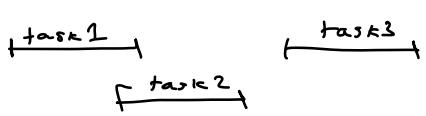
Computational Complexity.

Branch of cs, study of algorithms.

Computers and Algorithms

Interval Scheduling

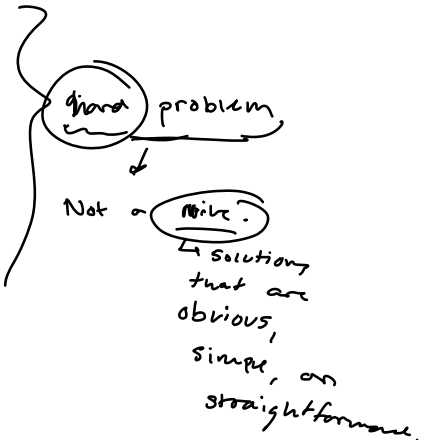


how do you schedule them with finite resources?

Worker 1: task 1 task 2 task 3

worker 2:

worker 3:



Graph Equality

Given a graph $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ how check if they are equal?

$V_1 = V_2$ and $E_2 = E_1$? If so they match.

E_1

	n_1	n_2						
n_1	x	1	x	x	x	x	x	x
n_2	1	1						

This has a naive solution, but how fast is it?

Algorithms Efficiency:

(1) times runs over code

(2) 151 learned linear time, quadratic

Algorithmic (provisional)

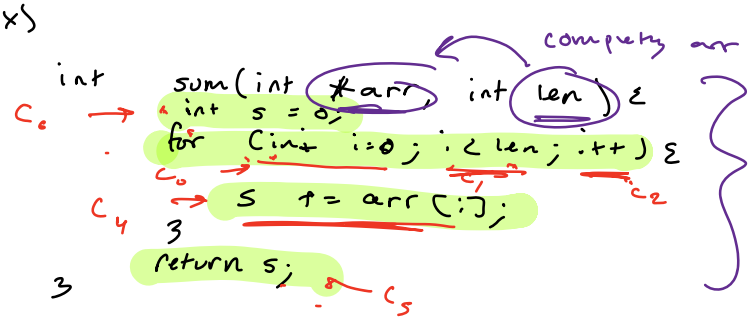
efficient runs quickly on real instances, → analytically?

Worst-case → this an example "contrived" that would be worst for algorithm,

best-case → fastest example on algorithm.

how algorithm change based on parameters because then we can study the performance analytically.

ex)



C_0 → time to declare and init an int;

C_1 → time to compare i < len

C_2 → i++

C_4 → update and add and retrive for s.

C_5 → return time

$$T_{sum}(n) = C_0 + C_0 + n(C_2 + C_1 + C_4) + C_5$$

n ← value

Asymptotic Time

Given $T(n)$ we want to ask given another function $f(n)$ whether or not, $\Theta(f(n))$, the order of $f(n)$, is a upper bound of $T(n)$.

Asymptotic Upper Bound

More precisely that $T(n)$ is $\Theta(f(n))$ if there exists a constant $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$

ex $T(n) = 13$

$f(n) = n^2$

$T(n) \leq c \cdot f(n)$

$13 \leq c \cdot n_0^2$

$13 \leq 100 \cdot (100)^2$

$T(n) \leq \Theta(n^2)$.

← this is not tight because, so I claim, $\exists\ \Theta(f(n))$ s.t. $\Theta(f(n)) < \Theta(n^2)$.

I claim $T(n) \leq \Theta(1)$.

$13 \leq c \cdot 1$

$13 \leq 15 \cdot 1$

Asymptotic Lower Bound

We say $T(n)$ is $\Omega(f(n))$ if

there exists $c > 0$ and $\forall n_0 \geq 0$ s.t., $T(n) \geq c \cdot f(n_0)$.

