CS1010 Laboratory 07

Compiler, Multidimensional Arrays, Efficiency, Exercise 5

Zhang Puyu

Group BD04

October 17, 2024

Plan of the Day

Memory Management

```
long *long_array = malloc(m);
```

Memory Management


```
long *long_array = malloc(m);
```

Meaning: Allocate a memory space of **size** m for `long` **values**!

Memory Management


```
long *long_array = malloc(m);
```

Meaning: Allocate a memory space of **size** m for `long` **values**!

Question: **How many** `long` values can be stored in this memory space (i.e. array)?

Memory Management


```
long *long_array = malloc(m);
```

Meaning: Allocate a memory space of **size** m for `long` **values**!

Question: **How many** `long` values can be stored in this memory space (i.e. array)?
Answer: `m / sizeof(long)`

In 64-bits OS, this evaluates to $\frac{m}{8}$, and $\frac{m}{4}$ in 32-bits OS.

Memory Management


```
long *long_array = malloc(m);
```

Meaning: Allocate a memory space of **size** m for `long` **values**!

Question: **How many** `long` values can be stored in this memory space (i.e. array)?
Answer: `m / sizeof(long)`

In 64-bits OS, this evaluates to $\frac{m}{8}$, and $\frac{m}{4}$ in 32-bits OS.

So how do we allocate a `long` array of size n with `malloc`?

Memory Management


```
long *long_array = malloc(m);
```

Meaning: Allocate a memory space of **size** m for `long` **values**!

Question: **How many** `long` values can be stored in this memory space (i.e. array)?
Answer: `m / sizeof(long)`

In 64-bits OS, this evaluates to $\frac{m}{8}$, and $\frac{m}{4}$ in 32-bits OS.

So how do we allocate a `long` array of size n with `malloc`?

```
long *long_array = malloc(n * sizeof(long));
```

Memory Management


```
long *long_array = calloc(n, m);
```

Memory Management


```
long *long_array = calloc(n, m);
```

Meaning: Allocate a memory space for n values, each of size m.

Memory Management

```
long *long_array = calloc(n, m);
```

Meaning: Allocate a memory space for n values, each of size m.

So it is natural to write the above as

```
long *long_array = calloc(n, sizeof(long));
```

Memory Management

```
long *long_array = calloc(n, m);
```

Meaning: Allocate a memory space for *n* values, each of size *m*.

So it is natural to write the above as

```
long *long_array = calloc(n, sizeof(long));
```

Some of you did this:

```
long *long_array = (long *)calloc(n,
sizeof(long))
```

This is **NOT necessary!**

concat.c: Why Does This Work?


```c
char *concatenate(char *str1, char *str2) {
    size_t len1 = length_of(str1);
    size_t len2 = length_of(str2);
    char *result = malloc(len1 + len2 + 1);
    // Copy str1 and str2 into result
    // Append '\0' at the end of result
    return result;
}
```

concat.c: Why Does This Work?

```
char *concatenate(char *str1, char *str2) {
    size_t len1 = length_of(str1);
    size_t len2 = length_of(str2);
    char *result = malloc(len1 + len2 + 1);
    // Copy str1 and str2 into result
    // Append '\0' at the end of result
    return result;
}
```

It seems that the memory size put into `malloc` should be `(len1 + len2 + 1) * sizeof(char)` instead.

concat.c: Why Does This Work?

```
char *concatenate(char *str1, char *str2) {
    size_t len1 = length_of(str1);
    size_t len2 = length_of(str2);
    char *result = malloc(len1 + len2 + 1);
    // Copy str1 and str2 into result
    // Append '\0' at the end of result
    return result;
}
```

It seems that the memory size put into `malloc` should be `(len1 + len2 + 1) * sizeof(char)` instead.

But **this program runs without errors!**

concat.c: Why Does This Work?


```
char *concatenate(char *str1, char *str2) {
    size_t len1 = length_of(str1);
    size_t len2 = length_of(str2);
    char *result = malloc(len1 + len2 + 1);
    // Copy str1 and str2 into result
    // Append '\0' at the end of result
    return result;
}
```

It seems that the memory size put into `malloc` should be `(len1 + len2 + 1) * sizeof(char)` instead.

