: set of functions $f : \mathbb{N} \to \mathbb{R}$ for which there are constants $c > 0$ and $n_0 \in \mathbb{N}$ with $|f(n)| \leq c\,|g(n)|$ for all $n \geq n_0$



Examples:

$n + 1 \in O(n)$

$3n + 5 \in O(n)$

$n^2 \notin O(n)$

$4n^2 + 3n + 5 \in O(n^2)$

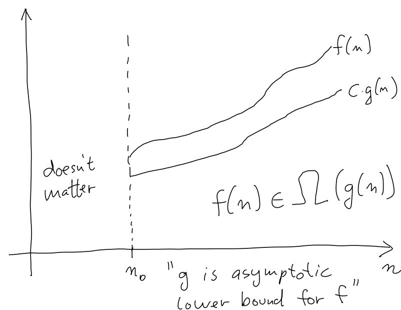
$\log(n) \notin O(1)$

$O(n) \subseteq O(n^2)$

Details: $\Omega(g(n))$

Read as "big omega of g of n"

- roughly: set of functions $f(n)$ that grow at least as fast as a constant times $g(n)$
- precisely: set of functions $f : \mathbb{N} \to \mathbb{R}$ for which there are constants $c > 0, n_0 \in \mathbb{N}$ such that $|f(n)| \geq c|g(n)|$ for all $n \geq n_0$



Examples:

$n - 1 \in \Omega(n)$

$3n + 5 \in \Omega(n)$

$n^2 \in \Omega(n)$

$4n^2 + 3n + 5 \in \Omega(n^2)$

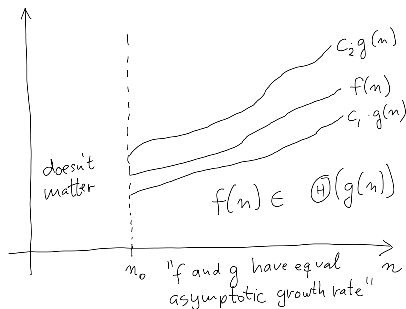
$\log(n) \in \Omega(1)$

$\Omega(n^2) \subseteq \Omega(n)$

Details: $\Theta(g(n))$

Read as "big theta of g of n"

- roughly: set of functions $f(n)$ that grow with the same rate as $g(n)$
- precisely: set of functions $f : \mathbb{N} \to \mathbb{R}$ for which there are constants $c_1 > 0, c_2 > 0, n_0 \in \mathbb{N}$ such that $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$



Examples:

$n - 1 \in \Theta(n)$

$3n + 5 \in \Theta(n)$

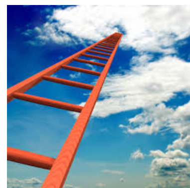
$n^2 \notin \Theta(n)$

$4n^2 + 3n + 5 \in \Theta(n^2)$

$\log(n) \notin \Theta(1)$

$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

Induction Proofs



Mathematical induction proves that we can climb as high as we like on a ladder, by proving that we can climb onto the bottom rung (the basis) and that from each rung we can climb up to the next one (the induction).
— Concrete Mathematics, page 3 margins

This is a powerful proof technique which is widely used in CS

(E.g., for proving algorithm correctness and analysing runtimes)

Formally: if we want to prove that some property about a natural number — say $P(n)$ — is true for all $n \in \mathbb{N}$, we can proceed as follows:

1. Prove that $P(0)$ is true
2. Prove that for arbitrary $n \in \mathbb{N}$: $P(n+1)$ is true if $P(n)$ is true

$\Rightarrow$ $P(n)$ is true for all $n \in \mathbb{N}$

Example

Claim: $\underbrace{\sum_{i=0}^{n} i = n(n+1)/2}_{P(n)}$ is true for all $n \in \mathbb{N}$.

Examples:

$P(0) : 0 = 0(0+1)/2$? correct

$P(1) : 1 = 1(1+1)/2$? correct

$P(4) : 1+2+3+4 = 4(4+1)/2$? correct

$P(10) : 1+2+3+4+5+6+7+8+9+10 = 10(10+1)/2$? correct

We suspect that $P(n)$ is true for all $n \in \mathbb{N}$

Induction Proof

Claim: $\underbrace{\sum_{i=0}^{n} i = n(n+1)/2}_{P(n)}$ is true for all $n \in \mathbb{N}$.

