CS 137

PART 1

INTRO

#include <stdio.h> ⇒ Header file w/ preprocessing command
(preprocess)

```
int main () {
    printf ("Hello world \n"); ⇒ Print output
                                  ↳ Newline
    return 0;
}
```

↳ Succesful ⇒ 0 or EXIT_SUCCESS

COMPILER

```
gcc my_file.c ⇒ Compile
./a.out       ⇒ execution
```

EUCLIDEAN ALGO

Pseudocode:

1. Find bigger num
2. Find remainder (a % b)
3. Make a = b, b = (a % b) → should be before updat
4. Repeat until b=0, a = gcd

Iterative:

```
While (b != 0) {
    r = a % b
    a = b
    b = r
}
return a
```

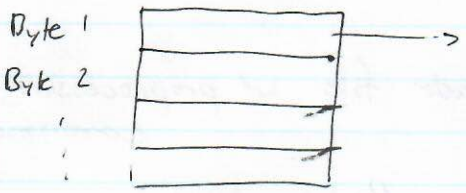
Recursive:

```
int recursive (int a, int b) {
    if (b == 0)
        return a;
    else {
        swap = a
        b = swap % b   ← a = b
        return recursive (a, b);
    }
```

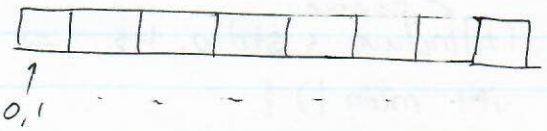
INPUT / OUTPUT

```
printf ("...", variable)
    ↳ Variable input: %d, %lu, %c
scanf ("...", &variable);
        ↑ format input
```

MEMORY



1 byte = 8 bits:



Representation: $(2^{bits} - 1, 0)$

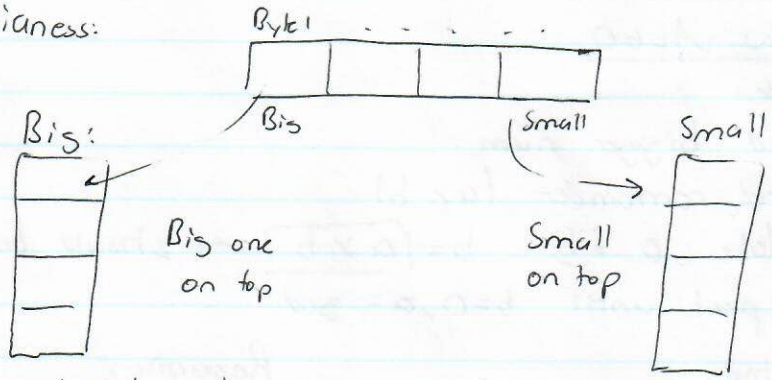
- Conversion:

  Decimal → binary: split into power of 2:

  $123_{10} = (10^0 \times 3) + (10^1 \times 2) + (10^2 \times 1)$

  $123_{10} \Rightarrow 123_2 \Rightarrow (1\times2^6) + (1\times2^5) + (1\times2^4) + (1\times2^3) + (0\times2^2) + (1\times2^1) + (1\times2^0) = 1111011_2$

- Endianess:



- Signed int: binary way of notating ⊕ and ⊖

  0100 ↳ ⊕ 1100 ↳ ⊖

  - Short cut of converting from ⊕ to ⊖ representation (negation)
    1. Write in binary
    2. Find right-most 1 bit
    3. Invert all bits to left of that

- Binary arithmetic:
  - Ex: Do $-4-3$ in binary:

    $4 = 0000\ 0100 \Rightarrow -4: 1111\ 1100$

    $3 = 0000\ 0011 \Rightarrow -3: 1111\ 1101$

    $'1111\ '1100$

    $+\ 1111\ 1101$

    $1111\ 1001 \Rightarrow$ Throw away last carry over

    $-(1111\ 1001) = 0000\ 0111$

- Overflow = undefined behaviour (underflow)
  ↳ Know limits of variables (eg. unsigned > 0)

OPERATIONS

BEDMAS: left associative
- ↳ 6 - 2 % 3 = (6-2) % 3 (evaluates left expression)

Assignment: right associative (/=, -=, +=)

Incremental:
- ↳ Prefix: ++var ⇒ increment then use
- ↳ Suffix: var++ ⇒ use it, then increment

} Effect on while loop

Relational: <, <=
Equality: ==, !=
Logical: &&, ||

} Boolean

Bitwise operators:

a = 101100 , b = 001110

NOT: ~a : 010011

AND: a & b : 001100 (1 if both 1)

OR: a | b : 101110 (1 if either 1)

XOR: a ^ b : 100010 ⇒ (1 if strictly either 1)

Left shift: a >> 2 : 001011

Right shift: a << 3 : 101100000

IF / ELSE

Dangling:
```
if ( - - )
    if (- - )
    else (- - )
```
} b/c of lack of brackets

Ternary: boolean_expr ? val1 : val2
(true ↑, false ↑)

Switch:
```
switch (expr) {
    case expr_value: . . . break;
    case expr_value: . . . break;   ← If no break, will continue on next case.
    :
    default: . . . break;
```

