CSE 13: Lecture 1/07/2020

Source Code Control system
- tracks files and the changes to those files
- collaboration tool (working remotely, coding w/ others)
- when you lose a file, lose a device, easy to save

- What is ssh?
  - ssh - secure shell
  - scp - secure copy
  - sftp - secure FTP

  } family of programs that allow computers to communicate w/ each other

- public key cryptology (public & private key)
- ssh-keygen -b 384 -t rsa (creating an ssh key)
- you don't risk anything when showing your public key
- choose very large primes p & q

$n = p \times q$

e is a random prime

$e \times d = 1 (mod) (p-1)(q-1)$

Encryption: $E(m) = m^e \bmod n = c$ $\quad E(D(x)) = D(E(x)) = x$

Decryption: $D(c) = c^d \bmod n = m$

Git is the main source control system (most widely used)
- keeps multiple versions of your files
- can share files w/ others in a controlled fashion
- files are stored in a repository
  - stores are opaque chunks
  - repo represents the history of each file
- have a local repo that can synchronize w/ the one stored on the server

- git pull brings your local repo up-to-date w/ remote repo
- git pull --merge allows you to combine 2 or more branches
- git pull --rebase brings in the history from the remote repo & applies them to your local repository, overwrites local repo

In C, you must declare a variable before using it
for now, we are only concerned about the scalar types: char, int, float, double

```
char *s,c;
int i;
float f;
double d;
```

Each pair of curly braces introduces what is called a scope
The scope of the variable tells us where the variable exists, or is defined

```
clang -Wall -Wextra -Werror -Wpedantic -o hello hello.c
```

if nothing comes up, then it was successful
to run the program, ./hello
cd - command to navigate to home directory
mv - short for move, used for renaming files

```
clang-format -i -style=file hello.c
```

↳ format hello.c
in place

diff - compare two files

```
git add hello.c
git commit -m "Final submission"
git ~~commit~~ --global user.email "mnguy181@ucsc.edu"
    config
```

To get the commit ID, use git log, first one is latest commit
↳ submit to canvas

rand() % 6 [0-5] 10 midnight = 5+5 = 10

What we need: 0 1

- array of names, lives ["Alec", "Brec", ...]

vampires can roll on the same round they were resurrected

Dead vampire does not roll

always in this order

What we need: 0 1 2

- array of names ["Alec", "Bree", ...]
- array of lives
- loop for each round
- track lowest roll and who rolled it (save person who rolled it first)
- loop for each game w/ exit condition

int left(position, players)

return position left of position

Design.pdf

- design
- structure
  - flowcharts
  - pseudocode
- updates
- summary of what your program is going to do
  - rules

cp ~/cse13s/resources/asgn1/ {Makefile,names.h} .

vi vampire.c

#include "names.h" #include <stdlib.h>

#include <stdio.h> #include <inttypes.h>

./vampire.c

printf("Rolls: %ld \n", random() % 6); → roll

int roll(void) {

return random() % 6;

}

```
unsigned seed = 0;
printf("Random seed: ");                    SCNu32  PRId32
scanf("%lu", &seed);
if (scanf("%u", &seed) != 1) {
        fprintf(stderr, "Invalid random seed\n");
        exit(1);
}
uint32_t seed = 0

    if (seed < 0 || seed > UINT32_MAX) {
        fprintf(stderr, "Invalid random seed\n");
        exit(1);

int left(int pos, int players) {
    return (pos-1) % players;
}
int right(~~pos+1 % players;~~ int pos, int players) {
    return (pos +1) % players;
}
```

Lecture Notes 1/19/2021:

- functions in C may or may not return a value
- if a function has a side-effect, make sure to note it
- in other languages, we may call them procedures or subroutines, methods (Java)
- function if it returns a value, procedure or subroutine if it does not
- function: block of code that performs a task
  - defined once
  - must be declared before they are used (declared before main())
  - programs can declare & call a function as many times as desired
- main() is a special function and is run when the program starts
- other functions are subordinate to main()
- functions should define abstractions, hide implementation, give names to seq. of code
- use functions to refactor repeated seq. of code, simplify code

```
return type function_name(parameters)   <- function head
{
    //declarations, assignment statements   <- function block/code
}
```