But **this program runs without errors!** Why?

concat.c: Why Does This Work?


```
char *concatenate(char *str1, char *str2) {
    size_t len1 = length_of(str1);
    size_t len2 = length_of(str2);
    char *result = malloc(len1 + len2 + 1);
    // Copy str1 and str2 into result
    // Append '\0' at the end of result
    return result;
}
```

It seems that the memory size put into `malloc` should be `(len1 + len2 + 1) * sizeof(char)` instead.

But **this program runs without errors!** Why?

Answer: `sizeof(char)` happens to be 1! (But still a bad practice to write like this!)

concat.c: Why Does This Work?


```c
char *concatenate(char *str1, char *str2) {
    size_t len1 = length_of(str1);
    size_t len2 = length_of(str2);
    char *result = calloc(2, len1 + len2 + 1);
    // Copy str1 and str2 into result
    return result;
}
```

concat.c: Why Does This Work?


```c
char *concatenate(char *str1, char *str2) {
    size_t len1 = length_of(str1);
    size_t len2 = length_of(str2);
    char *result = calloc(2, len1 + len2 + 1);
    // Copy str1 and str2 into result
    return result;
}
```

Tentative questions:

- Allocate 2 elements with size `len1 + len2 + 1` each???
- Never append `'\0'` to terminate the string???

concat.c: Why Does This Work?


```c
char *concatenate(char *str1, char *str2) {
    size_t len1 = length_of(str1);
    size_t len2 = length_of(str2);
    char *result = calloc(2, len1 + len2 + 1);
    // Copy str1 and str2 into result
    return result;
}
```

Tentative questions:

- Allocate 2 elements with size `len1 + len2 + 1` each???
- Never append `'\0'` to terminate the string???

Somehow the program still runs correctly. Why?

concat.c: Why Does This Work?

```
char *concatenate(char *str1, char *str2) {
    size_t len1 = length_of(str1);
    size_t len2 = length_of(str2);
    char *result = calloc(2, len1 + len2 + 1);
    // Copy str1 and str2 into result
    return result;
}
```

What happens in the backend: `calloc(m, n)` actually calculates mn and allocate a wholesome chunk of memory of size mn.

concat.c: Why Does This Work?

```
char *concatenate(char *str1, char *str2) {
    size_t len1 = length_of(str1);
    size_t len2 = length_of(str2);
    char *result = calloc(2, len1 + len2 + 1);
    // Copy str1 and str2 into result
    return result;
}
```

What happens in the backend: `calloc(m, n)` actually calculates mn and allocate a wholesome chunk of memory of size mn.

Since the size of `char` is 1 byte, `2 * (len1 + len2 + 1)` is **more than what we need**, so we can safely copy over the strings.

concat.c: Why Does This Work?

```c
char *concatenate(char *str1, char *str2) {
    size_t len1 = length_of(str1);
    size_t len2 = length_of(str2);
    char *result = calloc(2, len1 + len2 + 1);
    // Copy str1 and str2 into result
    return result;
}
```

`calloc` will also fill up the allocated memory space with 0's before returning, so the unused portion of `result` will be full of 0's.

concat.c: Why Does This Work?

```
char *concatenate(char *str1, char *str2) {
    size_t len1 = length_of(str1);
    size_t len2 = length_of(str2);
    char *result = calloc(2, len1 + len2 + 1);
    // Copy str1 and str2 into result
    return result;
}
```

`calloc` will also fill up the allocated memory space with 0's before returning, so the unused portion of `result` will be full of 0's.

However, `'\0' == 0`! So this unused portion does not affect the final output array.

`concat.c`: Why Does This Work?


```c
char *concatenate(char *str1, char *str2) {
    size_t len1 = length_of(str1);
    size_t len2 = length_of(str2);
    char *result = calloc(2, len1 + len2 + 1);
    // Copy str1 and str2 into result
    return result;
}
```

`calloc` will also fill up the allocated memory space with 0's before returning, so the unused portion of `result` will be full of 0's.