$f(n) \leq T(n) \leq c \cdot f(n_0)$

ex) $T(n) = 3n + 42$

$f(n) = n$

$3n_0 + 42 \geq c \cdot n_0$

$3 \cdot n_0 + 42 \geq 2n_0$ true for all n_0.

$\Omega(f(n))$ is a lower bound for $T(n)$

$T(n)$ having a running is by finding a $O(f(n))$ and $\Omega(f(n))$ so $\Omega(f(n)) \leq T(n) \leq O(f(n))$ then we say $T(n)$ is order $f(n)$.

ex) return time

$$T_{sum}(n) = c_0 + c_0 + n(c_2 + c_1 + c_4) + c_5$$

↑ value

Upper

$f(n) = n$;

Set $c = \max(c_0 + c_2 + c_1 + c_4 + c_5)$

$T(n_0) \leq c \cdot n_0$

$2c_0 + n_0(c_2 + c_1 + c_4) + c_5 \leq n_0 c$

$T(n) \leq O(n)$

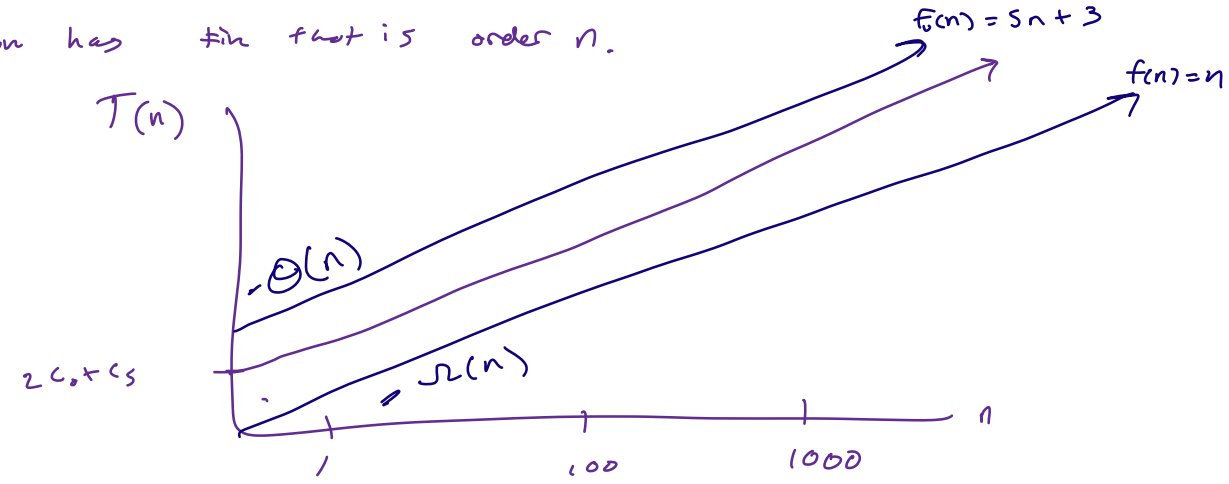
Lower bound:

$\varepsilon = \min(c_0, c_1, c_2, c_4, c_5)$.

$2c_0 + n_0(c_2 + c_1 + c_4) + c_5 \geq \varepsilon n_0$

$T(n) \geq \Omega(n)$

Sum has time that is order n.



$\Omega(n) \leq T(n) \leq \Theta(n)$

$T(n) = \Theta(n)$

For example Graph equality

Proof

$\Omega(f(n))$

Ω ← lower
O ← upper
Θ ← tight bound

Suppose I have a function called f.

I tell you that f has $\Theta(n^2)$ run time.

g has runtime $\Theta(n^2)$.