- The functions return a value, value may be void, can return any scalar value
- you cannot return an array, but you can return a pointer
- function name can't start w/ number or any punctuation other than _ or $
- snake case in this class for function names
- C uses call-by-value, except for arrays, because of relation to pointers
- Formal parameters
  - name of parameter as it is used inside of function
- Actual parameter
  - name of value passed to the function
- the called function copies values of supplied actual parameters which are pushed onto the call stack

- #define , a preprocessor directive that defines a macro for the program
- conditional directives , set of preprocessor directives that use conditional statements to include code selectively
- header files should only contain things that are shared between source files: function declarations, macro definitions, data structure, and enumeration definitions , global variable
- file extension .h
- typically used for modules, abstract data types
- standard header files: stdio.h for input/output, inttypes.h
- extern typically used for global variables, declared outside of function
- static can be declared inside and outside a function
  - used to persist/exist across function calls
  - used inside a file, so different from global variable
- array: homogeneous collection of elements, all elements have the same type, vector
- arrays start at zero a[0] and end at a[n-1]
- declaring an array: `int a[] = {1, 2, 3, 4};`
  `int b[10];` ten empty slots (usually zero , depending on where declared)
- arrays are the exception to the rule that parameters in C are always passed by value
- name of the array is just a pointer to the first element of the array
- & (address of) operator: gives the memory location of variable
- size of operator gives number of bytes used by a variable
- works on arrays, structures, unions, when compiler can know how much memory is used, use this operator carefully

Lecture 1/21/2021:

- a string is an array of characters that ends in '\0'
- declared as:

```
char s[] = "Hello World!"
char *s = "Goodbye cruel world!"
char s[] = {'L', 'e', 'g', 'a', 'l', 0}
```

- most people use the *s version
- Stacks:
  - objects added in LIFO order
  - capacity of a stack is the number of stack elements that will fit
  - capacity can be increased as number of elements increases
- pop() removes the element @ the top of the stack
- stacks can be implemented using linked lists
- queues are implemented using FIFO, can be implemented w/ arrays or linked lists

Assignment 2 Notes:

- implementing 4 Taylor-series (sin, cos, tan, exp) $\tan = \frac{\sin}{\cos}$
- Newton's method for log(x)
- will be centering our Taylor series around 0
- may not use any functions from <math.h>

$$\frac{x^k}{k!} = \frac{x^{k-1}}{(k-1)!} \times \frac{x}{k}$$

@ each step, we just need to compute $\frac{x}{k}$, starting with *$k = 0!$ where $0! = 1$ and multiply it by a previous value and add it to the total, turning it into a for or while loop

```
new = previous * current;
previous = current;
```

Set Epsilon = 10^{-14} for this assignment

- stop adding when smaller than epsilon

mathlib.c

```
#include "mathlib.h"
#define EPSILON 1e-14

double Sqrt(double y) {
    double x_n = 1.0;
    double old = 0.0;
    while (Abs(x_n - old) > EPSILON) {
        old = x_n;
        x_n = 0.5 * (x_n + y / x_n);
    } return x_n;
}
```

sin(x)

```
num = x
den = 1! = 1
term = num / den
sum = term
for (k = 3; abs(term > EPSILON; k += 2
    num = num - x - x
    den = den - (k - 1) - k
    term = num / den
    sum += term
}
return sum
```

mathlib-test.c

```
#include "mathlib.h"
#include <math.h>

int main(void) {
    for(double i = 0.0; i < 3.0; i printf(const char*, ...) → int
        printf("Sqrt(%7.4lf) = %lf, sqrt(%7.4lf) = %lf\n", Sqrt(i),
                                                          Sqrt(i))
    }
    return 0;
}
```

```
static inline Abs (x):
    return x < 0 ? -x : x;
```

