Using your laptop without Docker

1. Download `.PDF` file from `LMS.Workshop.Week 3`
2. Open editor (such as `Atom`, `jEdit`), and open an unix-style window (such as `Terminal`, `MobaXterm`, `minGW`)
3. When doing a programing question of the Workshop, just copy and paste the given skeleton into the editor's window and save in a proper `.c` file
4. If you cannot run `gcc` or `gdb` or `valgrind` in your laptop, you'd better to use `dimefox`/`nutmeg`:
   - In unix window, copy the `.c` file to uni's H: drive:

   ```
   scp my_prog.c bob@dimefox.eng.unimelb.edu.au:
   ```

   - Open another unix window and login into `dimefox`:

   ```
   ssh bob@dimefox.eng.unimelb.edu.au
   ```

   - then in this window you can run `gcc`, `gdb`, `valgrind`.
   - Pitfall: If you change your program, remember that it's on your laptop, so you have to `scp` it again before you can compile it on `dimefox`.

Now: Be ready with your laptop, and also navigate to github.com/anhvir/c203

COMP20003 Workshop Week 3

1	Asymptotic Complexity, Q2.1, Q2.2, Q2.4
2	Arrays: Static and Dynamic, Q2.3
3	Lab 1: • P2.1, P2.2
4	Lab 2: • github (Makefile/arrays) and/or • Programming Challenges

Asymptotic Complexity

We represent running time of an *algorithm* as $T(n)$, where n is the data size.

But then does $T1(n) = n^2 + 1$ differ from $T2(n) = 5n^2 - 7$?

We're interested on the *asymptotic* behaviour of $T(n)$.

Big-O (informal)

Equivalent writings:

1. $f(n) \in O(g(n)) : f(n) <= c.g(n)$ when n is big enough
2. $f(n) = O(g(n))$
3. $f(n)$ grows *no faster* than $g(n)$
4. $f(n)$ is dominated by $g(n)$
5. $\boldsymbol{g(n) = \Omega (f(n)) : g(n)}$ grows no slower than $\boldsymbol{f(n)}$

So, $2n + 3 = O(n) = O(n \log n) = O(n^2)$

$n^2 + 1 = \Omega(n^2) = \Omega(n \log n) = \Omega(n) = \Omega(1)$

and Big-Θ

Well, if $f(n) = O(g(n))$ and $f(n) = \Omega\ (g(n))$ then we say $f(n) = \Theta\ (g(n))$.

That is $f(n) = \Theta\ (g(n))$ iif $f(n)$ is sandwiched between $g(n)$ and $g(n)$, for when n big enough.

(Informal) Big-O Rules

Multiplicative constants can be reduced to 1:

$1000n^2$ or $0.0000001n^2$ is just n^2.

Base of logarithm doesn't matter:

$log_{10}n$ or $log_2 n$ is just $log\ n$

Lower-level additive parts can be omitted:

$2n^3 + 100000n^2 + 6n + 10^{12}$ is just n^3, and

$O(\ f(n) + g(n)\ ) = O(\ \max(f(n), g(n))\ )$

Also, for products:

$O(\ f(n)\ ) \times O(\ g(n)\ ) = O(\ f(n) \times g(n)\ )$

(Informal) Big-O Rules

Or, the order of growth if $1 < a < b$, $0 < c < d$:

$1 \ll (\log n)^a \ll (\log n)^b \ll n^c \ll n^d \ll a^n \ll b^n \ll n! \ll n^n$

const << log << poly << exp << factorial << n^n

Q 2.1

on 2.1 Given the following functions $f(n)$ and $g(n)$, is f in $O(g(n))$ or is f in $\Theta(g(n))$, or b

	$f(n)$	$g(n)$
(a)	$n + 100$	$n + 200$
(b)	$log_2(n)$	$log_{10}(n)$
(c)	2^n	2^{n+1}
(b)	2^n	3^n

Q2.2

1. big-O is used in the usual Computer Science sense, as the least upper bound (that is, same as big-Θ if the latter could be found).

2. big-O is used in its strictest sense (ie by definition) to mean any upper bound.

	Algorithms		Relative Performance	
	1	2	CS sense of big-O	strict sense of big-O
1	**$O(n \log n)$**	**$O(n^3)$**	?	?
2	**$\Theta(n \log n)$**	**$O(n^3)$**	?	?
3	**$O(n \log n)$**	**$\Theta(n^3)$**	?	?
4	**$\Theta(n \log n)$**	**$\Theta(n^3)$**	?	?

Q2.2

	Algorithms		Relative Performance	
	1	2	**CS sense of big-O**	**strict sense of big-O**
1	**$O(n \log n)$**	**$O(n^3)$**	Algorithm 2 grows faster than Algorithm 1	Algorithm 1 may run faster than Algorithm 2
2	**$\Theta(n \log n)$**	**$O(n^3)$**	?	?
3	**$O(n \log n)$**	**$\Theta(n^3)$**	?	?
4	**$\Theta(n \log n)$**	**$\Theta(n^3)$**	?	Algorithm 2 grows faster than Algorithm 1

For row 1, say, we can't say that algorithm 1 is faster than algorithm 2 (in both CS and strict senses).

For row 4, algorithm 1 is asymptotically better than algorithm 2 in terms of running time, but again, we can't say that 1 is faster than 2 for all n.

