COMP20007 Workshop Week 3

1	Prerequisites: Pseudocode, Sigma Notation, Q1
2	(Asymptotic) Complexity, Q2-3
	5-min break
3	(time permits) mergesort & k-way merge
	Lab:
	new & important: multi-file program, `make` & `Makefile`
	implement linked lists

Important Note

- A1 will be done in W4 and W5
- start early
- bring questions to W4 workshop

Notes on Pseudocodes

✍ Pseudocode: some statements

- *indentation is important,* just like in Python codes
- is not Python or C (so don't use specific stuffs like i++)
- assignment: ← (or **:=**), but *better be consistent*
- if & loops (**light blue** can be omitted, but *better be consistent*)

```
if a > b then
   max ← a
else
   max ← b
```

```
while i < n do
   S ← S + i
   ...
```

```
for i ← 1 to n do
   S ← S + i
   ...
```

```
for i ← n downto 1 do
   S ← S + i
   ...
```

```
for each x in set_X do
   S ← S + x
   ...
```

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

✓

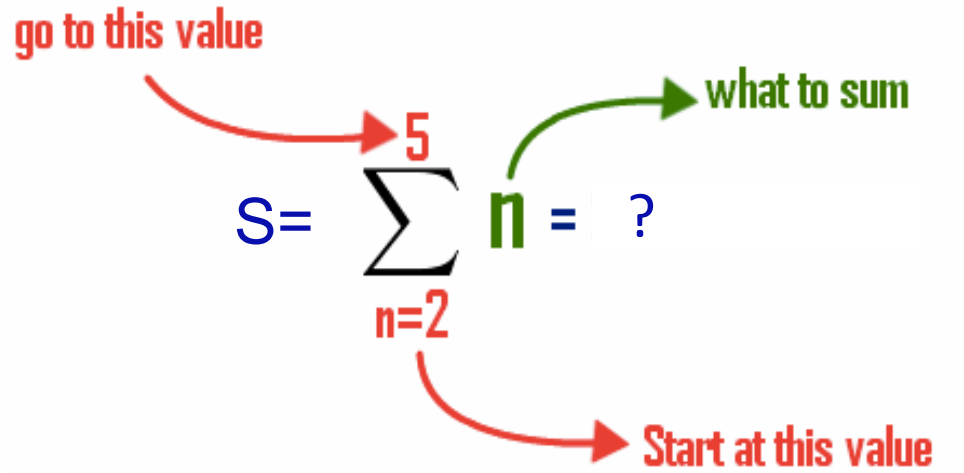
```
S := 0
for n:= 2 to 5
   S := S + n
```

✓

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

✗

Sigma Notation



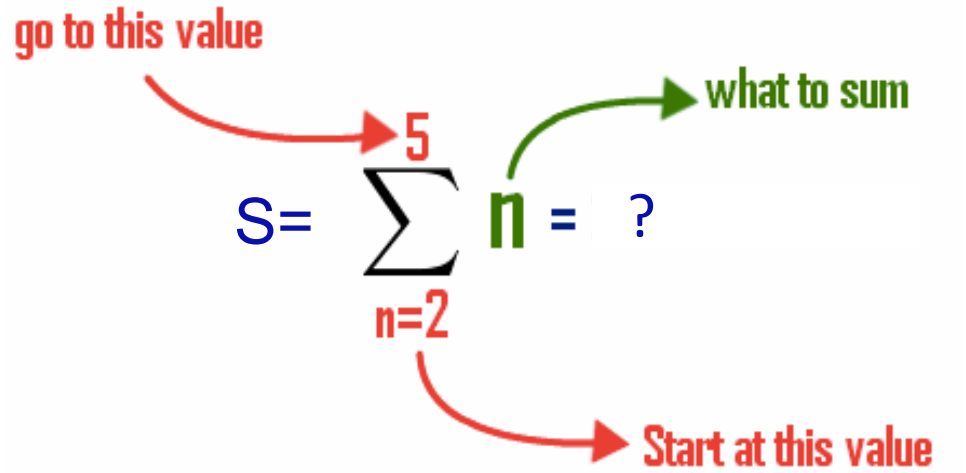
How do we interpret the above sigma:

- using high-school math?