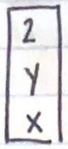
Makefile

```
CC = clang
CFLAGS = -Wall -Werror -Wextra -Wpedantic

all: mathlib-test

mathlib-test: mathlib-test.o mathlib.o

mathlib-test.o: mathlib-test.c
```

Lecture 1/26/2021:

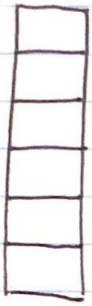
Stack

push x
push y
push 2
pop()
pop()



Stack

Items



Items array
top int
cap int

rule: the top always points to the next available spot
if the top = 0, then it is empty
rule for push: top ++
Items [top] = x } switch order *

Are recursive functions/algorithms inherently inefficient? answer is no
a function call requires creating a stack frame, takes time and space
Use recursion when it makes sense (natural for you to express your algorithm recursively)
Don't use recursion when it does not make the code cleaner
Never more efficient than using iterations
Example of natural: binary search (requires array be sorted, cut in half log base 2 amount of times)
use recursion for searching by dividing the space/search
recursion is natural, good for search, not inherently inefficient

```
pop(): top = top - 1
       x = Items[top]
stack *s = 0 (empty)
push(s)
   p = malloc()
   p → next = s
   s = p
```

free(t) ← don't forget
memory leak

Lab Section 1/26/2021

Game of Life: zero player game, its evolution is determined by its initial state, requiring no further input

finite 2-D grid of cells that represents a universe

- 2 possible states: either dead or alive
- game progresses in generations (steps in time)
- 3 rules to determine the state of the universe
  1) Any live cell with 2 or 3 live neighbors survives
  2) Any dead cell with 3 live neighbors becomes a live cell
  3) All other cells die, either due to loneliness or overcrowding

Universe data type (abstract)

must contain int rows, int cols, bool **grid, bool *** toroidal

Universe *uv_create (int rows, int cols, bool toroidal)

- creates a Universe
- if toroidal is true, then the universe is toroidal
- return of type Universe *
- use calloc() from <stdlib.h>

void uv_delete (Universe *u)

- destructor function
- frees any memory allocated for Universe
- free so there are no memory leaks
- free inside first
- use valgrind for memory leaks

instead of -lm, use -lncurses

```
struct Universe {
    rows
    cols
    **grid
    toroidal
};
```

```
Universe *uv_create (rows, cols, toroidal) {
    Universe *u = (Universe *) calloc(1, sizeof(Universe));
    u->rows = rows
    u->cols = cols
    u->toroidal = toroidal
    u->grid = (bool **) calloc (rows, sizeof(bool*))
```

allocating memory ← cast, pointer to Universe

calloc(count, # of elements, size)

sizeof(Universe) ↖ size

Lecture 1/28/2021:

Computational Complexity
multiply the complexity of loops nested inside
single loop = $O(n)$
two loops = $O(n^2)$
three loops = $O(n^3)$
add the complexity of loops @ the same time
You can think of c as the overhead of an algorithm
Optimizing your code will make c smaller
- may make the run-time of program faster
- only a constant speedup

Changing to another and better algorithm is much more impactful
- the existence of an efficient algorithm determines whether can solve a particular problem
- the algorithm is what matters

Bubble Sort better for less than 50 terms
Complexity can measure time, space, or both
- time is how many steps it takes

A pointer is a variable that holds a memory address
- location of an object in memory
- not all pointers contain an address (set to NULL pointer = 0)
- Null is a macro for
  - ((void *) 0)
  - 0
  - 0L (definition depends on the compiler)

multiple pointers can point to the same address
& operator assigns a pointer to a variable, access the address of a variable
the object a pointer points to can be accessed through dereference
* to instantiate a pointer variable and using it to dereference a pointer

Lecture 2/04/2021:

- Other people's code is always worse than yours, make your intent clear for others/future self
- Simple Things to Avoid:
  - crappy comments
  - dangling else
  - magic numbers
  - global variables
- Crappy comments:
  - modules/functions w/o head comment
- Good comments:
  - explain complexity
  - match the code
  - can be turned into documentation
    - Doxygen is your friend
  - are sometimes edited by English majors/Technical writers
- Dangling else:

```
- if (i < MAX) {
        some complicated;
        code that;
        is multiple statements;
  } else
        return (error);
```