However, `'\0' == 0`! So this unused portion does not affect the final output array.

In conclusion, **passing test cases $\neq$ correct!**

NULL Pointer Checks

IMPORTANT UPDATE: As of this semester, not doing `NULL` checks when allocating dynamic arrays **WILL BE PENALISED** in PEs!

NULL Pointer Checks

IMPORTANT UPDATE: As of this semester, not doing NULL checks when allocating dynamic arrays **WILL BE PENALISED** in PEs!

What's the difference of the following two ways of checking for NULL pointers?

```
long *a = calloc(10, sizeof(long));
if (a != NULL) {
    // Do something with a
}
return 0;
```

```
long *a = calloc(10, sizeof(long
if (a == NULL) {
    cs1010_println_string("Error
    return 1;
}
// Do something with a
return 0;
```

NULL Pointer Checks

IMPORTANT UPDATE: As of this semester, not doing NULL checks when allocating dynamic arrays **WILL BE PENALISED** in PEs!

What's the difference of the following two ways of checking for NULL pointers?

```
long *a = calloc(10, sizeof(long));
if (a != NULL) {
    // Do something with a
}
return 0;
```

```
long *a = calloc(10, sizeof(long
if (a == NULL) {
    cs1010_println_string("Error
    return 1;
}
// Do something with a
return 0;
```

In the first program, allocation **fails silently** but in the second program we use **error messages** and **non-zero exit value** to **indicate the error**!

`char` Comparisons

`char` Comparisons

We can use arithmetic comparators and operators **directly** between characters.

`char` Comparisons

We can use arithmetic comparators and operators **directly** between characters.

E.g. in `up.c`, many of you wrote `c >= 97 && c <= 122` to check whether the character `c` is a lower-case alphabet.

`char` Comparisons

We can use arithmetic comparators and operators **directly** between characters.

E.g. in `up.c`, many of you wrote `c >= 97 && c <= 122` to check whether the character `c` is a lower-case alphabet.

Although `'a'` and `'z'` indeed have ASCII codes 97 and 122 respectively, using `c >= 'a' && c <= 'z'` is **much more readable**.

Do the Maths to Simplify Computations

Do the Maths to Simplify Computations

`kendall.c` asks: **given a permutation of integers from 1 to n inclusive, how many distinct pairs (a, b) are there which violate their natural ordering?**

Do the Maths to Simplify Computations

`kendall.c` asks: **given a permutation of integers from 1 to n inclusive, how many distinct pairs (a, b) are there which violate their natural ordering?**

$$\tau = \frac{\text{number of differently ranked pairs}}{\text{number of total pairs}}$$

Do the Maths to Simplify Computations

`kendall.c` asks: **given a permutation of integers from 1 to n inclusive, how many distinct pairs (a, b) are there which violate their natural ordering?**

$$\tau = \frac{\text{number of differently ranked pairs}}{\text{number of total pairs}}$$

Numerator: Just count with a loop.

Do the Maths to Simplify Computations

`kendall.c` asks: **given a permutation of integers from 1 to n inclusive, how many distinct pairs (a, b) are there which violate their natural ordering?**

$$\tau = \frac{\text{number of differently ranked pairs}}{\text{number of total pairs}}$$

Numerator: Just count with a loop.
Denominator: Do we need to count it with a loop too?

Do the Maths to Simplify Computations

`kendall.c` asks: **given a permutation of integers from 1 to n inclusive, how many distinct pairs (a, b) are there which violate their natural ordering?**

$$\tau = \frac{\text{number of differently ranked pairs}}{\text{number of total pairs}}$$

Numerator: Just count with a loop.
Denominator: Do we need to count it with a loop too?
It's just **the number of ways to choose two integers from n distinct integers**!

Do the Maths to Simplify Computations

`kendall.c` asks: **given a permutation of integers from 1 to n inclusive, how many distinct pairs (a, b) are there which violate their natural ordering?**

$$\tau = \frac{\text{number of differently ranked pairs}}{\text{number of total pairs}}$$

Numerator: Just count with a loop.
Denominator: Do we need to count it with a loop too?
It's just **the number of ways to choose two integers from n distinct integers**!

$$C_2^n = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}.$$

subtract.c

This is a variant of a classic problem called **TwoSum** (or **AddTwoNumbers**).

