COMP20007 Workshop Week 3

1	Sigma notation & some important formulas, Q1 [10-15 min]
2	(Asymptotic) Complexity, Q2-3 [20-25 min]
3	mergesort, k-way merge, Q4-5 [15-20 min]
	5-min break
	Lab (new & important):
	multi-file program, `make` & `Makefile`

Sigma Notation

go to this value

what to sum

$$S = \sum_{n=2}^{5} n = \ ?$$

Start at this value

Sigma Notation

go to this value

what to sum

$$S = \sum_{n=2}^{5} n = 2+3+4+5 = 14$$

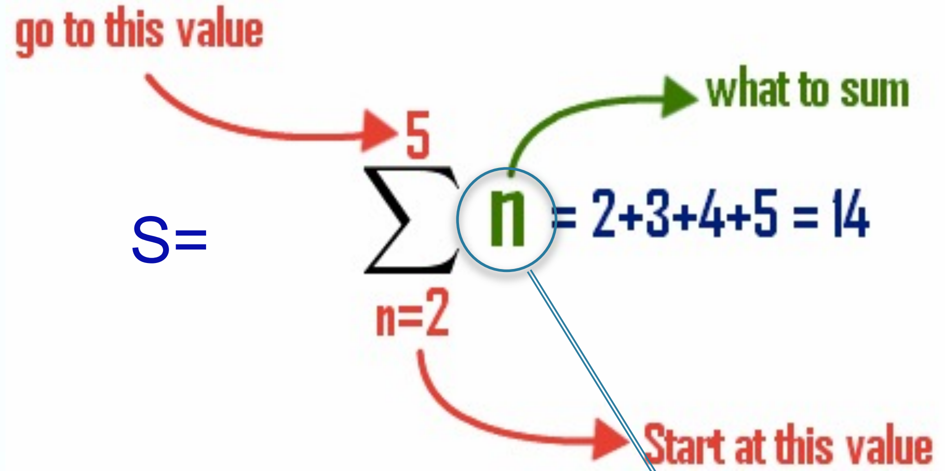
Start at this value

So, above sigma is just for computing:

```
S ← 0
for n ← 2 to 5
  S ← S + n
```

Sigma Notation

S=



So, above sigma is just for computing:

```
S ← 0
for n ← 2 to 5
  S ← S + n
```

What if this n is changed to:

a) 1
b) 2n+1
c) a_n

Sigma Notation

- Q: Use the sigma notation for the sum

a) $x_1 + x_2 + \dots + x_n$

$$= \sum_{?}^{?} ?$$

b) $a_{00} + a_{01} + \dots + a_{0n} + a_{10} + a_{11} + \dots + a_{nn}$

$$= \sum_{?}^{?} ?$$

Sum Manipulation Rules

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

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

Examples

$$\sum_{k=3}^{n} 5 \quad =$$

$$\sum_{k=0}^{n} (8\sqrt{i} + 3\ lna) \quad =$$

$$\sum_{i=0}^{n} 8ka^{i} \quad =$$

Important Sums

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

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

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

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

could be more familiar as:

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

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n + \gamma, \text{ where } \gamma \approx 0.5772\ldots$$

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

a) =

b) =

c) =

d) =

e) =

f) =

big-O: how to represent 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 or a constant number of elementary operations;
   - *basic operation*= operation that executed most frequently

sample algorithm
(not quite meaningful)