- Keep the braces (unless you're coding in Python)
- Magic numbers:
  - code should rarely have bare numbers
  - all constant values must have names (all else is madness)
    ex: name_buf[128]; // hold the name of the file
  - 128 is a magic numbers
  - What happens when we change the buffer to hold 256?
    - how many lines will have to change to match?
  - fundamental #5: 0, 1, 2

- Proper Constants
  - #define MAX_FILE_NAME 128
  - const int MAX_FILE_NAME = 128;
- Dingleberries
- Version Control Systems
  - don't need hackey sacks (no longer 1975)
- global variables
  - are a sign of failed abstractions
  - makes debugging large programs very difficult
  - must be protected in multi-threaded programs
    - a source of bugs
    - protecting proper access is expensive
      - locks
      - cache effects

Lab Notes: Asgn 4

```
typedef enum ham_rc {
        HAM_ERR = -1,
        HAM_OK = 0,
        HAM_ERR_OK = 1
} ham_rc;
```

index 7 6 5 4 3 2 1 0

byte = MSB 0000 0000 LSB

0000 1000

if you wanted to set the 3rd bit,

byte = MSB 0000 1000 LSB

mask = MSB 0000 0001 LSB

mask = mask << 3 (to get 1 to index 3)

mask = MSB 0000 1000

result = byte | mask

MSB 0000 0000 LSB

OR MSB 0000 1000 LSB

result = MSB 0000 1000 LSB

Another example:

byte = MSB 1011 1001 LSB

mask = MSB 0100 0000 LSB

mask = 1 << 6

result = byte | mask

result = MSB 1111 1001 LSB

} set bit

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

NOT	
0	1
1	0

CLEARING A BIT

byte = MSB 1111 1001 LSB goal: MSB 1011 1001 LSB

mask = MSB 1011 1111 LSB AND

mask = 1 << 6

= MSB 0100 0000 LSB

mask = ~ (1 << 6)

= ~ 0100 0000

= 1011 1111 LSB

result = byte & mask

Getting a bit

byte = MSB 1011 1001 LSB
mask = MSB 0001 0000 LSB (goal)
mask = 1 << 4
result = byte & mask
result = MSB 0001 0000 LSB = 16
result = result >> 4
result = MSB 0000 0001 LSB

```
                                  , index
uint8_t setbit (uint8 byte):
        index = index % 8 (restrict index [0-7])
        mask = 1<< index
        result = byte | mask
    return result
uint8_t  clr bit (uint8 byte , index)
          index = index % 8
          mask = ~(1 << index)
          result = byte & mask
          return result
uint8  get bit (uint 8 byte, index)
          index = index % 8
          mask = 1<< index
          result = byte & mask
          result = result  >> index
          return result
```

BitMatrix

cols ↓↓↓↓

rows
{MSB 0000 0000 LSB}
{MSB 0000 0000 LSB}
{ 0000 0000 }
{ 0000 0000 }

```
struct BitMat {
        u32 rows
        u32 cols
        u8 **mat;
}
```

```
u32 bytes (u32 bits)          doesn't work if
    if (bits % 8 == 0) {      allocating 0 bits
        return bits /8
    else
        return bits /8 +1
```

```
bm_create (rows, cols )
    BitMat *m = calloc (1, sizeof(BitMat*))
    m → rows = rows
    m → cols = cols
    m → mat = (BitMat**) calloc (rows, sizeof (u8 *))
    for (u32 r=0 ; r < rows; r+=1)
        m → mat [r] = (u8 *) calloc(
                                    sizeof(
bytes(cols
(u8 *) calloc (bytes (cols), size of (u8));
```

```
bm_create(4,8)
        rows = 4
        cols = 8
bm_set_bit(BitMat *m, row, col)
 byte = m->mat[row][col/8]
        temp = col % 8
        mask = 1 << temp
        m->mat[row][col/8] = byte | mask
        return;
```