Q 2.4a

Give a characterization, in terms of big-O, big-Ω and big-Θ, of the following loops:

```
int p= 1;
for (int i = 0; i < 2*n; i++) {
     p=p*i;
}
```

Q 2.4b

Give a characterization, in terms of big-O, big-Ω and big-Θ, of the following loops:

```
int s = 0;
for(int i = 0; i < 2*n; i++){
    for(int j = 0; j < i; j++){
        s=s+i;
    }
}
```

Memory Management in C: static memory allocation

A variable declared *in a function* is **allocated** a suitable memory area by the compiler *at the start of the function execution*, and that memory is **freed** automatically when the *function execution ends*.

Our Function	Action of the Compiler
`… XXX (...) {`	
`int n;`	allocate 4 bytes for `n`
`int A[10];`	allocate 10 x 4 = 40 bytes for `A`
`char *p;`	allocate 8 bytes for `p`
`int *q;`	allocate 8 bytes for `q`
`...`	
`}`	free (ie returns to the system) all the 56 bytes

We use the above **static memory allocation** when we know in advance the size (or maximal size) of variables we employ.

Memory Management: if data size is unknown in advance

A Solution: work out the maximal size of the data …

→ simple, dangerous, offten innefficient, not always works

Solution 1: guess the maximal size	Notes

`#define MAX 20`
`… XXX (...) {`
`  char name[MAX+1];`
`  int A[10], B[10000], n=10;`
`  ...`
`  strcpy(name, “Trump”);`
`  strcpy(name, “1234567890123456789012”);`
 	
`  for (i=0; i<=nl; i++) {`
`    A[i]= i;`
`    B[i*i]= i*i;`
`  }`
`  ...`
`}`	

Memory Management: dynamic memory allocation

Better solution: we programmers, not the compilers, allocate and freed memory . Discipline: **malloc – check – free, each malloc must have one free**

	Comments
`… XXX (...) {`	
`char *name;`	Compiler allocates 8 bytes for `name`
`...`	
`name= calloc(4+1, sizeof(char));`	We allocate 5 bytes for `name`
`assert(name);`	We check to make sure that `calloc`/ `malloc` worked successfully
`// strlen("Trump") is 4`	
`strcpy(name, "Trump");`	
`...`	
`free(name);`	We free (ie returns to the system) all 5 bytes we allocated
`}`	

malloc & free: what’s wrong

```
int *p, *q;
*p= 100;
p= malloc(sizeof(int));
q= malloc(sizeof(*q));
*p= 100;
p= malloc(sizeof(*p));
free(p);


return 0;
```

Arrays: Static & Dynamic

What are static and dynamic memory allocation.

Compare:

Static	Dynamic

`#define N 100`
`int a[N];`
`scanf("%d", &n);`
 	
`for (i=0; i<n; i++)`
`  a[i]= i;`
…
	
	`int *a;`
	`scanf("%d", &n);`
	`a= calloc( n,  sizeof(*a));`
	`for (i=0; i<n; i++)`
	`  a[i]= i;`
	…
	
	`free(a);`

Q 2.3

What is the difference between the two following declarations?

```
1. int  a[10][20];
2. int *b[10];
```

1. How could you use them both as 2-dimensional arrays? (write the code)
2. What advantages might there be to declaring an array like `*b[]` above, instead of like `a[][]` above?
3. How could you make a variable declared as `int **c` into a 2-dimensional array? (write the code)

Q 2.3

What is the difference between the two following declarations?

1. `int  a[10][20];`
2. `int *b[10];`

Q 2.3

What is the difference between the two following declarations?

```
1. int  a[10][20];
2. int *b[10];
```

1. How could you use them both as 2-dimensional arrays? (write the code)

Q 2.3

What is the difference between the two following declarations?

1. `int  a[10][20];`
2. `int *b[10];`

2. What advantages might there be to declaring an array like `*b[]` above, instead of like `a[][]` above?

Q 2.3

What is the difference between the two following declarations?

1. `int  a[10][20];`
2. `int *b[10];`

3. How could you make a variable declared as `int **c` into a 2-dimensional array? (write the code) ???

Q 2.3

3. How could you make a variable declared as `int **c` into a 2-dimensional array of m rows, n columns? (write the code)

```
c= (int **) malloc ( m * sizeof(int *));
assert (c != NULL );  /* so now c has m cells */
for (i=0; i<m; i++) {
  c[i]= (int *) malloc( n * sizeof(int));
  assert(c[i] != NULL);
  /* now, each c[i] becomes an array of n int */
}

….
```

Q 2.3

3. After malloc memory for, say, c, we can use it. But after using c, before ending our program, we have to free all the allocated memory. Here is how to do it for c. We had:

```
c= (int **) malloc ( m * sizeof(int *));
for (i=0; i<m; i++) {
  c[i]= (int *) malloc( n * sizeof(int));
}
```

Now, for free, we just do 1 free for *every* malloc:

```
for (i=0; i<m; i++) {
  free (c[i]);
}
free (c);
```

Lab: see LMS→Workshops→Week3→Workshop.pdf

Lab 1:

- P2.1, P2.2

Lab 2:

- `github`: play with `Makefile` and `toy` and/or
- Programming Challenge 2.2