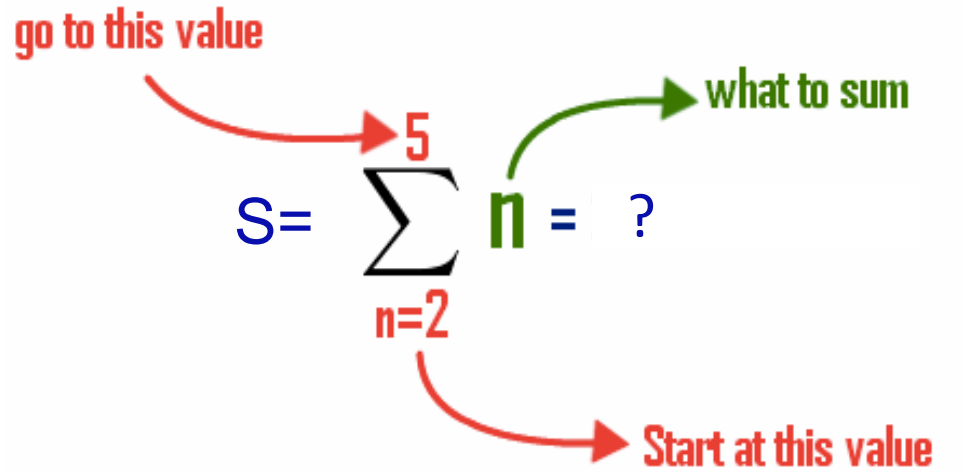
COMP20007 Workshop Week 3

1	Sigma notation & some important formulas
2	(Asymptotic) Complexity, Q2-3
3	mergesort & k-way merge, Q4-5
	5-min break
	Lab (new & important): multi-file program, `make` & `Makefile`

Sigma Notation



How do we interpret the above sigma:

- using high-school math?

  S = ?

- using pseudocode?

Q3.1: Sums

Give closed form expressions for the following sums.

(a) $\sum_{i=1}^{n} 1$

(b) $\sum_{i=1}^{n} i$

(c) $\sum_{i=1}^{n} (2i + 3)$

(d) $\sum_{i=0}^{n-1} \sum_{j=0}^{i} 1$

(e) $\sum_{i=1}^{n} \sum_{j=1}^{m} ij$

(f) $\sum_{k=0}^{n} x^k$

Review: Important Sums

$$\sum_{i=1}^{n} 1 = n$$

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$$

$$\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \quad (a \neq 1);$$

Sum Manipulation Rules

1. $$\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i$$

2. $$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$$

could be more familiar as:

$$\sum_{i=0}^{n} x^i = \frac{1 - x^{n+1}}{1 - x}$$

Big-O in simple words

In CS meaning (last year)	In strict meaning (this year)
$T(n) \in O(f(n)) \Leftrightarrow$ $f(n)$ is the lowest bound such that $T(n) \leq f(n)$ for all big n	$T(n) \in O(f(n)) \Leftrightarrow$ $T(n) \in \boldsymbol{\Omega}(f(n)) \Leftrightarrow$ $T(n) \in \theta(f(n)) \Leftrightarrow$

Example: In the luckiest case, my algo runs in nlogn time, but in the worst scenario it run in n^2 time, so

In CS meaning (last year)	In strict meaning (this year)
?	?

big-O: how to work out the running time of an algorithm?

Running time T = function of input size, ie. T= T(n)

Two models:

1. T(n)= number of *elementary computations* such as +, -, *, /, comparison, assignment…

2. T(n)= number of *basic operations*. Here:
   - *operation*= elementary operation
   - *basic operation:* usually the most time-consuming operation in the algorithm's innermost loop
   - = operation that executed most frequently, excluding the update of loop counters

sample algorithm (not quite meaningful)

```
function f(a0..n-1)
  x ← dosomething(a)
  S ← 0
  for i ← 0 to n-1 do
    S ← S + ai

  for i ← 0 to n-1 do
    for j ← 1 to n-2 do
      if ai = aj then
        S ← S + aj-1*aj+1

  return S + x/2;
```

model 2: basic operation = ?

big-O: how to work out the running time of an algorithm?