`subtract.c`

This is a variant of a classic problem called **TwoSum** (or **AddTwoNumbers**).

Just like doing **column subtraction**, we want to first "right-align" the two strings (i.e. iterate backwards).

subtract.c

This is a variant of a classic problem called **TwoSum** (or **AddTwoNumbers**).

Just like doing **column subtraction**, we want to first "right-align" the two strings (i.e. iterate backwards).

For the *i*-th digit, if `str1[i] >= str2[i]`:
`ans[i] = str1[i] - str2[i]`.
Otherwise:
Borrow 1 from the previous digit.

(Rather Brief) Introduction to `clang`

(Rather Brief) Introduction to clang

- In CS1010, we hide the details of compilation with `Makefile`.

Demo with the following program `test.c`:

```
int main() {
    return 0;
}
```

(Rather Brief) Introduction to `clang`

- In CS1010, we hide the details of compilation with `Makefile`.
- How to do it post-CS1010?

Demo with the following program `test.c`:

```c
int main() {
    return 0;
}
```

One thing you will notice is that if we compile it with `clang test.c`, we get `a.out` as the executable.

(Rather Brief) Introduction to `clang`

- In CS1010, we hide the details of compilation with `Makefile`.
- How to do it post-CS1010?
- Two major C/C++ compilers: `gcc` and `clang`.

Demo with the following program `test.c`:

```
int main() {
    return 0;
}
```

One thing you will notice is that if we compile it with `clang test.c`, we get `a.out` as the executable.

If we wish to rename the compiled program, we use `clang test.c -o name`.

(Rather Brief) Introduction to `clang`

- In CS1010, we hide the details of compilation with `Makefile`.
- How to do it post-CS1010?
- Two major C/C++ compilers: `gcc` and `clang`.

Demo with the following program `test.c`:

```c
int main() {
    return 0;
}
```

One thing you will notice is that if we compile it with `clang test.c`, we get `a.out` as the executable.

If we wish to rename the compiled program, we use `clang test.c -o name`.

CAUTION: `clang test.c -o test.c` will overwrite your code!

Compilation Errors/Warnings

We now try to purposely code something illegal:

```
int main() {
    main();
    return 0;
}
```

and compile again.

Compilation Errors/Warnings

We now try to purposely code something illegal:

```
int main() {
    main();
    return 0;
}
```

and compile again.

No warning is generated by default. We have to compile with `clang -Wall test.c` to enable warnings.

Compilation Errors/Warnings

We now try to purposely code something illegal:

```
int main() {
    main();
    return 0;
}
```

and compile again.

No warning is generated by default. We have to compile with `clang -Wall test.c` to enable warnings.

Compilation Errors/Warnings

Now try

```
int main() {
    int *p = NULL;
    *p = 1;
    return 0;
}
```

and compile with `clang -fsanitize=address test.c` to enable address sanitizer.

Compilation Errors/Warnings

Now try

```
int main() {
    int *p = NULL;
    *p = 1;
    return 0;
}
```

and compile with `clang -fsanitize=address test.c` to enable address sanitizer.

You will notice that the line number of the buggy code is not displayed as expected. To make it visible, we need to use the `-g` flag by calling:
`clang -fsanitize=address -g test.c`

External Libraries

Now try

```
#include <math.h>

int main() {
    sqrt(4)
    return 0;
}
```

Compiling the above will trigger something called **linker command failure**.

External Libraries

Now try

```
#include <math.h>

int main() {
    sqrt(4)
    return 0;
}
```

Compiling the above will trigger something called **linker command failure**.

This is because we use a program called `ld` to resolve external function calls, and need to tell `clang` where the function `sqrt` can be found (in this case, it is defined in `libm.a` under `/usr/lib`).

External Libraries

Now try

```
#include <math.h>

int main() {
    sqrt(4)
    return 0;
}
```

Compiling the above will trigger something called **linker command failure**.