PART 2

LOOPS:

1. while (condition) {- - }
2. do { - - } while (condition); ⇒ Executes at least once
3. for (initializer; condition; update) { - - }

- Commands:
  1. break; ⇒ exit loop
  2. continue; ⇒ skips to next iteration
  3. goto.

lable: . . . .
goto lable
} Looping potential

FINDING PRIMES

1. Read in number
2. If $n \leq 1$ ⇒ Not prime
3. Else:

```
while (div * dN ≤ number) ⇒ dN = 2 ⟶ √n
    if (n % dN == 0)
        Not prime; break;
    else:
        dN ++;
if (dv * dN ≤ number) {
    Not prime;
} else { prime }
```

} If the number is a prime, div ~~dN~~ > $\sqrt{n}$ by while loop increment

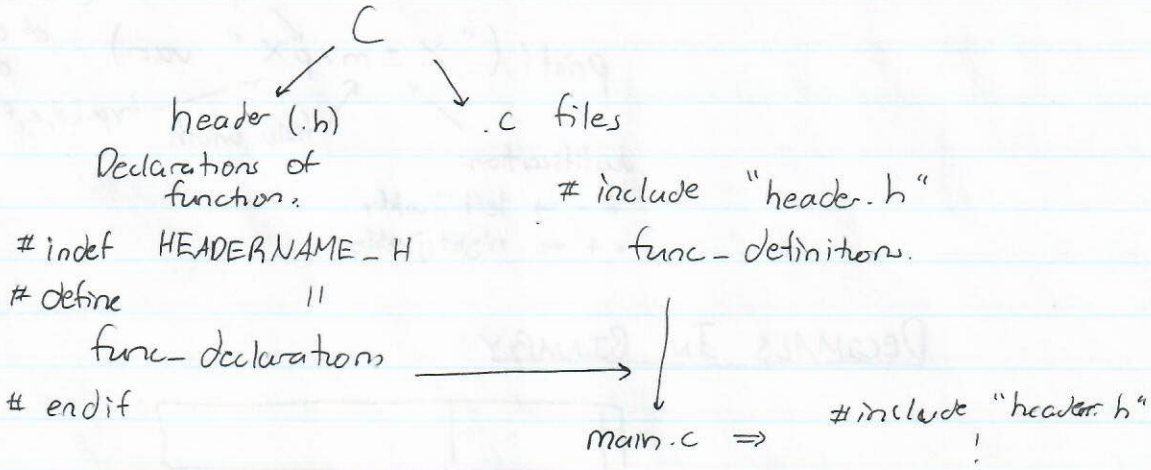
FUNCTIONS

declaration

return_type func_name (parameter) { - - }

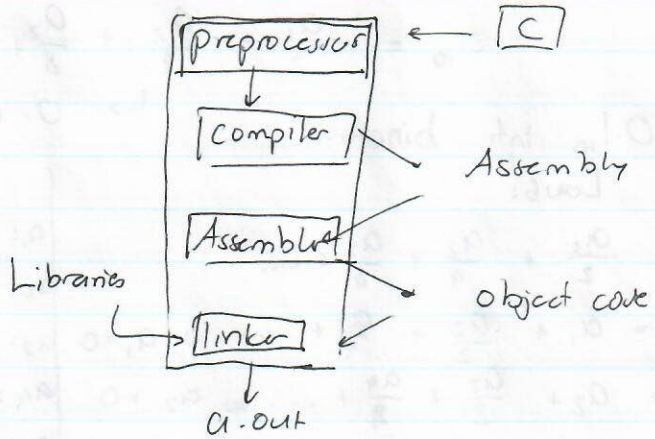
- Function stack is gone when it hits 'return'
- Pass by values: aside from array, not changing variables

Modules



- Compilation:

gcc -o power power.out c-files ⇒ gcc main.c header.c



Macros

#define PI 3.1415...
↳ Constant name

Recursion

- Iterative → recursive (useful if no while, for, do-while, goto allowed)
- 2 rules:
  1. Base case (0, 1 ...)
  2. Progression towards base case.

PART 3

```
printf( "%.±m.pX", var)
```

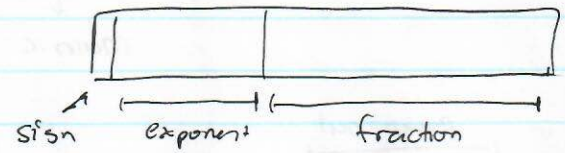
precision (# of decipoints (e-f), # of digits (d), # of sig digs (g))

type (d, e, f, g)

field width

Justification:
- - ⇒ left justify
- + ⇒ right justify

DECIMALS IN BINARY



$$(-1)^{sign} \cdot fraction \cdot 2^{exponent}$$

$$a_{10} = \frac{a_1}{2} + \frac{a_2}{4} + \frac{a_3}{8} + \dots$$

↳ $0.a_1 a_2 a_3 \dots_2$

- Ex: // 0.1_{10} into binary:

Long:

$$0.1 = \frac{a_1}{2} + \frac{a_2}{4} + \frac{a_3}{8} + \dots$$

×2 ↓

$$0.2 = a_1 + \frac{a_2}{2} + \frac{a_3}{4} + \dots \Rightarrow a_1 = 0$$

×2 ↓

$$0.4 = a_2 + \frac{a_3}{2} + \frac{a_4}{4} + \dots \Rightarrow a_2 = 0$$

⋮

Notice repeating pattern

Easy:

a_1: $0.1 \times 2 = 0.2 \Rightarrow a_1 = 0$

a_2: $0.2 \times 2 = 0.4 \Rightarrow a_2 = 0$

a_3: $0.4 \times 2 = 0.8 \Rightarrow a_3 = 0$

a_4: $0.8 \times 2 = 1.6 \Rightarrow a_4 = 1$

a_5: $0.6 \times 2 = 1.2 \Rightarrow a_5 = 1$

0.2

$$0.1_{10} = 0.0\overline{0011}_2$$

ERRORS

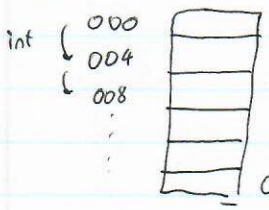
Absolute: $|decimal - actual|$

Relative: $\frac{|decimal - actual|}{|actual|}$

- Comparisons + equality check is no good for decimals
  ↳ Use tolerance/epsilon: $x == y \Rightarrow (x - y) < \varepsilon$
- Division:

$\frac{float}{int}$ or $\frac{int}{float}$ or $\frac{(float)a}{b}$ ← typecasting

ARRAYS




```
int arr[5] = { . . . }
int [] = { . . . }
int [5] = {1, 2, 3}  (last 2 = 0)
int [5] = {0}
```

```
int arr[5] = {[1]=4, [2]=5}
```

Size
- sizeof (~~array~~ var) = # of bytes of that var
  - ↳ sizeof (int) = 4
- # of elements in array: sizeof (array) / sizeof (element_in_array)

Passing:

Most things pass by value. Not array: pass by reference

func (array) ⇒ function gets the address of the array

↳ func (a[]) ⇒ a is address to 1st element in arr.

Pointer decay: reverting from full array in main to just memory address in function ⇒ lose information (size)

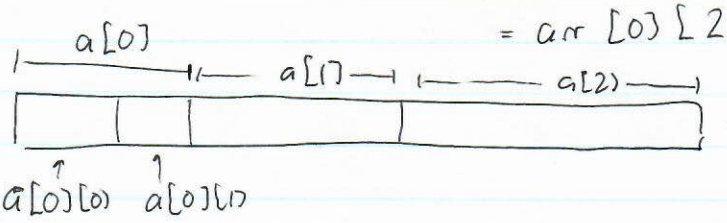
Copying: copy value 1 by 1 into new array. (a[0] returns value)

MULTIDIMENSIONAL ARRAYS

int arr [][2] = {{3, 4}, {4, 3}, {5, 9}} ⇒ 3×2 array

↳ 9: arr [2][1] OR arr [0][row × sizeof (row) + col]

= arr [0][2×2 + 1]



⇒ Reason behind why this works!

MATH.H

fabs(x) , pow(x,y) $\Rightarrow x^y$, log(x) $\Rightarrow \ln x$

ROOT FINDING

① Bisection method:

1. Get some a and b s.t. $f(a) \gtrless 0$ and $f(b) \lessgtr 0$
2. Find midpoint: $m = \frac{a+b}{2}$
3. If $f(m) < 0$: $b = m$
4. If $f(m) \gtrless 0$: $a = m$
5. Loop until $|m_{n-1} - m_n| < \varepsilon$ / # of iterations.

② Fixed point:

Works if looking at function: $g(x_0) = x_0 \Rightarrow$ Fixed point

↳ Ex: Root of $x - \cos(x) = 0$, same thing as $\cos(x) = x$

1. Find $g(x_0) = x_1$
2. If $|x_1 - x_0| < \varepsilon$, return x_1
3. Else: repeat with $x_0 = x_1$

FUNCTION POINTERS

Declaration of function pointed to: normal

Declaration of function using pointer:

return_type func_name(..., return_type (*name)(param_type...)

Calling: give function name.

Using in function: use name as given in param_list.

POLYNOMIALS

Represent as array of coefficients.

Computation:

1. Traditional: $x \xrightarrow{n \text{ times}} x^n \xrightarrow{n \text{ times}} ax^n \rightarrow$ add $\Rightarrow$ # of x: $2n-1$. # of +: n
2. Horner: $2 + 9x + 4x^2 + 3x^3 = 2 + x(9 + x(4 + 3x)) \Rightarrow n$ multi + add

```
for (i = n-2; i >= 0; i--) {
    y = y*x + p[i];   ← x is evaluating, p[i] is coefficient.
}                       y initial = p[n-1];
return y
```

PART 4

STRUCTURES

```
struct struct_name {
    int var1;
    int var2;
};
⋮
struct struct_name instance_name = {__, __};
                                 = {.var1 = __};
instance_name.var2 ⇐ Access
```

struct ⟶ function (pass by value!)