[bm_set_bit()
bm_get_bit()] ← do it in this order
bm_print()

```
// hamming.c
static BitMat *generator
static BitMat *parity  // H_transposed
ham_rc ham_init(void){
        generator = bm_create(4,8)
        bm_set_bit(generator, 0,0)
        bm_set_bit(generator, 1,1)
                :

        bm_print(generator) // for debug
            for(r=0; r<rows; r+=1)
                for(c=0, c<cols; c+=1)
                printf(bm_get_bit(m,r,c))
    for(i=0, i<n)
        for(j=0, j<p)
            for(k=0; k<m)
                C[i][j] += A[i][k] * B[k][j]
```

A is $n \times m$
B is $m \times p$
$C = A \cdot B$ where C is $n \times p$
$0 \le i \le n$
$0 \le k < m$
$0 \le j < p$

(0 0 1 1) (1000 0111
0100 1011
0010 1101
0001 1110)

C = [0000 0000
C[0][0] = 0×1 + 0×0 + 1×0 + 1×(

G_0 =

	0 1 2 3	4 5 6 7
0	1000	0111
1	0100	1011
2	0010	1101
3	0001	1110

reversed
1000 0111
0100 1011
0010 1101
0001 1110

```
G = bm_create()  //could fail
while ! EOF
byte = fgetc()
lo nibble  = lower_ nibble (byte)
hi nibble = higher_nibble (byte)
lo code = ham_encode(lo nibble)  //8 bits → u8
hi code = ham_ encode(hi nibble)
fputc (lo code)
fputc (hi code)
ham_destroy()
ham_ encode (u8 data, u8 *code); formal parameters

ham_ encode (lonibble, &lo code) actual parameters
```

Decoder

```
        //parse getopt()
while != EOF
        locode  = fgetc()
        hicode = fgetc()
        lonibble = ham_decode(locode)      { count bytes
        hinibble = ham_decode(hicode)      { count errors
        byte = pack_ byte(lonibble, hinibble)   count corrections
        fput c (byte)
    ham_destroy()

                                          "rb"
ham_destroy()                             "wb"
        bm_delete (&G)
        bm_ delet (&H)
```

Comparative Sorting Algorithms

DESIGN: how are you going to do comparisons?

$O(n^2)$ sorting algorithms
- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort (worst case)

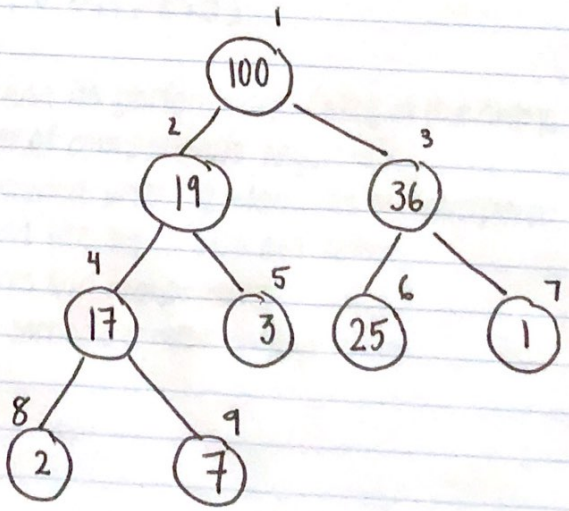
- Shell Sort is an $O(n^{\frac{5}{3}})$ sorting algorithm

```
def gap(n):
    while n>1:
        n=1 if n<=2 else 5*n // 11
        yield (n)
def shellSort(s):
```

Heaps: many different types
- minimum/maximum heaps
- leftist heaps
- binomial heaps

Max Heap
- a single node is a heap

(is a heap if)
- heap is the parent, trees rooted @ both children are heaps
- parent's value (key) is > than that of a child
- no order among children



Lab Section: Assignment 5

Bubble Sort: probably change the (<) to a (>) if you wanted the smallest elements to float to the top instead (stable sort)