  S = ?

- using pseudocode?

Sigma Notation



- using high-school math:

  S = 2+3+4+5 = 14

- using pseudocode?

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

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

Sum Manipulation Rules

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

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

Big-O definition

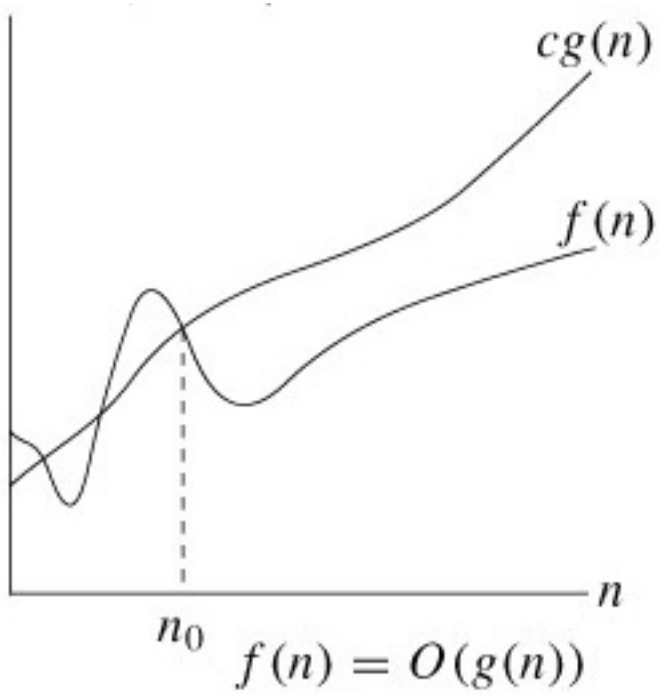
The complexity of an algorithm is the number of operations/steps and is expressed as a function f(n) of the *input size* n

Def: We say f(n) belongs to the class O(g(n)), that is, f(n) O(g(n)), iif

- There are constants c and n_0 such that f(n) < c.g(n) for all n > n_0

Underlying meaning:

- f(n) grows slower than, or as the same rate as, g(n)
- f(n) ≤ c.g(n) for some c and all large enough n

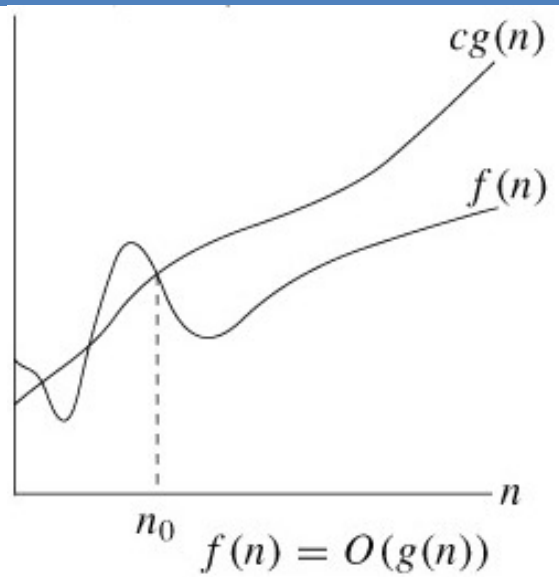


https://web.engr.oregonstate.edu/~huanlian/teaching/cs570/

Example: $f(n)= 3n^2 + 6n + 20$

prove that $f(n) \in O(n^2)$

from Big-O to Big-Ω and Big-θ

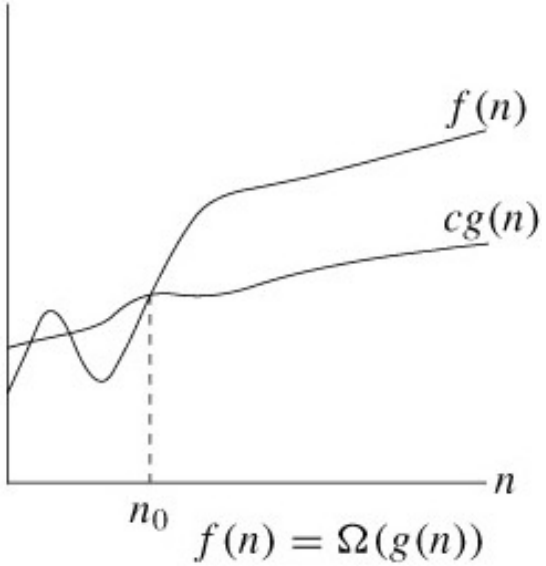


f(n) ∈ O(g(n))