Induction Base: $P(0)$ is true (see previous slide)

Induction Step:

Suppose $P(n)$ is true for an arbitrary $n \in \mathbb{N}$ (Induction Hypothesis (I.H.))

$P(n+1)$ means: $\sum_{i=0}^{n+1} i = (n+1)((n+1)+1)/2 \qquad (*)$
(replaced n by $n+1$ in $P(n)$ definition)

$\sum_{i=0}^{n+1} i = (\sum_{i=0}^{n} i) + (n+1) \overset{\text{I.H.}}{=} n(n+1)/2 + (n+1) = (n+1)(n/2+1)$

which equals the right-hand side of $(*)$

This means $P(n+1)$ is true. Thus, $P(n)$ holds for all $n \in \mathbb{N}$ $\square$

Fundamental Data Structures

- Array

  Items stored in consecutive memory locations, fixed size, direct item access by index usually starting with 0

- Vector

  Similar to arrays, but size can change

- Linked List

  Sequence of items with links to successor/predecessor, fast insertion/deletion, no direct access

- Set

  Set of items, supports item insertion/deletion, membership test

- Map/Dictionary

  Stores (x, y) pairs, where x is mapped to exactly one y

C++ Examples

```
// using STL (Standard Template Library)
#include <vector>
#include <set>
#include <map>
using namespace std;

int main() {
  int A[20];  // integer array of 20 elements
  A[0] = 1;   // set first element to 1

  vector<char> v;    // empty character vector
  v.push_back('a'); // add one character at the end

  set<double> s;     // empty set of double values
  s.insert(3.0);     // add 3.0 to set

  map<int,string> m;   // empty map of int to string
  m.insert({3,"foo"}); // map 3 to "foo"
  return 0;
}
```

Program Verification with Loop Invariants (1)

Program verification involves proving formally that 1) the program terminates on all relevant inputs, and 2) that in case the program terminates it has generated the correct output.

A popular tool for proving the correctness of programs containing loops is using a loop invariant, which is a predicate on the variables of the program, that is shown to be true at the beginning of the loop (Initialization) and maintained after executing the loop body (Maintenance).

Applying mathematical induction, loop invariants therefore hold after any number of loop body executions, and together with the loop exit condition can be used to prove the loop's correctness.

To illustrate the process, we consider an example: `selection-sort`

Program Verification with Loop Invariants (2)

```
function selection-sort(A[0..n-1])
  k ← 0
  while k < n do
    (*)
    i ← an index j in {k..n-1} for which A[j] is minimal
    swap A[k] and A[i]
    k ← k + 1 (**)
  done
end
```

Let A' be the original array passed on to the function.

Termination

The loop stops after exactly n iterations, because k counts from 0 to n.

Loop Invariant

A[0..k-1] contains the smallest k elements of A' in non-decreasing (sorted) order, and A is a permutation of A'.

Program Verification with Loop Invariants (3)

Initialization

Before the first loop body execution, $k = 0$ and A[0..-1] (an empty array) contains the smallest 0 elements of A' (trivially true), and A is a permutation of A' because A = A'.

Maintenance

Assume invariant holds at point (*) in the program execution, and show that it also holds after the loop body is executed (at point (**)).

After the inner loop is executed, i contains an index of a smallest element in A[k..$n-1$]. Swapping it with A[k] results in sorted array A[0..k] containing the smallest $k+1$ values of A', because A[0..$k-1$] was sorted and A[k..$n-1$] contained values bigger or equal to A[$k-1$].

Also, A was a permutation of A'. So, when swapping two elements, it still is. k is then incremented and the loop invariant holds once more.

Program Verification with Loop Invariants (4)

Output Correctness

Right after exiting the loop, we have $k = n$.

Plugging this value of k into the loop invariant shows that after exiting the loop, A[0..$n-1$] contains all elements of A' in sorted order

Basic Program Runtime Analysis (1)

After establishing a program's correctness the next important properties are its runtime and space requirements, as we usually prefer faster algorithms or those that need less memory.

To simplify runtime analysis we often assume the unit-cost measure, i.e., executing a single statement costs 1 time unit, irrespective of the size of the involved data items.

We then count the number of steps and relate it to the input size n using big-O notation.