- Simplify:

```
typedef struct { . . . } name;
⋮
name instance_name = { . . . };
```

- Pointer: *p = &struct1

  (p → var1) ⇒ Access to element in structure pointer (no dereference)

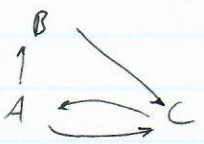
  ↳ Equivalent to (*p).var1

PAGE RANK

$$R(P) = \sum_{i=1}^{M} \frac{r(Q_i)}{|Q_i|} \Rightarrow \begin{array}{l} M: \text{number of linking pages} \\ Q_i: \text{a linking page} \\ |Q_i|: \text{\# of outbound links of } Q_i \end{array}$$

⟵ Normalize to 1 ⟶

- Ex: //



Find rank(A), R(B), R(C):

①: Write down equations:

①: $R(A) = \frac{R(C)}{1}$ ③: $R(C) = R(B) + \frac{R(A)}{2}$

②: $R(B) = \frac{R(A)}{2}$ ④: $R(A) + R(B) + R(C) = 1$

②: Make all ranks in terms of 1:

①: $R(B) = \frac{R(C)}{2}$

∴ ④: $R(C) + \frac{R(C)}{2} + R(C) = 1 \Rightarrow \begin{array}{l} R(C) = \frac{2}{5} \\ R(A) = \frac{2}{5} \end{array}$ $R(B) = \frac{1}{5}$

- Random surfer model:

$$P(R) = \frac{1-\delta}{N} + \delta \sum_{i=1}^{M} \frac{r(Q_i)}{|Q_i|}$$

→ δ: P(escaping page)
↳ Can be initialized

N: # of pages in net

  - Jacobi's Method:
    1. Set all rank of pages equal
    2. Use random surfer model to converge to rank faster

- Sinks:

  - Fix:

$$P(R) = \frac{1-\delta}{N} + \delta \sum_{i=1}^{M} \frac{r(Q_i)}{|Q_i|} + \delta \sum_{i=1}^{M_0} \frac{r(S_i)}{N}$$

(M_0 ← # of sinks; N ↖ # of pages)

⟵ sink addition ⟶

PART 5

POINTERS

```
type-var-pointing  *point_var = var_pointing-to;
```

↙ point_var: address
↳ Memory address of variable I'm pointing to you

↘ dereference: *point_var: value
↳ Gets value of thing it is pointing to.

- Arithmetic: $a + m \quad (m \in \mathbb{Z})$
  - Starting from a and moving m bytes (# of bytes depends on type of a)

$a[8] = \{2, 3, \overset{①}{4}, 5, 6, 7, \overset{②}{8}, 9\}$ (③, ④)

```
int *p, *q;
```

① : p = &(a[2])
② : q = p + 4
③ : p += 4
④ : *p = 10

* However:
a = &a[0] = 100
a + 1 = 104 (4 bytes / int)
&a + 1 = 136 (whole array + 1)

- Array pointer:

$a[8] = \{1, 2, 3, 4, 5, 6, 7, 8\}$

a[0] a[1] a[2] ... ...

a → 1, a+3 → 4, a+5 → 7

Combining ++ and *:

*++p ⇒ *(++p) (Increment p then use *p)

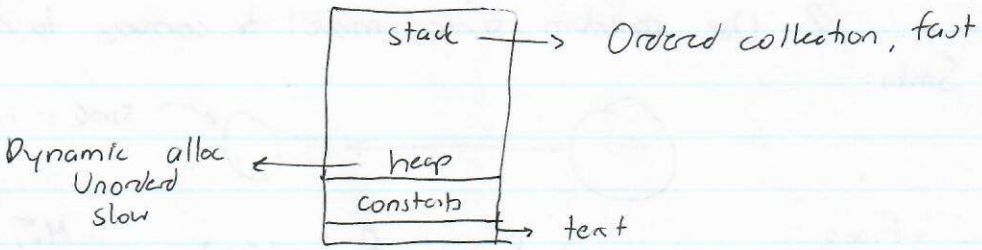
*p++ ⇒ *(p++) (Use *p then increment p)

(*p)++ ⇒ (Use *p then increment *p)

Know 2D array pointer: **a → a[i][j] = *(*(arr+i)+j)

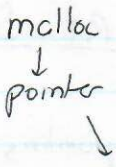
= *(a + (i×no-cols + j))

MEMORY MANAGEMENT



#include <stdlib.h>

- void *malloc (size_t size): allocates 'size' number of bytes from heap → pointer.



Useful | null ⇒ Null pointer should be checked.

- void free (void *): free memory allocated to pointer.
  - ↳ Memory leaks: computer runs out of memory bcuz no memory was freed.
- void *calloc (nmemb, size): allocates nmemb elements of 'size' bytes to 0
- void *realloc (void *p, size): reallocate old block to new block.