- **Def:** There are constants c and n_0 such that

  f(n) < c.g(n) for all n > n_0

- f(n) grows slower than, or as the same rate as, g(n)
- f(n) ≤ c.g(n)



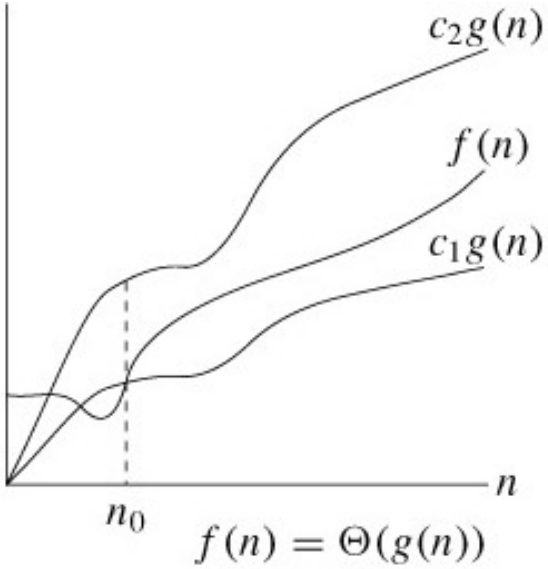
f(n) ∈ **Ω**(g(n))

⇔ g(n) ∈ O(f(n))

- **Def:** There are constants c and n_0 such that

  f(n) > c.g(n) for all n > n_0

- f(n) grows faster than, or as the same rate as, g(n)
- f(n) ≥ c.g(n)



f(n) ∈ θ(g(n))

⇔ f(n) ∈ O(g(n)) & f(n) ∈ **Ω**(g(n))

- **Def:** There are constants c_1, c_2 and n_0 such that

  c_1.g(n) < f(n) < c_2.g(n) for all n > n_0

- f(n) grows as the same rate as g(n)
- c_1.g(n) ≤ f(n) ≤ c_2.g(n)

Notes: if you want to **prove** that f(n) is O, or **Ω**, or **Θ** of some function, you need to use these definitions, you need to work out the value for the constants n_0 and c.

A simpler method for getting Big-Θ (or Big-O)

- Reduce f(n) to its simplest form (ie. complexity classes) using Big-O Reasoning

Big-O Reasoning

- $\Theta(f(n)+g(n)) = \Theta(\max(f(n), g(n)))$
- $\Theta(c\, f(n)) = \Theta(f(n))$
- $\Theta(f) * \Theta(g) = \Theta(f*g)$

For a complexity function, we can:

- Rule 1: keep the most dominant term only
- Rule 2: drop constants

Example:

- $f(n)= 2n^2 + 1000000n + 200$
- $g(n)= 5n + 9n\log n + 8$

Notes: The above reasonings are correct for Big-O and Big-Θ.
For Big-Ω, need to change max to min.

Panorama 1: Complexity Analysis for an Algorithm

Problem	Show the time efficiency of a given algorithm
Expected Solution	Point out the complexity class that the algorithm belongs to, using **Θ** or **O**. Some basic complexity classes: $1 \prec \log n \prec n^{\epsilon} \prec n^{c} \prec n^{\log n} \prec c^{n} \prec n^{n}$

Step 1	count t(n)= number of executions of the basic operations Notes: If t(n) also depends on data arrangement, we need to find t(n) for 2 different cases: the worst case (Big-O) and the best case (Big-

algorithm max	**algorithm** having_duplicates

`function num_duplicates(A[0..n-1])`
`  count= 0`
`  for i := 0 to n-1`
`  for j := i+1 to n-1`
`  if A[i]=A[j]`
`  count := count+1`
`  return count`
	`function having_duplicates(a0..n-1)`
	`  for i ← 0 to n-1 do`
	`  for j ← i+1 to n-1 do`
	`  if ai = aj then`
	`  return YES`
	`  return NO`

?	?