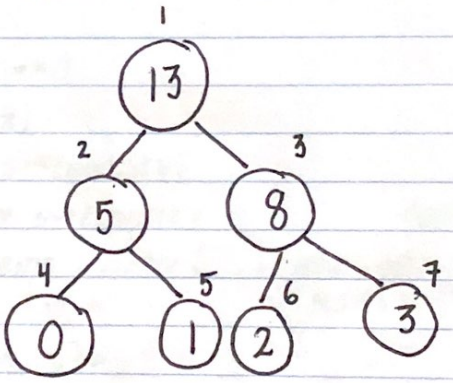
Shell Sort starts with distant elements and moves out-of-place elements into position faster than a simple nearest neighbor exchange (like BubbleSort)

- takes elements that are gap sizes apart and swaps them instead, thus making it generally faster than BubbleSort (depends largely on the gap sequence)

QuickSort: on average, the most efficient sorting algorithm

- we are going to be pushing the array bounds into the stack

HeapSort: for any index k, the index of its left child is 2k and the right child is 2k+1.



[13, 5, 8, 0, 1, 2, 3]

sorting.c acts as a test harness

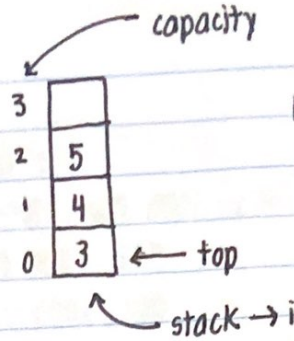
gather statistics about each sort and its performance (size of the array, number of moves required, number of comparisons required)

A comparison is only required/counted when 2 elements are compared

bitmask bottom 30 bits are 1s and the top 2 bits are zeros

general file that keeps track of sorts and comparisons

- BubbleSort does extremely bad in sorting a reverse ordered array



push(3)	pop() = 5
push(4)	pop() = 4
push(5)	pop() = 3

```
struct stack {
        u32 top
        u32 capacity
        int64 *items
    }
stack *stack_create(void)
        --- memory allocation ...
          s→top = 0
          s→capacity = MIN_CAPACITY   //16  Dynamically growing stack
        ---
   }
bool stack_empty(s):
        return s→top == 0
bool stack_push(s, x)
        if s→top == s→capacity{
                double s→capacity
                s→items = realloc(...) // return false if realloc fails
        }
        s→items[s→top] = x
        s→top += 1
        return true
bool stack_pop(s, *x)
        s→top -= 1                    ← check if stack empty
        *x = s→items[s→top]
```

realloc(s→items, ____)
pointer to original memory — size in bytes to reallocate

s→capacity * sizeof(int64_t)
int *arr = (int *) calloc(16, sizeof(in

```
typedef uint8_t set;
        u8 → 8 items
        u32 → 32 items
        u64 → 64 items
empty set = MSB 0000 0000 LSB
add(bubble)                    ⇓
set w/ bubble = MSB 0000 0001 LSB
add(shell)
set w/ bubble & shell = MSB 0000 0011 LSB
```

```
enum sorts {
        bubble = 0
        shell = 1
        quick = 2
        heap = 3
}
```

add (insert)

set s = 0 MSB 0000 0000 LSB

mask = 1 << bubble // MSB 0000 0001 LSB

s | mask ⇒ 0 MSB 0000 0000 LSB

OR MSB 0000 0001 LSB

MSB 0000 0001 LSB

```
set sorts = set_empty ()
while (getopt () ...)
    switch (opt)
        case 'b':
            sorts = set_insert (bubble)
        case 'h':
            sorts = set_insert (heap)

for (sorts s = bubble;
#include <stdio.h>  #include <stdint.h>  #include <stdbool.h>
typedef enum Dogs {
        Clifford = 0,
        Snoopy,
        Pluto,
        Odie
} Dogs;
typedef uint8_t Set;
Set set_insert (Set s, Dog d) {
        return s | (1 << d % 8);
}
Set set_empty () {
        return 0; // MSB 0000 0000 LSB
}
int main (void) {
        Set dogs = set_empty ();
        dogs = set_insert (dogs, Clifford);
        dogs = set_insert (dogs, Snoopy);
        for (Dog d = Clifford ; d <= Odie; d += 1) {

bool
set_member (Set s, Dog d) {
        return s & (1 << d % 8)
}
```

```
if (set_member(dogs, d)) {
        switch(d) {
            case Clifford :
                puts("clifford is in the set"); break
            case Snoopy : ---
            case Pluto ---
            default ---
```