This is because we use a program called `ld` to resolve external function calls, and need to tell `clang` where the function `sqrt` can be found (in this case, it is defined in `libm.a` under `/usr/lib`).

We will use `clang test.c -lm`.

External Libraries

What if we want to use a third-party library like the CS1010 I/O Library?

```
#include "cs1010.h"

int main() {
    cs1010_println_long(1);
    return 0;
}
```

Since `cs1010.h` is not a standard C library, we need to tell `clang` to look for it under `~cs1010/include` by `clang -I ~cs1010/include test.c`.

External Libraries

What if we want to use a third-party library like the CS1010 I/O Library?

```c
#include "cs1010.h"

int main() {
    cs1010_println_long(1);
    return 0;
}
```

Since `cs1010.h` is not a standard C library, we need to tell `clang` to look for it under `~cs1010/include` by `clang -I ~cs1010/include test.c`.

But after doing so, there are still errors! This is because we also need to tell `clang` where to find the function `cs1010_println_long`. Run:

`clang -I ~cs1010/include -L ~cs1010/lib test.c -lcs1010`

Compiler Flags

We can save the compiler flags into a `.txt` file with one flag per line.

Compiler Flags

We can save the compiler flags into a `.txt` file with one flag per line.

A more modernised approach:
https://code.visualstudio.com/docs/languages/cpp

An Array of Arrays

Consider an $(m+1) \times (n+1)$ matrix:

$$\boldsymbol{A} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m0} & a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}.$$

An Array of Arrays

Consider an $(m+1) \times (n+1)$ matrix:

$$\boldsymbol{A} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m0} & a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}.$$

It can be re-written as

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{a}_0 \\ \boldsymbol{a}_1 \\ \vdots \\ \boldsymbol{a}_m \end{bmatrix}$$

where each of the $\boldsymbol{a}_i$'s is an $1 \times (n+1)$ matrix, i.e., **an array of size $n+1$!**

An Array of Arrays

Consider an $(m+1) \times (n+1)$ matrix:

$$\boldsymbol{A} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m0} & a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}.$$

It can be re-written as

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{a}_0 \\ \boldsymbol{a}_1 \\ \vdots \\ \boldsymbol{a}_m \end{bmatrix}$$

where each of the $\boldsymbol{a}_i$'s is an $1 \times (n+1)$ matrix, i.e., **an array of size $n+1$!** So the matrix is essentially **an array of size $(m+1)$** whose elements are **arrays of size $(n+1)$!**

Multidimensional Arrays: Non-contiguous Memory

```c
long **matrix = calloc(m, sizeof(long *));
if (matrix == NULL) {
    cs1010_println_string("Memory allocation failed");
    return 1;
}
for (long i = 0; i < m; i += 1) {
    matrix[i] = calloc(n, sizeof(long));
    if (matrix[i] == NULL) {
        cs1010_println_string("Memory allocation failed");
        for (long j = 0; j < i; j += 1) {
            free(matrix[j]);
        }
        free(matrix);
        return 1;
    }
}
// Do something with matrix
for (long i = 0; i < n; i += 1) {
    free(matrix[i]);
}
free(matrix);
return 0;
```

Multidimensional Arrays: Contiguous Memory

```c
long **matrix = calloc(m, sizeof(long *));
if (matrix == NULL) {
    cs1010_println_string("Memory allocation failed");
    return 1;
}
matrix[0] = calloc(m * n, sizeof(long));
if (matrix[0] == NULL) {
    cs1010_println_string("Memory allocation failed");
    free(matrix);
    return 1;
}
for (long i = 1; i < m; i += 1) {
    matrix[i] = matrix[i - 1] + n;
}
// Do something with matrix
free(matrix[0]);
free(matrix);
return 0;
```

Play with Pointers

Due to **array decay**, the address of an array is the same as the address of its first element.

Play with Pointers

Due to **array decay**, the address of an array is the same as the address of its first element.

Any 2D array `a` is still an array, so `a` is just the **address of its first element**, i.e., `a == &a[0]` (they are the same both in value and in meaning).

Play with Pointers

Due to **array decay**, the address of an array is the same as the address of its first element.