Step 2	Reduce t(n) to an efficiency class using the rules from Big-O reasoning: • If there are no distinct worst/best cases, we will find g(n) so that t(n)∈ **Θ**(g(n), otherwise • if and the best case is **Θ**(f(n)) and the worst case is **Θ**(g(n)) then t(n)∈**Ω**(f(n)) and t(n)∈O(g(n))

examples

algorithm max	**algorithm** having_duplicates

`function num_duplicates(A[0..n-1])`
`  count= 0`
`  for i := 0 to n-1`
`  for j := i+1 to n-1`
`  if A[i]=A[j]`
`  count := count+1`
`  return count`
	`function having_duplicates(`$a_{0..n-1}$`)`
	`  for i ← 0 to n-1 do`
	`  for j ← i+1 to n-1 do`
	`  if` $a_i = a_j$ `then`
	`  return YES`
	`  return NO`

basic op= key comparison t(n)= n(n-1)/2 t(n) ∈ O(?) **Ω**(?) θ(?)	basic op= key comparison best case t(n)= worst case t(n)= t(n) ∈ O(?) **Ω**(?) θ(?)

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) the best case

b) the worst case

c) the average case

d) general

?

?

?

?

sequential search

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

?

Ω/O/θ?

What is the most important?

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

Panorama 2: comparing 2 function t(n) and g(n) in terms of Big-O …

Method 1:

- Simplify `t(n)` and `f(n)` using the Big-O rules, then
- Using well-known classes to compare

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

Method 2:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies } t \text{ grows asymptotically slower than } g, \text{ ie. } t(n)= O(g(n)) \ \&\ t(n) \neq \Omega(g(n)) \\ c & \text{implies } t \text{ grows asymptotically as the same rate as } g, \text{ ie. } t(n)= \Theta(g(n)) \\ \infty & \text{implies } t \text{ grows asymptotically faster than } g, \text{ ie. } t(n)= \Omega(g(n)) \ \&\ t(n) \neq O(g(n)) \end{cases}$$

Method 3:

- Using definition of Big-O, Big-Ω, Big-Θ by pointing out the constants n_0 and c
- Example: To prove that t(n) ∈ O(f(n)), we need to point out 2 constants n_0 and c so that f(n) < c.g(n) for all n> n_0

Q3.2 – Group Work Please!

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 $g(n) = n^n$

Divide & Conquer (if time permits)