For our assignment, use the switch to run whichever program is specified
When "a" is specified, take the complement of the empty set

```
def shell_sort(array)
    for gap in [4,2,1]
        for i in range(gap, len(arr)):
            for j in range(i, gap-1, -gap):
                if A[j] < A[j-gap]
                    swap(A[j], A[j-gap])
```

A 0 1 2 3 4
array = [2, 5, 4, 3, 1] len(arr) = 5

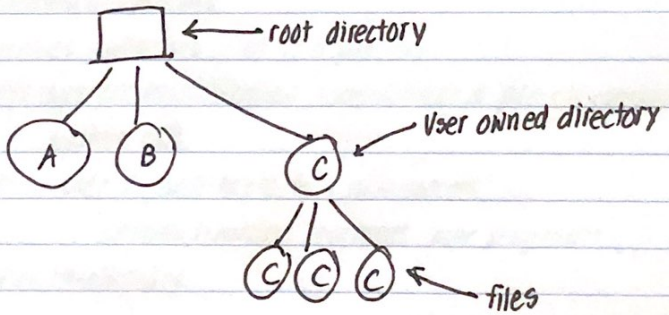
gap = 4
for i = 4; i < 5
    for j = 4; j > gap-1; j -= gap A[j]
        if A[j] < A[j-gap]
            swap

array = [1, 5, 4, 3, 2]
gap = 2
    for i in range(2,5)
        for j in range(2, gap-1, -gap)

  0 1 2 3 4
array = [1,5,4,3,2]
array = [1,3,4,5,2]
array = [1,3,2,5,4] gap=1,
array = [1,3,2,5,4]
array = [1,2,3,5,4]
array = [1,2,3,4,5]

Lecture 2/16/2021

- want large amounts of data to store, def. of large has changed by 6 orders of magnitude over the years
- Memory (DRAM) does not survive turning the computer off
- files are accessed using names, memory is accessed using addresses
- computer has millions of files
  - data center has billions of files
  - internet has trillions of files
- file naming, files have base names and extensions
- in some operating systems, the extension is a separate entity
- in Unix, the extension is only used by convention (doesn't care what the file extension is)
- file Operations (create, delete, open, close, read, write)
- naming is still better than using numbers, still limited
- use hierarchy to manage complexity
- file systems allow this to be done with directories, may call them folders
- a single directory system: contains 4 files, owned by 3 different people



- directory operations (create, delete, opendir, closedir, readdir, rename, link, unlink), read, insert, and delete
- processes
  - code, data, and stack
  - program state
    - CPU registers, program counter, stack pointer
  - only one process can be running in a single CPU core @ any given time
- Address space
  - programs execute code, each instruction has an address
  - each byte of data has its own address
  - address space is the region of a computer's memory where the program executes

- the loader could relocate the instructions by address a base address to each one
- could be registers that point to the first byte of the program's memory (base register) that is added
- memory is divided into levels
  - Cache: small amount of fast expensive memory (L1 on CPU chip, L2 on or off the chip)
  - Main memory: medium speed, medium price DRAM (L3 offchip, SRAM)
  - Disk: GB of slow, cheap, storage
- memory management might include OS
- memory needs 2 things for multiprogramming: relocation, protection
- OS cannot be certain where a program will be executed
- allocation structures must be updated when memory is freed
- buddy allocation: allocate memory in powers of two
- virtual memory: allows OS to handout more memory than exists on the system
- keeps recently used stuff in physical memory
- keeps all this hidden from processes
- VM useful for multiprogrammed systems
- program uses virtual addresses, addresses local to a process
- processes created in 2 ways: system initialization, execution of a process creation system call
- systems calls can come from user request to create a new process
  - already running processes: user programs, system daemons
- processes end: voluntary or involuntary
- voluntary: normal exit