Basic Program Runtime Analysis (2)

For example, what is the runtime of the following function?

```
 1 // input: array A containing N>0 integers
 2 // output: maximum element in A
 3 function find_max(A[N])
 4   max ← A[0]
 5   for i ← 1 to N-1 do
 6     if A[i] > max then
 7       max ← A[i]
 8     end
 9   done
10   return max
11 end
```

Lines 3..4,10 will be executed once	3 steps
Line 5 will be executed N times	N steps
Lines 6,8,9 will be executed $N-1$ times	$N-1$ steps
Line 7 is executed 0 up to $N-1$ times	$0..(N-1)$ steps
Total runtime	$(2N+2)..(3N+1)$ steps

Basic Program Runtime Analysis (3)

Bounds for $t(N)$, the worst case runtime for input size N:

$$t(N) \geq 2N + 2 \quad \Rightarrow \quad t(N) \in \Omega(N)$$

$$|t(N)| \leq 3N + 1 \quad \Rightarrow \quad t(N) \in O(N)$$

Therefore, $t(N) \in \Theta(N)$

Can we find an algorithm that finds the maximum element and runs asymptotically faster than $\Theta(N)$?

No! We need to visit each element at least once

So, `find_max` is asymptotically runtime optimal

Basic Program Runtime Analysis (4)

To establish the growth rate of a function that consists of a few additive terms we can often easily identify the largest growing term and remove minor terms and constant coefficients.

Examples:

$$t(n) = 3 + \lceil \log(n) \rceil + 1000\, n^2 + 5\, n^4 \quad \Rightarrow \quad t(n) \in \Theta(n^4)$$

$$t(n) = 5 + \lfloor n \log(n) \rfloor + 6\, n^2 \quad \Rightarrow \quad t(n) \in \Theta(n^2)$$

$$t(n) = n^5\, 2^n + 10 \cdot 3^n \quad \Rightarrow \quad t(n) \in \Theta(3^n)$$

$\lceil x \rceil$ = smallest integer $\geq x$ (round up, “ceiling”)
$\lfloor x \rfloor$ = biggest integer $\leq x$ (round down, “floor”)

Basic Program Runtime Analysis (5)

A common mistake:

To compare the asymptotic runtimes t_A and t_B of two algorithms A and B, O is insufficient because it describes upper bounds. E.g.:

$$t_A(n) \in O(n) \ \wedge \ t_B(n) \in O(n^2) \overset{?}{\Rightarrow} A \text{ asymptotically faster than } B$$

No! This is like saying: $x \leq 5$ and $y \leq 10$, therefore $x < y$, which is obviously wrong.

For instance, $t_A(n) = n$, and $t_B(n) = 1$, would satisfy the O-expressions, but the conclusion is wrong because our O-bound for t_B is way off.

The conclusion is valid if we find an upper bound for t_A and a lower bound for t_B. E.g.,

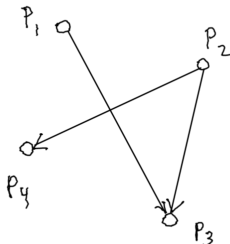
$$t_A \in O(n) \ \wedge \ t_B \in \Omega(n^2) \Rightarrow A \text{ asymptotically faster than } B$$

Graphs (1)

Graphs model binary relations $R \subseteq V \times V$

- Objects are represented by vertices
- Related objects are connected by edges

Example: Relation R with $(x, y) \in R \Leftrightarrow$ person x knows person y



P_1 knows P_3

P_2 knows P_4

P_2 knows P_3

Relations are not necessarily symmetric [$(x, y) \in R \Rightarrow (y, x) \in R$]

Therefore, we need directed edges ("arcs")

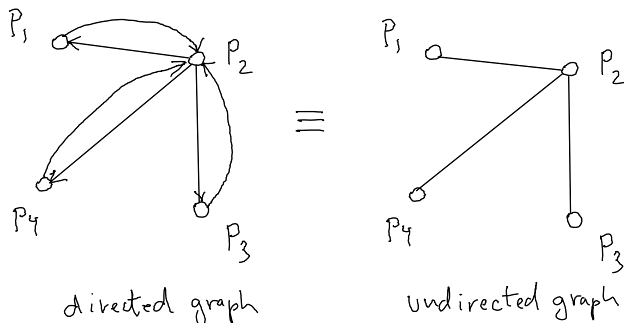
Graphs (2)