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

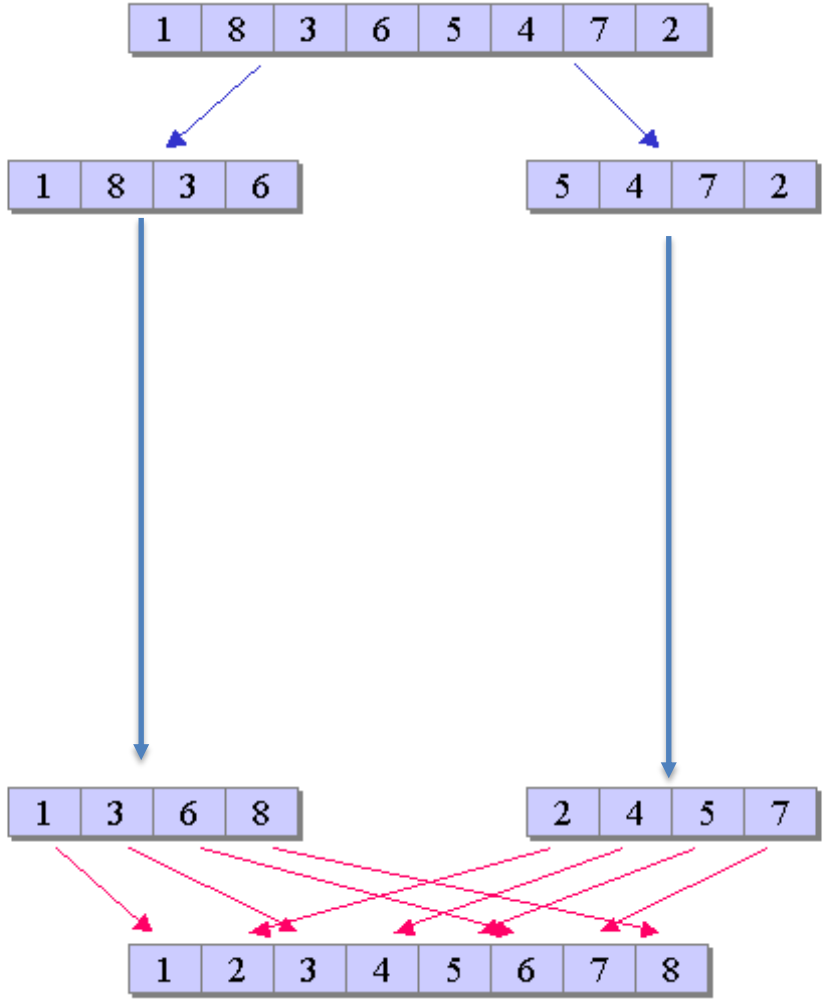


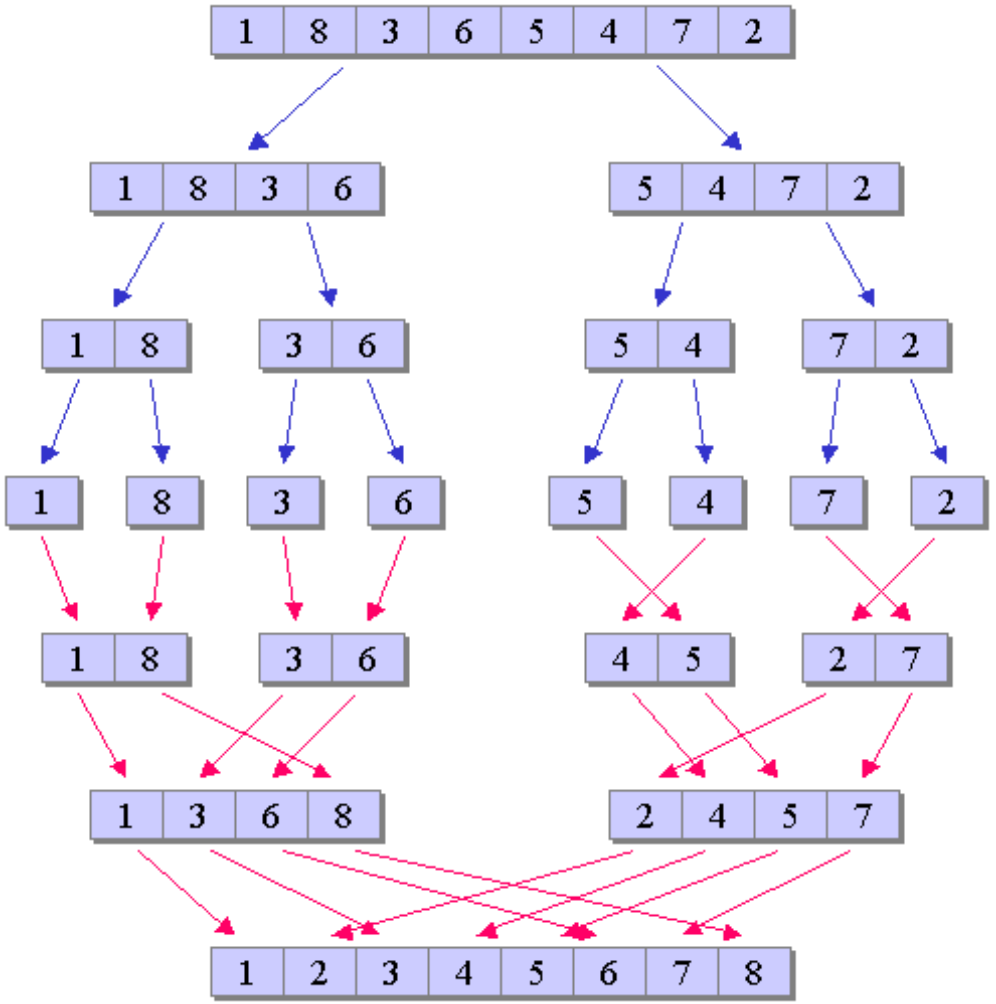
Image Sources: https://commons.wikimedia.org/wiki/File:Caricatures_of_Napoleon_I_of_France_detail,_Caricature_gillray_plumpudding_%28cropped%29.jpg, cs.amherst.edu