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

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

  return x + x/2;
```

model 1: T(n)=
model 2: T(n)=

big-O: what?

We associate the running time T(n) with a complexity n class. For instance:

- T(n)=O(n) : T(n) grows no faster than n, the running time is at most linear to n
- T(n)=Θ(n) : T(n) grows at the same rate as n, the running time is always linear to n

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

class O(n)? O(n^2)? Θ(n)?	class O(n)? O(n^2)? Θ(n)? Θ(n^2)?

- We're interested on the *asymptotic* behaviour of *T(n)*, ie. when $n \to \infty$.

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

?

Finding 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

- $f(n)= 2n^2+6n+1 \ = \Theta(\,)$
- $g(n)= 5n\log_2 n + 1000n + 10^{20} = \Theta(\,)$

Note: Check your understanding of the big-O definition by using it to prove a complexity. For example, prove that g(n)= $O(n \log_2 n)$.

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 $g(n) = \frac{n}{4}\sqrt{n}$

?

comparing f and g: method 1

Basic complexity classes given in the lecture

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

Method 1:

- **Step 1**: using Rule 1 and Rule 2 to reduce f and g to their classes
- **Step 2**: compare the classes using the list above

(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 $g(n) = \frac{n}{4}\sqrt{n}$

Method 2 (Powerful): 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

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

(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 $g(n) = \frac{n}{4}\sqrt{n}$

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.

(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 $g(n) = \ln n$

?

Useful Logarithm Rules

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

→ 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 $0 < c < d$

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

$\log_b b^n = n$

$\log_b a = 1/\log_a b$

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., $f \in \Theta(g)$).

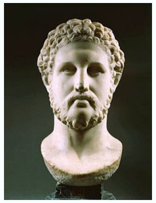
Show your workout.

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

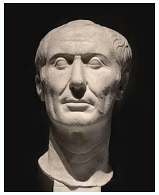
?

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

?


Philip II of Macedon

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.

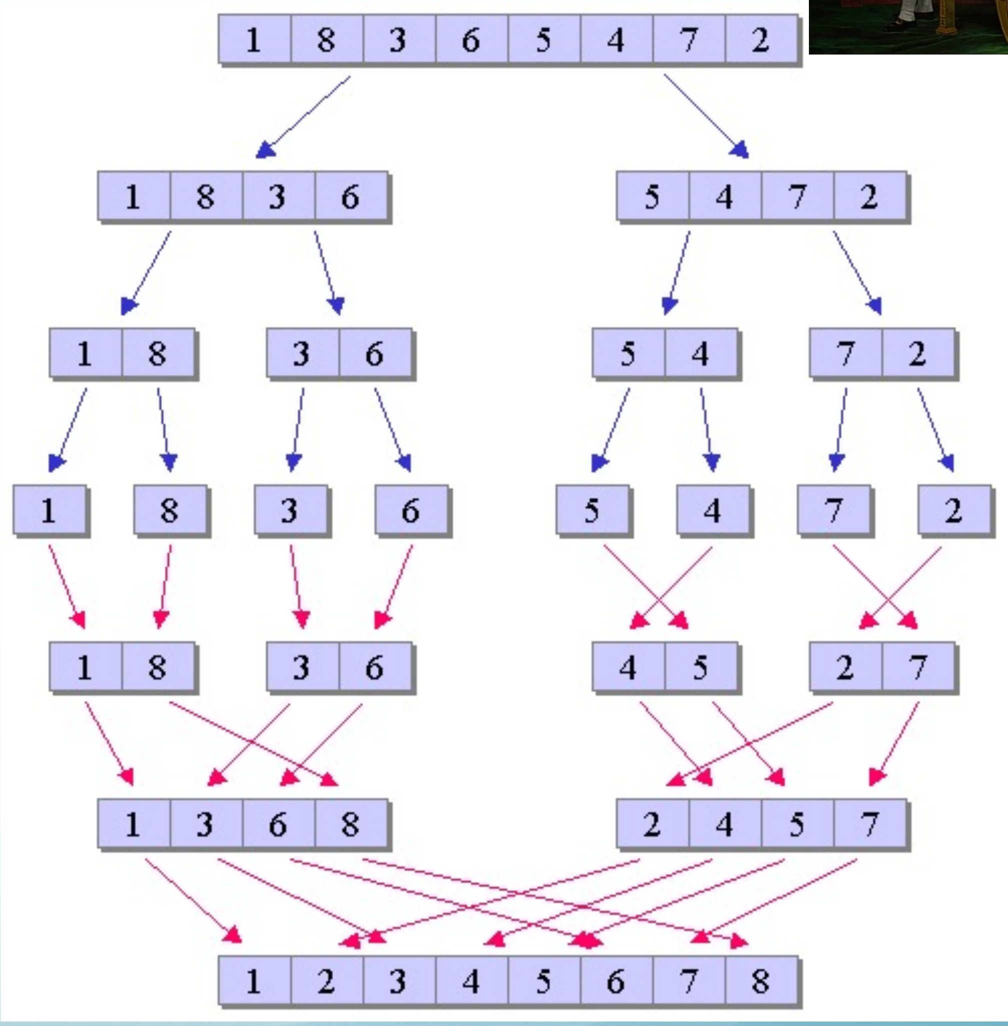


Image Sources: Wikipedia, cs.amherst.edu

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

Lab Week 3: Modular Programming

Modular programming: separating the functionality of a program into independent, interchangeable **modules**, such that each contains everything necessary to execute only one aspect of the desired functionality.

```
#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 for
working with stacks

declaration of other functions

… 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 and un-manageable!
- modules can easily be re-usable

```
#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
for working with stacks

declaration of other functions

… 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
```

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"   // paste the content of list.h here
#include "stack.h"

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

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

stack.h module "stack" stack.c

```
declarations & function
prototypes for working with
linked list
```

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

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?**

module “list”

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

list.c

#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

module “stack”

stack.h

declarations & function prototypes for working with linked list

stack.c

```
#include “stack.h”
#include “list.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
```

and that's a standard format for all header files (header files = .h files)

module "list"

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
```

list.c

```
#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
```

module "stack"

stack.h

```
declarations & function
prototypes for working with
linked list
```

stack.c

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

Lab Week 3: Modular Programming

- Follow steps 1—2 of the lab for building and testing module `racecar`
- Follow step 3 for using `make` and `Makefile` to auto-compile a multi-file program
- See `github.com/anhvir/c207` for a few different (and equivalent) versions of `Makefile` for `racecar` and `racecar_test`. First, just copy and paste `Makefile1` from `github` to your `Makefile` and try to "`make`".
- Follow step 4 to learn more about, and to add a function into, module `racecar`
- Do Step 5: 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).
- [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