Running time T = function of input size, ie. T= T(n)

Our model:

T(n)= number of *basic operations*.
Here:

- *operation*= elementary operation or a constant number of elementary operations;
- *basic operation*= operation that executed most frequently, normally in the algorithm's innermost loop

sample algorithm (not quite meaningful)

```
function f(a0..n-1)
  x ← dosomething(a)   a0 + an-1
  S ← 0
  for i ← 0 to n-1 do
    S ← S + ai

  for i ← 0 to n-1 do
    for j ← 1 to n-2 do
      if ai = aj then
        S ← S + aj-1*aj+1

  return S + x/2;
```

basic operation = ?
complexity =

Q3.3: Sequential Search & Complexity

Use O, Ω and/or Θ to make *strongest possible claims* about the runtime complexity of sequential search in:

a) general

b) the best case

c) the worst case

d) the average case

?

?

?

?

sequential search

```
function search(A0..n−1, key)
  for i ← 0 to n
    if Ai = key
      return i
  return NOTFOUND
```

?

Ω/O/θ?

What is the most important?

- Big-θ, if available, otherwise:
- Big-O

big-O: examples for strict meaning

What'd we say in strict meaning:

algorithm 1	algorithm 2

function search($A_{0..n-1}$, key)
  for i ← 0 to n
  if A_i = key
  return i
  return NOTFOUND
	function sum(int A[1..n])
	  sum := 0
	  for i := 1 to n
	  sum := sum + A[i]
	  return sum

Ω/O/θ?	**Ω**/O/θ?

Q3.2

For each of the pairs of functions $f(n)$ and $g(n)$, determine if $f \in O(g)$, or $f \in \Omega(g)$, or both (ie., $f \in \Theta(g)$). Show your workout.

a. $f(n) = \frac{1}{2}n^2$ and $g(n) = 3n$

b. $f(n) = n^2 + n$ and $g(n) = 3n^2 + \log n$

c. $f(n) = n\log n$ and $g(n) = \frac{n}{4}\sqrt{n}$

d. $f(n) = \log(10n)$ and $g(n) = \log(n^2)$

e. $f(n) = (\log n)^2$ and $g(n) = \log(n^2)$

f. $f(n) = \log_{10} n$ and $g(n) = \ln n$

g. $f(n) = 2^n$ and $g(n) = 3^n$

h. $f(n) = n!$ and $g(n) = n^n$

Review: Simple Rules to find complexity classes

Basic complexity classes given in the lecture

$$1 \prec \log n \prec n^{\epsilon} \prec n^{c} \prec n^{\log n} \prec c^{n} \prec n^{n}$$

Simply apply 2 rules:

- **Rule 1**: In a sum, keep only the highest order element
- **Rule 2**: Replace any free (ie. not inside any function) constant with 1

Examples: find the complexity class for

$f(n) = 2n^2 + 6n + 1 = \Theta(\ )$

$g(n) = 5n\log_2 n + 1000n + 10^{20} = \Theta(\ )$

Review: Alternative & Powerful Method: Use lim to compare …

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\begin{cases}0 & \text{implies } t \text{ grows asymptotically slower than } g\\ c & \text{implies } t \text{ and } g \text{ have same order of growth}\\ \infty & \text{implies } t \text{ grows asymptotically faster than } g\end{cases}$$

$$t(n)= O(\,g(n)\,)\ \&\ t(n)\neq \boldsymbol{\Omega}(g(n))$$

$$t(n)= \Theta(\,g(n)\,)$$

$$t(n)= \Omega(\,g(n)\,)\ \&\ t(n)\neq O(\,g(n)\,)$$

Important

L'Hôpital Rule

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\lim_{n\to\infty}\frac{t'(n)}{g'(n)}$$

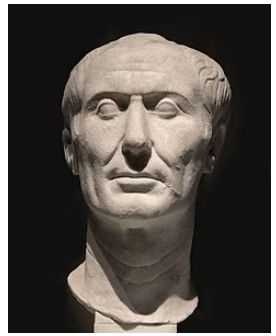
Self Review: How To Prove a Big-O, Big-Ω, Big-Θ

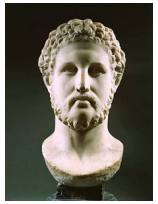
Use definitions!