DYNAMIC ARRAYS

Structure malloc: struct struct_name * var = malloc (sizeof (struct st. name

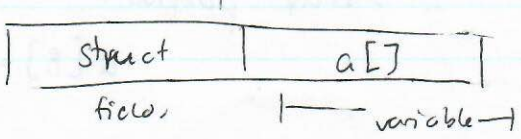
↳ Access fields: (*var).field OR var → field

Struct hack:

1.
```
struct name {
    field1 = ...
    field2 = ...
    :
    int a[];
};
```
end must be array

2. Malloc:

malloc (sizeof(struct) + array_size * sizeof(int))



- Vector implementation:

1. struct vector {
   int *arr;
   int size, length;
   };

2. Creation:
   struct vector *v = malloc (sizeof (struct vector))
   v → size = . .
   v → arr = malloc (sizeof (int) * 4)

3. Deletion:
   free (v → arr);
   free (v);
   } Must be in this order: will have leak.

4. Add:
   if (length == size):
   newSize = size * 2
   int *newArr = malloc (sizeof (int) * newSize)
   } New array w/ 2x size
   for (int i = 0; i < ~~oldSize~~ size; i++)
   update newArr w/ old arr elements
   } Copying
   newArr [v → size] = newValue
   free (v → arr)
   v → size = newSize
   v → arr = newArr.
   } Updating
   else:
   v → arr [v → length] = value
   ++ v → length

PART 6

CHARACTERS

char firstChar = 'a';

- ASCII:
  - 48-57: Numbers. ⇒ Convert character # to real number via num - '0'
  - 65-90: ~~lower~~ upper case letters
  - 97-122: lower case letters.
  } Corresponding character is 32 away.

- Convert 'a' to 'A'.
  'a' - 32 ⇒ 'A'
- You can add numbers to letters in chars.
  ↳ Numbers can also be represented by characters!
  'a' - ' ' ⇒ 'A'

STRINGS

1. char s[] = "Hello";
2. char s[] = {'H', 'e', 'l', 'l', 'o', '\0'}; (really important)
   ↳ Checking if string is traversed:
   s[i] = '\0'
3. char *s = "Hello"
   ↳ String literal: cannot change (undef. behavior)
   ↳ char *s = "Hello" "World!" ⇒ combines into 1
   ↳ sizeof(s) is not always = sizeof("...")

CONSTANTS

const ... ⇒ immutable

const int i = 10

Only way to change:

```
const int i = 10
int *a = &i
*a = 3
```

} Undefined

Know the differences between constants + macros

const int *p
OR int const *p

p is a pointer to a constant int
↳ Can change address, but cannot change numb.

Let r: int *r;
r = p ⇒ error
r = (int *)p ⇒ Fine
*r = 3 ⇒ Fine

int * const q:

Cannot change address
↳ Pointer constant.

STRING LIBRARIES

#<strings.h>

size_t strlen(const char *) : returns length of string (w/out null)

↳ Compare to sizeof(char [7]) ⇒ size including null.
sizeof(char *) ⇒ size of address.

char *strcpy(char *s0, const char *s1):

↳ Copies s1 → s0. s0 might not == sizeof(s1) ⇒ undef. behavior
↳ Includes '\0'

char *strncpy(char *s0, const char *s1, size_t n):

↳ Copies first n characters from s1 → s0
↳ If strlen(s1) < n null padded.
↳ No null character added!

char *strcat(char *s0, const char *s1):

↳ Concatenates s1 → s0
↳ Make sure s0 has enough space.

char *strncat(char *s0, const char *s1, size_t n):

↳ Concatenates first n characters from s1 → s0.
↳ Adds '\0' after.

~~char~~ int strcmp(const char *s0, const char *s1):

↳ <0: s0[i] < s1[i] || all equal except s1 is bigger.
>0: s0[i] > s[i] || all equal except s0 is bigger.
=0: completely equal.

```
char c[] = "hello"
char d[] = "hello"
if(c == d) { "!" }   ⇐ Compares addresses! Need strcmp!
else { "not equal" }
```

Printing strings: printf("%s", char[])

Get strings: scanf("%s", &var) ⇒ stops at whitespace
gets("%s", &var) ⇒ stops at \n

Hilroy

Other String Functions

void * memcopy (void *restrict s1, const void* restrict s2, size_t n)

Copying n bytes from s2 to s1

- "restrict" ⇒ no other pointer can access array/memory.
  - ↳ Overlap not allowed
- Always remember to copy '\0' at end of string.

void * memmove (void * s1, void * s2, size_t n)

Same thing as memcpy, s1 + s2 bytes can overlap

int memcmp (void *s1, const void * s2, size_t n)

Compares bytes of memory (similar to strcmp)

↳ Only looks at first n bytes.

< 0	= 0	> 0
↓	↓	↓
1st byte that does not match is smaller in s1	equal	1st byte that does not match is greater than s1.

void * memset (void * s, int c, size_t n)

Fills n bytes of s with byte c (converted to char)

Unicode

1 Character = 21 bits!

Divided into 21 planes (Plane 0 is Basic Multilingual Plane / BMP)

PART 7.

BIG OH

$g(x) \in O(f(x)) \Rightarrow g(x) \leq c f(x)$ $(c \in \mathbb{R})$ where $x \geq X$

$g(x)$ has same, or lower, growth rate as $f(x)$ and I can multiply $f(x)$ by some number s.t. the function will be bigger than $g(x)$ at after/or a certain point.