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

Peer Activity: Dynamic Arrays

What is the right ordering for these code snippets to implement a function

```
int ensure_array_size(struct array *arr)
```

that expands a struct array's data space when it is full? Assume that there is a "`return 0;`" at the end of the function body.

A. 3–2–5–1–4

B. 3–2–5–4–1

C. 2–5–1–4-3

D. 2–5–4–1-3

```
/* Snippet 1 */
arr->data = res;

/* Snippet 2 */
arr->size *= 2;

/* Snippet 3 */
if (arr->used < arr->size) return 0;

/* Snippet 4 */
if (res == NULL) {
    arr->size /= 2;
    return 1;
}

/* Snippet 5 */
void *res =
  realloc(arr->data, arr->size*sizeof(void*));
```

Peer Activity: Linked Lists - deleteHead

What is the correct ordering for these code snippets to implement a function

`void delete_head(struct list *lst)`

that deletes a `struct list`'s head node?

a. 2-5-3-4-1

b. 5-2-3-4-1

c. 2-5-4-1-3

d. 5-2-4-1-3

```
/* Snippet 1 */
free(old);
(lst->size)--;

/* Snippet 2 */
if ((lst->head == NULL) && (lst->tail == NULL))
    return;

/* Snippet 3 */
if (new == NULL) lst->tail = NULL;

/* Snippet 4 */
lst->head = new;

/* Snippet 5 */
struct lnode *old = lst->head,
             *new = old->next;
```

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. A module communicates with application programs or other modules through well-defined interfaces.

Compiling multi-file programs can be done automatically using command `make`.

Modular Programming: Example

module "dynamic_array"

- interface (`.h`): data type defs, and function prototypes
- source (`.c`) : implementation of all functions in the interface

module "list"

- interface (`.h`): data type defs, and function prototypes
- source (`.c`) : implementation of all functions in the interface

application program

```
#include <stdio.h> ...
// include the interface of module list
// include the interface of module dynamic array

int main(…) {
 …
 //using dynamic arrays & linked list facilities
 …
}
```

Benefits:

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

_test

1. Understand stuffs in W3.1
2. Do W3.2, need to know
   - how to compile `racecar.c` and `racecar.c` into the executable `racecar_test`

2 methods to compile racecar.c and racecar_test.c into the executable racecar_test