Any 2D array `a` is still an array, so `a` is just the **address of its first element**, i.e., `a == &a[0]` (they are the same both in value and in meaning).

However, note that `a[0]` is also an array, so what is `a[0]`?

Play with Pointers

Due to **array decay**, the address of an array is the same as the address of its first element.

Any 2D array `a` is still an array, so `a` is just the **address of its first element**, i.e., `a == &a[0]` (they are the same both in value and in meaning).

However, note that `a[0]` is also an array, so what is `a[0]`? It is the **address of the first element of `a[0]`**, i.e., `a[0] == &a[0][0]` (they are the same both in value and in meaning).

Play with Pointers

Due to **array decay**, the address of an array is the same as the address of its first element.

Any 2D array `a` is still an array, so `a` is just the **address of its first element**, i.e., `a == &a[0]` (they are the same both in value and in meaning).

However, note that `a[0]` is also an array, so what is `a[0]`? It is the **address of the first element of `a[0]`**, i.e., `a[0] == &a[0][0]` (they are the same both in value and in meaning).

Question: Do you think

1. `a == &a`?
2. `a[0] == &a[0]`?

Play with Pointers

Due to **array decay**, the address of an array is the same as the address of its first element.

Any 2D array `a` is still an array, so `a` is just the **address of its first element**, i.e., `a == &a[0]` (they are the same both in value and in meaning).

However, note that `a[0]` is also an array, so what is `a[0]`? It is the **address of the first element of `a[0]`**, i.e., `a[0] == &a[0][0]` (they are the same both in value and in meaning).

Question: Do you think

1. `a == &a`?
2. `a[0] == &a[0]`?

Answer: **No** because they have different types.

Time Complexity Analysis Using Big-O

Big-O Notation (the Real Definition)

Let f and g be real-valued one-dimensional functions. If there exists some $x_0 \in \mathbb{R}$ such that for all $x > x_0$, we have

$$|f(x)| \leq Mg(x)$$

for some $M > 0$, then we say that $f(x)$ is **big-O** of $g(x)$, denoted by $f(x) = O\big(g(x)\big)$ or $f \in \mathcal{O}(g)$.

Time Complexity Analysis Using Big-O

Big-O Notation (the Real Definition)

Let f and g be real-valued one-dimensional functions. If there exists some $x_0 \in \mathbb{R}$ such that for all $x > x_0$, we have

$$|f(x)| \leq Mg(x)$$

for some $M > 0$, then we say that $f(x)$ is **big-O** of $g(x)$, denoted by $f(x) = O\big(g(x)\big)$ or $f \in \mathcal{O}(g)$.

In human language, this means that the **growth rate** of f is **at most** a constant multiple of the growth rate of g.

Time Complexity Analysis Using Big-O

Big-O Notation (the Real Definition)

Let f and g be real-valued one-dimensional functions. If there exists some $x_0 \in \mathbb{R}$ such that for all $x > x_0$, we have

$$|f(x)| \leq Mg(x)$$

for some $M > 0$, then we say that $f(x)$ is **big-O** of $g(x)$, denoted by $f(x) = O\big(g(x)\big)$ or $f \in \mathcal{O}(g)$.

In human language, this means that the **growth rate** of f is **at most** a constant multiple of the growth rate of g.

Meaning of $\mathcal{O}(g)$:

Time Complexity Analysis Using Big-O

Big-O Notation (the Real Definition)

Let f and g be real-valued one-dimensional functions. If there exists some $x_0 \in \mathbb{R}$ such that for all $x > x_0$, we have

$$|f(x)| \leq Mg(x)$$

for some $M > 0$, then we say that $f(x)$ is **big-O** of $g(x)$, denoted by $f(x) = O\big(g(x)\big)$ or $f \in \mathcal{O}(g)$.

In human language, this means that the **growth rate** of f is **at most** a constant multiple of the growth rate of g.

Meaning of $\mathcal{O}(g)$: it is the set of all functions whose growth rate is different from the growth rate of g by **at most a constant factor**.