For symmetric relations undirected edges suffice

Example:

$$(x, y) \in F \;\Leftrightarrow\; x \text{ is a friend of } y$$

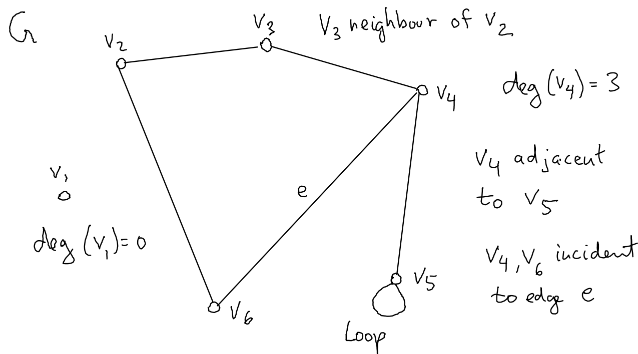
Then (usually) $(x, y) \in F \Leftrightarrow (y, x) \in F$



Graphs (3)

Definition:

An **(undirected) graph** is a pair $G = (V, E)$ that is composed of a non-empty vertex (or node) set V and an edge set E, such that each edge contains one or two vertices, i.e. for all $e \in E$, $e \subseteq V$ and $1 \leq |e| \leq 2$. The number of neighbours of v is called **degree** ($\deg(v)$).



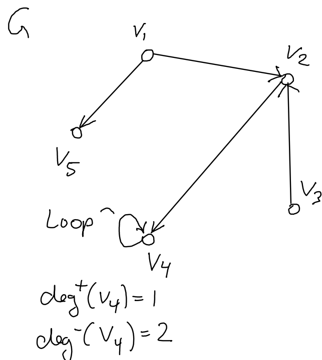
$$G = (V, E)$$

$$V = \{v_1 \dots v_6\}$$

$$E = \{ \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_5\}, \{v_5\}, \{v_4, v_6\}, \{v_2, v_6\} \}$$

Graphs (4)

Definition:
A **directed graph** is a pair $G = (V, E)$ that is composed of a non-empty vertex set V and an edge set $E \subseteq V \times V$. Edge $e = (u, v)$ connects vertex u with vertex v. $\deg^+(v)$: out-degree, $\deg^-(v)$: in-degree



$\deg^+(v_2) = 1$

$\deg^-(v_2) = 2$

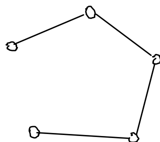
v_2 adjacent to v_4

v_4 adjacent from v_2

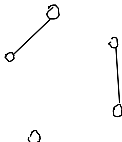
$G = (V, E),\ V = \{v_1 \ldots v_6\}$
$E = \{(v_1, v_2), (v_1, v_5), (v_2, v_4), (v_3, v_2), (v_4, v_4)\}$

Graphs (5)

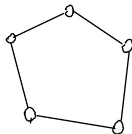
Fundamental Graph Properties and Classes:



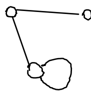
connected



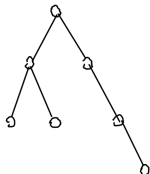
disconnected



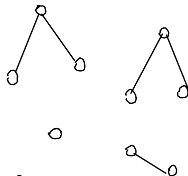
cycle



Loop ↰

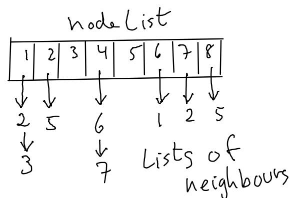


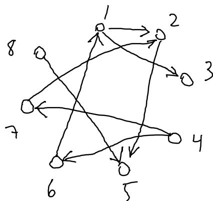
Connected + no cycles = tree

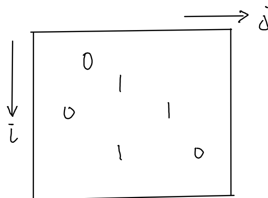


forest: each connected component is tree

Graph Data Structures







$V = \{v_1, \ldots, v_n\}$

Boolean $n \times n$ matrix A

$A[i,j] = 1 \Longleftrightarrow (v_i, v_j) \in E$

Adjacency lists are smaller if G is **sparse** ($|E| \in O(|V|)$). If G is **dense** ($|E| \in \Theta(|V|^2)$), adjacency matrices save space and allow direct access to edge information. Which representation is better depends on the algorithm used.

Content and Image Sources

- Various WWW resources (e.g., Wikipedia, xkcd)
- Own past course notes