Example:

Prove that $5n\log_2 n + 1000n + 10^{20} = \Theta(n\log_2 n)$

Divide & Conquer





Philip II of Macedon

Divide
&
Conquer

Image Sources: Wikipedia, cs.amherst.edu, https://commons.wikimedia.org/wiki/File:Caricatures_of_Napoleon_I_of_France_deta d%29.jpg

Divide & Conquer

To solve a size-*n* problem:

- Break the problem into a set of similar sub-problems, each of a *smaller-than-n* size,
- Solve each sub-problem in the same way (if simple enough, solve it directly), and
- Combine the solutions of sub-problems into the total solution.

Example: mergesort

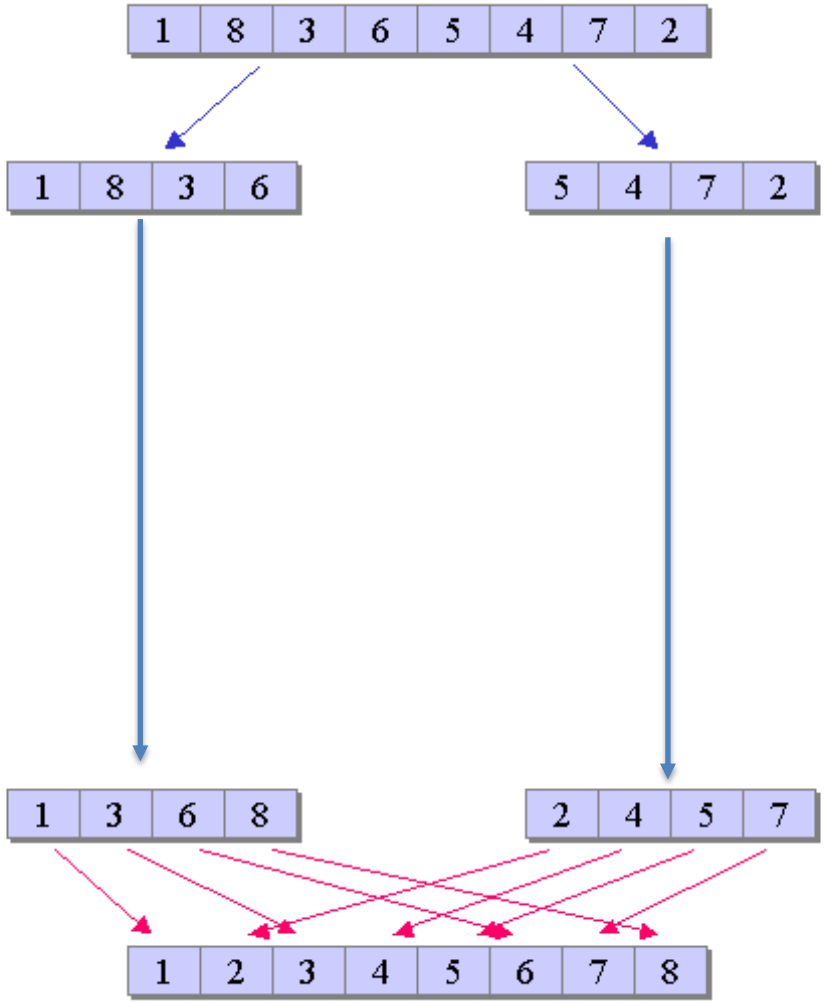


Image Sources: cs.amherst.edu

Q3.5 on Mergesort

Mergesort is a divide-and-conquer sorting algorithm made up of three steps (in the recursive case):

- Sort the left half of the input (using mergesort)
- Sort the right half of the input (using mergesort)
- Merge the two halves together (using a merge operation)

Using your intuition, see what you might expect **the complexity of mergesort** to be.

Q3.4: k-merge (aka. k-way merge)

Consider a modified sorting problem where the goal is to merge k lists of n sorted elements into one list of kn sorted elements.

One approach is to merge the first two lists, then merge the third with those, and so on until all k lists have been combined. What is the time complexity of this algorithm? Can you design a faster algorithm using a divide-and-conquer approach?

? complexity of merging one-by-one

? a faster algorithm using divide-and-conquer

5-min break

Lab Week 3: Modular Programming

Modular programming: breaking down a large program into smaller, independent modules or functions that can be developed and tested separately.