- Method 1

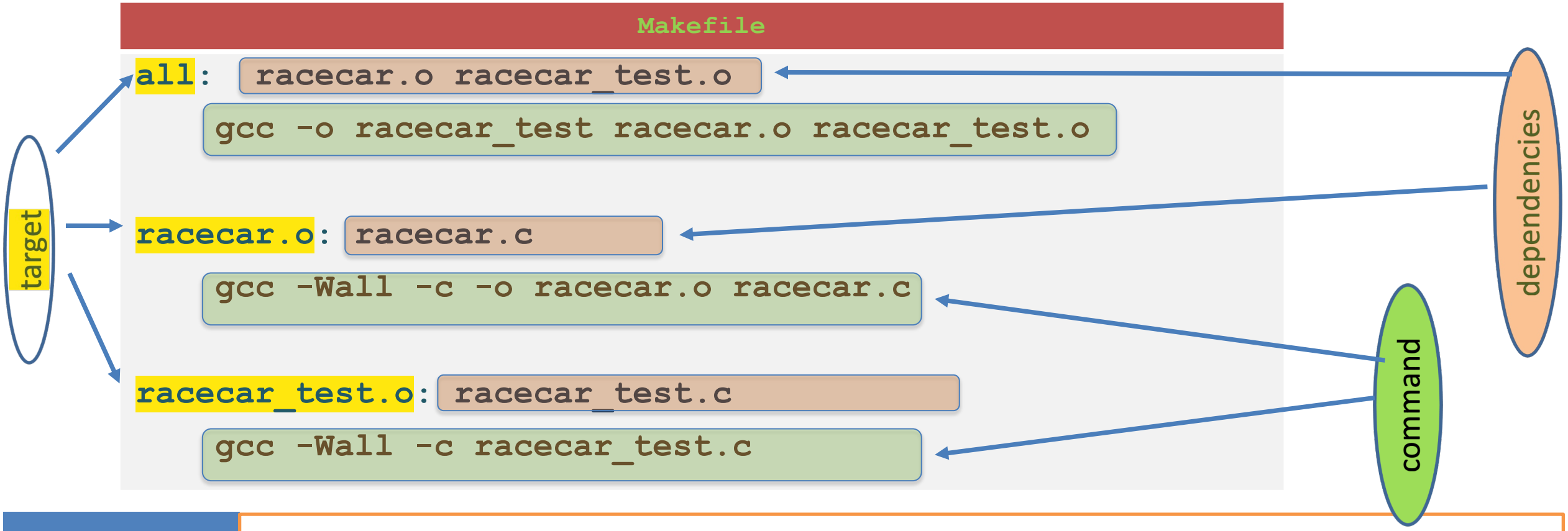
```
gcc –o racecar_test racecar.c racecar_test.c
```

- Method 2

```
gcc –c –o racecar.o racecar.c
gcc –c –o racecar_test.o racecar_test.c
gcc –o racecar_test racecar.o racecar_test.o
```

Notes:

- Method 2 is preferable for big project, and is used with make
- Components `–o racecar.o` and `–o racecar_test.o` can be omitted



auto-compiling with `make`

`$make` reads file **`Makefile`** and executes the first target (**`all`**, in this case)

- target **`all`** depends on 2 targets **`racecar.o`**, **`racecar_test.o`**
- first, executes 2 targets one-by-one
  - target **`racecar.o`** depends on **`racecar.c`**, which is ready as a file
  - executes command **`gcc -Wall -c -o racecar.o racecar.c`**
  - does similarly for racecar_test.o,
- second, runs the accompanied command to build **`racecar_test`**:

`gcc -o racecar_test racecar.o racecar_test`

Work to do right now

1. Understand stuffs in W3.1
2. Do W3.2, 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. [time permitting] Start Problem W3.4
   - build `list.c` and `list.h` (you can start with Alistair's listops.c, just google "`listops.c`", or use `list.c` and `list.h` from last week, but PLEASE at least implement function deleteHead)
   - build `list_test.c` (say, build and print the list 1→2→3→4→5)
   - build `Makefile` accordingly

Homework

1. Finish outstanding work of the lab today
2. Complete W3.4

Additional Slides

Important Sums & Rules

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

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

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

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

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:

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

Useful Exponent/Logarithm Rules

$\log_a b \times \log_b n = \log_a n$ where $\log_a b$ could be considered as just a constant

→ in the Big-O classes, base of logarithms is not important :
$\log_{10} n$, $\ln(n)$, or $\log_2 n$ is just $\Theta(\log n)$

Exponent Rules	Logarithm Rules
$a^m \cdot a^n = a^{m+n}$	$\log a + \log b = \log(ab)$
$\frac{a^m}{a^n} = a^{m-n}$	$\log a - \log b = \log\left(\frac{a}{b}\right)$
$(ab)^n = a^n b^n$	$\log(a^n) = n \log a$
$(a^m)^n = a^{mn}$	$a^{\log_b n} = n^{\log_b a}$

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