↳ $g(x) \in O(f(x))$ if $g(x)$ grows at equal/slower rate.

Asymptotic growth: look at values $\to \infty$ (highest power term matters)

↳ $\lim_{x \to \infty} \frac{g(x)}{f(x)} < \infty \Rightarrow g(x) \in O(f(x))$

$\lim_{x \to \infty} \frac{g(x)}{f(x)} = \infty \Rightarrow f(x) \in O(g(x))$

Logs: bases don't matter:

$$\log_a x = \frac{\log_b x}{\log_b a} = \boxed{\frac{1}{\log_b a}} \log_b x$$

↙ constant: no effect for $x \to \infty$

Properties: Let $g_0(x) \in O(f_0(x))$ and $g_1(x) \in O(f_1(x))$

1. $g_0(x) + g_1(x) \in O(f_0(x) + f_1(x))$
2. $g_0(x) \cdot g_1(x) \in O(f_0(x) \cdot f_1(x))$
3. $O(f_0(x) + f_1(x)) \in O(\max\{f_0(x), f_1(x)\})$

Transistivity: $h(x) \in O(g(x)) \wedge g(x) \in O(f(x)) \Rightarrow h(x) \in O(f(x))$

Growth rates:

$$1 \ll \log n \ll n^2 \leq c^n \ll n! \ll n^n$$

Others:

- Little-o: $o(f(x))$: $<$
- Big omega: $\Omega(f(x))$: $\geq$
- Little omega: $\omega(f(x))$: $>$
- Big Theta: $\Theta(f(x))$: $=$

Tips: look @ how many times code is run

SEARCHING

Linear search:

```
int lin_search (int a[], int n, int value) {
  for (int i = 0; i < n; ++i) {
    if (a[i] == value) {
      return i;
    }
  }
  return; }
```

Analysis:

Best case: value at beginning ⇒ $O(1)$ (add + compare)

Worst case: value at end ⇒ # of ops = $1 + (1+2+1)n + 1$ $\in O(n)$

Average case: $O(n)$

Binary search:

~~int bin_search (int a[], int n, int value) {~~
~~int beg = 0, end = n-1;~~
~~for{ while ((a[$\frac{end-beg}{2}$] != value~~

```
int bin_search (int a[], int n, int value) {
  int left = 0, int right = n-1;
  while (left <= right) {
    int middle = left + (right-1)/2
    if (a[middle] == value)
      return m;
    if (a[middle] < value)
      left = middle + 1;
    else
      right = middle - 1;
  }
  return -1.
```

⇒ Worst case: $O(\log n)$

Requires sorted list

SORTING

- Selection sort:
  1. Finds the smallest element
  2. Swaps to front
  3. Repeat w/ array

```
void selection_sort (int a[], int n) {
  for (int i = 0; i < n; ++i) {
    int min = i
    for (int j = i+1; j < n; ++j) {
      if (a[j] < a[min]) min = j;
    }
    int temp = a[i];
    a[i] = a[min];
    a[min] = temp;
  }
}
```

→ Find smallest element above i

} Swapping.

- Time complexity: $O(n^2)$ in worst case, best case + average case.
- Problem: checking constantly if array is sorted, in a sense. (inner loop)

- Insertion sort:
  1. For all x, find where x should go
  2. Shift elements less than x
  3. Insert x + repeat

→ 1 pass of whole array!

```
void insertion_sort (int * a, int n) {
  for (i = 0; i < n; i++) {
    x = a[i];
    for (j = i; j > 0 && x < a[j-1]; j--) {
      a[j] = a[j-1];
    }
    a[j] = x;
  }
}
```

Iterating as long as element before is larger than x.

← Duplicating value to current position

We can just simply drop in x.

- Time complexity: $O(n^2)$ in worst case. $O(n)$ in best ca.. as inner loop never executed if sorted.

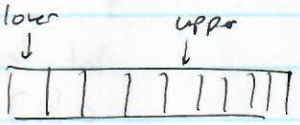
Hilroy

- Merge sort:
  1. Keep dividing array in half
  2. Sort recursively
  3. Combine

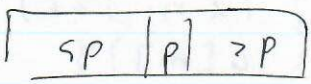
  } O(nlogn) for all cases.
  ↳ n for comparison, $\log_2 n$ merge.

- Functions:

```
void main-merge (int a[], int n) {
    int *t = malloc (n * sizeof (int));   ⇐ Creating temp. array to store values.
    merge-sort (a, t, n);
    free (t);
}
void merge_sort (int a[], int t[], int n) {
    if (n <= 1) return;
    int middle = n/2
    int *lower = a
    int * upper = a + middle
    // Take subarrays + sort
    merge-sort (lower, t, middle);
    merge-sort (upper, t, n-middle);
    // Combine:
     int i = 0;
     int j = middle;
     int k = 0;   ⇒ temp index.
    // Going through both subarrays + placing in temp.
     while (i < middle && j < n) {
        if (a[i] < a[j]) t[k++] = a[i++];
        else t[k++] = a[j++];
     }
    // Putting in any remaining elements.
     while (i < middle) { t[k++] = a[i++]; }
     while (j < n) { t[k++] = a[j++]; }
    // Copy into actual array.
     for (int i=0; i<n; i++) a[i] = t[i];
}
```