Trivial Results

- For all $c > 0$, $f(x) = O(c) \iff f(x) = O(1)$.
- For all $c > 0$, $f(x) = O\big(g(x)\big) \iff f(x) = O\big(c \cdot g(x)\big)$.
- For all $m, n > 0$,
  $f(x) = O\big(g(x) + h(x)\big) \iff f(x) = O\big(m \cdot g(x) + n \cdot h(x)\big)$.
- For all $a_1, a_2, \cdots, a_n > 0$,

$$f(x) = O\left(\sum_{i=1}^{n} g_i(x)\right) \iff f(x) = O\left(\sum_{i=1}^{n} a_i g_i(x)\right).$$

The proofs are left to the reader as an exercise.

Some Serious Maths

Rule of Highest Power

Let $p(x)$ be a polynomial, i.e.,

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0,$$

where $n \in \mathbb{N}$ and a_i's are real constants, then

$$f(x) = O\big(p(x)\big) \iff f(x) = O\big(x^n\big).$$

The ($\Longleftarrow$) direction is somewhat useless in the context of programming, so we will prove the ($\Longrightarrow$) direction only.

Some Serious Maths

Rule of Highest Power

Let $p(x)$ be a polynomial, i.e.,

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0\,,$$

where $n \in \mathbb{N}$ and a_i's are real constants, then

$$f(x) = O\big(p(x)\big) \iff f(x) = O(x^n)\,.$$

Observe that for all $x > 1$,

$$\begin{aligned} p(x) &\leq |a_n|x^n + \cdots + |a_1|x + |a_0| \\ &\leq \big(|a_n| + |a_{n-1}| + \cdots + |a_0|\big)x^n \end{aligned}$$

Take $N = \big(|a_n| + |a_{n-1}| + \cdots + |a_0|\big)$. Since $f(x) = O\big(p(x)\big)$, there is some $x_0 \in \mathbb{R}$ and some $M > 0$ such that

$$|f(x)| \leq Mp(x)$$

for all $x > x_0$. Take $x' = \max\{x_0, 1\}$, then for all $x > x'$,

$$|f(x)| \leq Mp(x) \leq M\big(Nx^n\big) = MNx^n.$$

Since $MN > 0$, this means $f(x) = O(x^n)$. $\square$

Compare Growth Rate (the Problematic Statement)

"We say that $f(n)$ grows faster than $g(n)$ if we can find a n_0, such that $f(n) > cg(n)$ for all $n > n_0$ and for some constant c."

Compare Growth Rate (the Problematic Statement)

> *"We say that $f(n)$ grows faster than $g(n)$ if we can find a n_0, such that $f(n) > cg(n)$ for all $n > n_0$ and for some constant c."*

Counter example:

- $f(n) = n^3$,
- $g(n) = 6n^3$.

Compare Growth Rate (the Problematic Statement)

> *"We say that $f(n)$ grows faster than $g(n)$ if we can find a n_0, such that $f(n) > cg(n)$ for all $n > n_0$ and for some constant c."*

Counter example:

- $f(n) = n^3$,
- $g(n) = 6n^3$.
- Fix $n_0 = 1$, clearly for all $n > 1$, we can fix constant $c = \frac{1}{12}$ and
- $f(n) > cg(n)$.

Compare Growth Rate (the Problematic Statement)

> *"We say that $f(n)$ grows faster than $g(n)$ if we can find a n_0, such that $f(n) > cg(n)$ for all $n > n_0$ and for some constant c."*

Counter example:

- $f(n) = n^3$,
- $g(n) = 6n^3$.
- Fix $n_0 = 1$, clearly for all $n > 1$, we can fix constant $c = \frac{1}{12}$ and
- $f(n) > cg(n)$.
- However, this is ridiculous because $g(n) \geq f(n)$ for all $n \geq 0$.

Compare Growth Rate (the Problematic Statement)

> *"We say that $f(n)$ grows faster than $g(n)$ if we can find a n_0, such that $f(n) > cg(n)$ for all $n > n_0$ and for some constant c."*

Counter example:

- $f(n) = n^3$,
- $g(n) = 6n^3$.
- Fix $n_0 = 1$, clearly for all $n > 1$, we can fix constant $c = \frac{1}{12}$ and
- $f(n) > cg(n)$.
- However, this is ridiculous because $g(n) \geq f(n)$ for all $n \geq 0$.
- Both functions are in $\mathcal{O}(n^3)$ so neither should grow faster than the other.