Each module is designed to perform a specific task or set of tasks, and communicates with other modules through well-defined interfaces.

Why?

How: in this lab we'll see how we technically do that in C.

Modular Programming

```
#include <stdio.h> …

declarations & function prototypes for working with linked list
typedef … node_t;
typedef … list_t;
list_t *create_list() ;
prototypes of other list functions

declarations & function prototypes...

… main(…) {
  …
  using linked lists
  using stacks
  …
}

implementation of linked list function
list_t *create_list() {
     …
     return …;
}
impl. of other list functions
implementation of stack functions
implementation of other functions
```

Why Modular Programming?

- program could be too long, complicated and un-manageable!

Modular Programming: Example

module "list"

- interface: data type defs, and function prototypes showing how to use linked lists
- implementation of all functions in the interface (and perhaps some useful internal functions)

module "stack"

- interface: data type defs, and function prototypes showing how to use stacks
- implementation of all functions in the interface (and perhaps some useful internal functions)
- possibly employ the list module

application program

```
#include <stdio.h> ...
// decl the use of module list
// decl the use of module stack


… main(…) {
 …
 //using linked list & stack facilities
 …
}
```

Benefits:

- each module can be developed and tested separately
- modules are reusable
- …

Work to do right now

1. Understand stuffs in Problem 1
2. Do Problem 2, need to know
   - how to compile `racecar.c` and `racecar_test.c` into the executable `racecar`_test
3. Do Problem 3 together with Anh, need to know
   - how to build a `Makefile` for automatic compilation
   -
4. Do Problem 4 to understand the practice of accessing module's facilities

Work to do at home

1. Finish outstanding work of the lab today
2. Do Problem 5
   - you can start with Alistair's `listops.c`, just google "`listops.c`"
3. [Optional] Make a minimal effort on building a module for stack (using linked list) and test it with "enter a series of number, then print them in reversed order.

Optional but Desirable Work

See `github.com/anhvir/c207` for a few different (and equivalent) versions of `Makefile` for this task

- First, just copy and paste `Makefile3` from `github` to your `Makefile` and try it.
- Read `Makefile1`, `Makefile2`, `Makefile3` in that order to understand `Makefile3`

Be confident that you can build a (naive, but working, just like Makefile3) `Makefile` for any C project.

Lab Week 3: Today's Work

Other work, perhaps at home:

Review: File Makefile and command make

Do It Yourself, at home: Lab 3.4: learn about data hiding, and to add a function into, module `racecar`

[Do Together if time permits]: Lab 3.5 (module for Linked List) : build your own `list.h`, `list.c`, `list_test.c`. As a reference, you can use Alistair's `listops.c` (just google "listops.c" to get the file).

[Extra, Optional] Use the list module to build a stack module in a least effort way. Test your stack by writing `stack_test.c` that input a series of integers, and then print them in reverse order. Build a single `Makefile` for testing both list and stack

Additional Slides

Useful Logarithm Rules

$\log_a n = \frac{\log_b n}{\log_b a} = \log_a b \times \log_b n = const \times \log_b n$

→ Base of logarithm doesn't matter:

$\log_{10} n$, $\ln(n)$, or $\log_2 n$ is just $\Theta(\log n)$

$\log_b n^k = \log_b (n^k) = k \times \log_b n = \Theta(\log n)$

but:

$(\log n)^c$ grows slower than $(\log n)^d$ iif $0 < c < d$

$\log_b xy = \log_b x + \log_b y$

$\log_b b^n = n$

$\log_b a = 1/\log_a b$

Notes on pseudocode

Indentation is obligatory (just like Python)

Other than indentation, the pseudocode styles are different in our 3 text books, and the style used in Skiena's book is even inconsistent sometimes.

Choose a style that you like, but be consistent

assignment can be written as ← or := or even =

In Skiena, the pseudocode is not quite consistently written (for example: "while something" can be used with or without "do")

Some forms that could be used:

My notes: Better to be consistent!