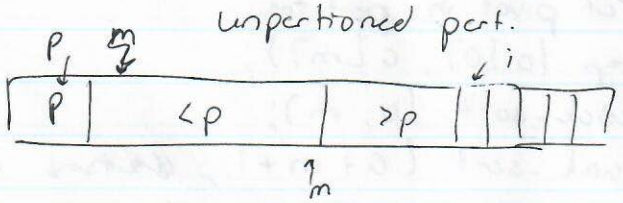
- Quick sort:
  1. Pick a pivot in array
  2. Split array into partitions:



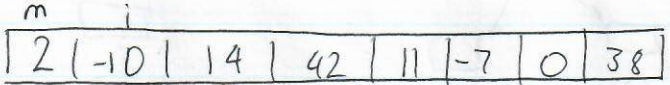
  3. Sort 2 partitions.

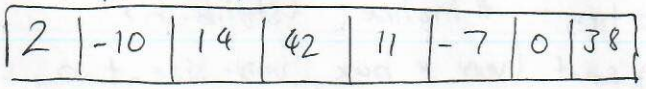
Space is conserved! No new arrays are created.

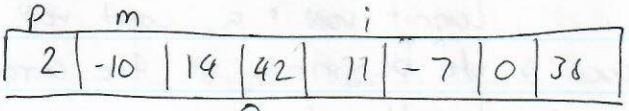
- Lomuto Partition:
  - Pivot is 1st element.
  - Index variables: m (last index of partition <p), i is 1st index of unpartioned part.

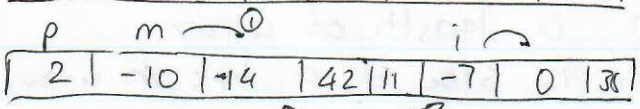


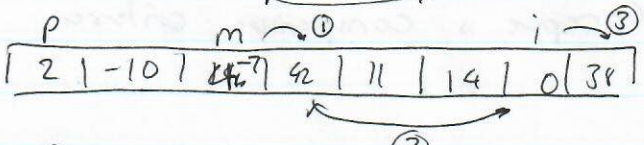
  - Steps:
    1. If a[i] < p: put into '<p' portion. Increment m, swap a[m] ↔ a[i], increment i
    2. If a[i] ≥ p: put into '>p' portion. Increment i.
    3. End: if i == n, then a[m] ↔ a[0]. Partition in middle.
- Ex: //

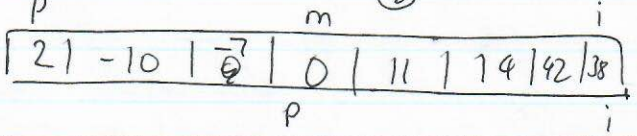
1.


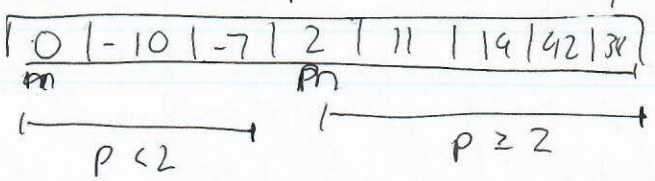
2.


3.


4.


5.


6.


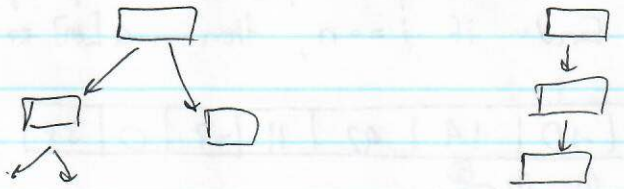
7.


⇒ Recurs. Hilroy

```
void quick_sort (int a[], int n) {
  if (n <= 1) return;
  int m = 0;
  for (int i = 1; i < n; i++) {
    if (a[i] < a[m]) {
      ++m;
      swap (a[i], a[m]);
    }
  }
  // Put pivot in partition.
  swap (a[0], a[m]);
  quick-sort (a, m);
  quick-sort (a+m+1, n-m-1);
```

- Time complexity: average + best case: $O(n \log n)$. Worst $O(n^2)$
- Space: long list of calls on stack.

- Tail recursion: have 1 recursive call @ end of function



- Built in sorting: `#include <stdlib.h>`

```
void qsort (void * base, size_t n, size_t size, int(*compar)
            (const void * a, const void * b));
```

- Base is the beginning of the array
- n is length of array
- Size is size of a byte in array
- Compar = comparison criteria.

TAIL RECURSION

- Fibonacci:

$$f(n) = f(n-1) + f(n-2)$$

Iteratively → tail recursion.