Compare Growth Rate (the Real Way)

What do we mean by f grows faster than g:

As $x \to \infty$, $f(x)$ is **significantly larger** than $g(x)$.

Naturally, we check

$$\lim_{x \to \infty} \frac{f(x)}{g(x)}.$$

If $\frac{f(x)}{g(x)}$ diverges (i.e. limit “equals” ∞), then f grows faster. If the limit is 0, then g grows faster. If the limit is some non-zero constant, then f and g have the same growth rate.

add.c

`add.c`

These are just two **fixed-sized 2D arrays** (3×3 matrices). We just need to read them in and perform element-wise addition.

line.c

`line.c`

- Make sure you understand the maths part.

`line.c`

- Make sure you understand the maths part.
- Note that the origin is at **top-left** instead of bottom-left.

`line.c`

- Make sure you understand the maths part.
- Note that the origin is at **top-left** instead of bottom-left.
- Note also that the point (x, y) corresponds to the element `matrix[y][x]`.

`line.c`

- Make sure you understand the maths part.
- Note that the origin is at **top-left** instead of bottom-left.
- Note also that the point (x, y) corresponds to the element `matrix[y][x]`.
- `round` function from `math.h` truncates a floating point number to the nearest integer.

`line.c`

- Make sure you understand the maths part.
- Note that the origin is at **top-left** instead of bottom-left.
- Note also that the point (x, y) corresponds to the element `matrix[y][x]`.
- `round` function from `math.h` truncates a floating point number to the nearest integer.
- For i from 0 to $x_2 - x_1$, calculate the coordinates $(x_1 + i, y_1 + im)$ and find the indices after rounding.

`line.c`

- Make sure you understand the maths part.
- Note that the origin is at **top-left** instead of bottom-left.
- Note also that the point (x, y) corresponds to the element `matrix[y][x]`.
- `round` function from `math.h` truncates a floating point number to the nearest integer.
- For i from 0 to $x_2 - x_1$, calculate the coordinates $(x_1 + i, y_1 + im)$ and find the indices after rounding.
- Is it possible for the coordinates to be outside of the matrix's boundary?

popular.c

$$\begin{bmatrix} 1 & & & & & \\ ? & 1 & & & & \\ ? & ? & 1 & & & \\ ? & ? & ? & 1 & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ ? & ? & ? & ? & \cdots & 1 \end{bmatrix}$$

`popular.c`

$$\begin{bmatrix} 1 & & & & & \\ ? & 1 & & & & \\ ? & ? & 1 & & & \\ ? & ? & ? & 1 & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ ? & ? & ? & ? & \cdots & 1 \end{bmatrix}$$

- Understanding the problem: if `matrix[i][j] == 1` and $i \neq j$, it means that i and j are friends.

popular.c

$$\begin{bmatrix} 1 & & & & & \\ ? & 1 & & & & \\ ? & ? & 1 & & & \\ ? & ? & ? & 1 & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ ? & ? & ? & ? & \cdots & 1 \end{bmatrix}$$

- Understanding the problem: if `matrix[i][j] == 1` and $i \neq j$, it means that i and j are friends.
- Given any x, our task: find out how many integers $i \neq x$ satisfy `matrix[x][i] == 1` or `matrix[i][x] == 1`

popular.c

$$\begin{bmatrix} 1 & & & & & \\ ? & 1 & & & & \\ ? & ? & 1 & & & \\ ? & ? & ? & 1 & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ ? & ? & ? & ? & \cdots & 1 \end{bmatrix}$$

- Understanding the problem: if `matrix[i][j] == 1` and $i \neq j$, it means that i and j are friends.
- Given any x, our task: find out how many integers $i \neq x$ satisfy `matrix[x][i] == 1` or `matrix[i][x] == 1`
- What does the indices tell you about the "search direction"?