Notes on basic operation (from Levitin's sections 2.1, 2.3, 2.4)

Section 2.1: The thing to do is to identify the most important operation of the algorithm, called the ***basic operation***, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

As a rule, it is not difficult to identify the basic operation of an algorithm: it is usually the most time-consuming operation in the algorithm's innermost loop. For example, most sorting algorithms work by comparing elements (keys) of a list being sorted with each other; for such algorithms, the basic operation is a key comparison. As another example, algorithms for mathematical problems typically involve some or all of the four arithmetical operations: addition, subtraction, multiplication, and division. Of the four, the most time-consuming operation is division, followed by multiplication and then addition and subtraction, with the last two usually considered together

See Section 2.3: for non-recursive algorithms. As a rule, it is located in the inner-most loop.

Section 2.4: for recursive algorithms

if n=0 return 1else return F(n-1)*n: The basic operation of the algorithm is multiplication. Alternatively, we could count the number of times the comparison n = 0 is executed, which is the same as counting the total number of calls made by the algorithm

Modular Programming

```
#include <stdio.h> …

declarations & function prototypes for working with linked list
typedef … node_t;
typedef … list_t;
list_t *create_list() ;
prototypes of other list functions

declarations & function prototypes...

… main(…) {
  …
  using linked lists
  using stacks
  …
}

implementation of linked list function
list_t *create_list() {
     …
     return …;
}
impl. of other list functions
implementation of stack functions
implementation of other functions
```

Why Modular Programming?

- program could be too long, complicated and un-manageable!

Modular Programming: Example

module "list"

- interface: data type defs, and function prototypes showing how to use linked lists
- implementation of all functions in the interface (and perhaps some useful internal functions)

module "stack"

- interface: data type defs, and function prototypes showing how to use stacks
- implementation of all functions in the interface (and perhaps some useful internal functions)
- possibly employ the list module

application program

```
#include <stdio.h> ...
// decl the use of module list
// decl the use of module stack


… main(…) {
 …
 //using linked list & stack facilities
 …
}
```

Benefits:

- each module can be developed and tested separately
- modules are reusable
- …

How to compile?

Method 1:

```
gcc -o main main.c list.c stack.c
```

Method 2:

```
1. gcc -c main.c
```
that creates an *object file* `main.o`, which is in machine language, but containing some un-resolved function calls.

2. `gcc -c list.c`
3. `gcc -c stack.c`

that creates `list.o` and `stack.o`

4. `gcc -o main list.o main.o stack.o`

that combined the three object files into an executable file main, with all function calls resolved.

Q:
Method 2 seems complicated. What advantages?

Is there any problem with the line
`#include "stack.h"`
in main.c?

list.h — module "list" — list.c

```
declarations & function prototypes for working with linked list
typedef … node_t;
typedef … list_t;
list_t *create_list();
// prototypes of other list functions
```

```
#include "list.h"

list_t *create_list()
{
    …
    return …;
}
// impl. of other list functions
```

main.c

```
#include <stdio.h> …
#include "list.h"
#include "stack.h"

… main(…) {
  …
  using linked lists & stacks
  …
}

implementation of other non-list, non-stack functions
```

stack.h — module "stack" — stack.c

```
#include "list.h"
declarations & function prototypes for stack
```

```
#include "stack.h"
// impl. of  stack functions
```

Method 2 seems complicated. What advantages?

→ **avoid unnecessary recompilation**

→ **we can auto-compile with** `Makefile`

Is there any problem with the line

`#include "stack.h"`

in main.c?

- **duplication of some declarations**
- **we can avoid by:**

list.h

```
#ifndef _LIST_H_
#define _LIST_H_
    declarations & function
    prototypes for working with
    linked list
    typedef … node_t;
    typedef … list_t;
    list_t *create_list()
    ;
    // prototypes of other list
    functions

#endif
```

(header files = .h files)

list.h — module "list" — list.c

```
declarations & function
prototypes for working with
linked list
typedef … node_t;
typedef … list_t;
list_t *create_list();
// prototypes of other list
functions
```

```
#include "list.h"

list_t *create_list()
{
    …
    return …;
}
// impl. of other list functions
```

main.c

```
#include <stdio.h> …
#include "list.h"
#include "stack.h"

… main(…) {
  …
  using linked lists & stacks
  …
}

implementation of other non-list, non-stack functions
```

stack.h — module "stack" — stack.c

```
#include "list.h"
declarations & function
prototypes for stack
```

```
#include "stack.h"
// impl. of  stack functions
```