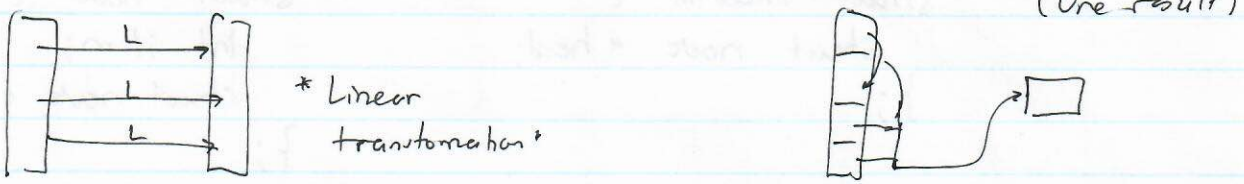
↳ Pass a counter:

```
int fib (int n) {
    if n == 0 return 0;
    return helper (0, 1, n);
}
int helper (int a, int b, int n);
    if n == 0 return a;
    return helper (a+b, a, n-1);
}
```

(a+b: store, a: storage, n-1: counter)

HIGHER-ORDER FUNCTIONS

void ____ (____, ____, function)



Mapping function:

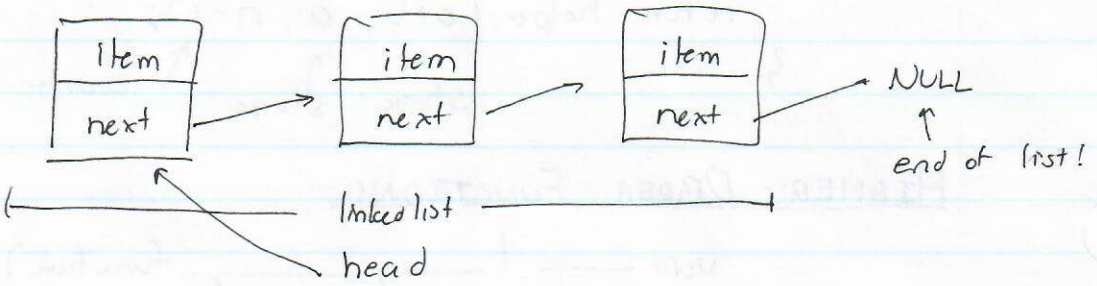
```
void map (void *src, size_t n, size_t srcb, void *dest, size_t
        *destb, void (*f)(void *, void *)) {
    for (size_t i = 0; i < n; i++) {
        f (src, dest);
        src += srcb
        dest += destb
    }
}
```

(# of elements; size of 1 byte (increment); Increments up to next array element)

Reducing function.

```
void reduce (void *src, size_t n, size_t srcb, void *dest, void (*f)(void *, void*))
{
    if (n==1) {
        f(src, dest);
        return;
    }
    reduce (src+srcb, n-1, dest, f);
    f (src, dest);
}
```

LINKED LIST



Implementation:

1. Linked list structure:

```
struct linkedlist {
    struct node *head;
};
```

2. Node structure.

```
struct node {
    int item;
    struct node * next;
};
```

* Do not lose pointer references! Be careful of order of operation *
Important technique: have pointer to previous + current data nodes.

Re

- Polynomial linked list

```
struct llnode {
    void * data;
    void struct llnode * next; };
struct ll {
    void struct llnode * head; };
```

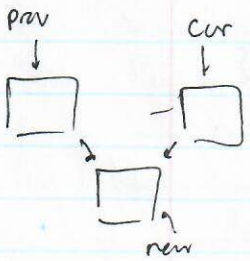
1. Create:

```
struct ll * llcreate () {
    struct ll *new = malloc (sizeof (struct ll));
    new -> head = NULL;  return new;
}
```

2. Adding elements.

```
void addToFront (struct ll * l, void * element) {
    struct llnode *new = malloc (sizeof (struct llnode));
    llnode new -> next = l -> head;
    new -> data = element;
    ll -> head = new;
    return;
}
```

```
void addInOrder (struct ll * l, void * element, int (*cmp) (void *, void *))
{
    struct llnode * cur = ll -> head;   ← Next point
    struct llnode * prev = NULL;        ← Previous point
    for (; cur && cmp (cur -> data, element) < 0; prev = cur;
          cur = cur -> next) // Finds spot where cur -> elem.
```



Is it in front? ⇒ `if (! prev) addToFront (l, element);`

Addin elem.

```
    struct llnode * new = malloc (sizeof (struct llnode));
    new -> ~~str~~ next = cur;
    new -> data = element;
    prev -> next = new;
}
```

3. Copying.

Creating new structure

```
struct ll * copy (struct ll * src) {
    struct ll *new = malloc (sizeof (struct ll));
    new -> head = copy_helper (src -> head);
    return new;
}
```

Recursive definition useful!

```
struct llnode * copy_helper (struct llnode * src) {
    if (!src) return NULL;
    struct llnode *new = malloc (sizeof (llnode));
    * new -> next = copy_helper (src -> next);
    new -> data = copy_int (src -> data);
    return new;
}
void * copy_int (void *data);
    int *src = malloc (sizeof (int));
    *src = * (int *) data;
    return src;
}
```

4. Deletion.

```
void delete_ll (struct ll *l) {
    delete_ll_node (ll -> head);
    free (l);
}
void delete_ll_node (struct llnode * node) {
    if (!node) free (node); return;
    delete (node -> next);
    free (node);
    return;
}
```