What is the runtime of

$$\Theta(f + g) = \max(\Theta(n^2), \Theta(n^2))?$$

↳ Why well if g is say n^3 time, then f+g will be n^3 since f is only n^2 and $\Theta(n^2) \leq \Theta(n^3)$.

ex)

```
llist * findsValue(llist * list, void *elem) {
    llist * pos = list;                          c0
    while (pos != NULL) {                        c1
        if (pos->item == elem)                   c2
            {return pos;}                        c3
        pos = pos->next;                         c4
    }
    return NULL;                                 c5
}
```

$T(n) = c_0 + n(c_1 + c_4) + c_3 + cc_5$;

$\Theta(T(n)) = n$ ← Linear time

$\Omega(n) \leq T(n) \leq O(n)$

ex) Quadratic Time

```
insert_in_list() {
    // found spot        ← n to look all positions
    // push element      ← n
    // put
}
for(i=0; i < len; i++) {
    insert_in_list
}
```

$T_2(n) = 2n$

n

$T_1(n) = n\,T_2(n)$;

$= 2n^2$

$T_1(n) = 2n^2$

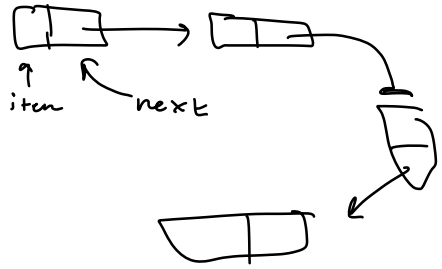
$\Omega(n^2) \leq 2n^2 \leq O(n^2)$

$\Theta(n^2)$.

Sorting better → $n \log n$ time.

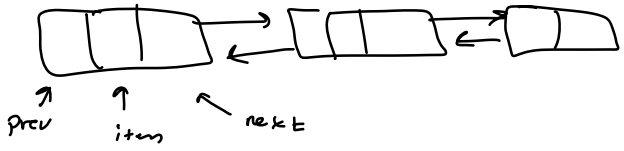
Data Structures

– Singly Linked List



– Doubly linked list

There's two pointers to prev and to next.



For example

struct dllist;

```
void remove(dllist** plist, void *elem) {
  dlist *list = *plist;
  dlist * tmp;
  dlist *pos = list;
  while( pos!=NULL) {
      if( pos == list && pos->item == elem) { // first node
            if (pos->next != NULL) {
                  pos->next->prev = NULL;
            }
            *plist = pos->next; // moves the value of the pointer in the caller
            pos = pos->next;
            free(list);
            list = plist.
      }    return;
      if(pos->item == elem) {                    // also if it's the last.
            pos->prev->next = pos->next;
            if (pos->next != NULL) {
                  pos->next->prev = pos->prev;
            }
            tmp = pos;
            pos = pos->next;
            free(tmp);
      }
}
```

Generic Types

What? void * → Void → no type

void * is a pointer to an undisclosed type or value.

YOU CANNOT dereference a void pointer.

But you can cast it.

ex)

```
int a = 6;
int *pa = &a;
printf(..., *pa); // 6.

void * v = (void *) &a;
printf( *v ) //6   NO   *v → what type is that??
printf( *((int *)v) ) //→ 6  so yes!
```

```
typedef struct linked_list llist_t;
struct linked_list {
    void * item;
    llist_t * next;
};
```

```
typedef struct {
    int r, g, b;
} pixel;

pixel * makePixel(int r, int g, int b) {
    pixel * p = malloc(sizeof(pixel));
    p->r = r;
    p->g = g;
    p->b = b;
    return p;
}
```

```
int main() {
    llist_t *pixelList = makeList();
    pixel *p = makePixel(...);

    llist_append(pixelList, (void*) p);
```

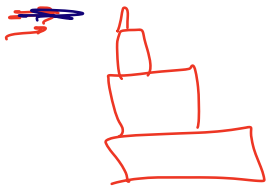
```
void *item = getHead(pixelList);
pixel recovered = *(pixel*)item;
llist_t intList = someList;
int a = 6;
llistAppend(intList, &a);
```

```
void llist_append(llist_t *list, void *item) {
  // check list != NULL;
  while (list->next != NULL) {
      list = list->next;
  }
  llist *new = makeNewListNode(item);
  list->next = new;
}

// head for front
// tail for last
void* getHead (llist *list) {
  if (list != NULL) {
      return list->item;
  }
  return NULL
}
```

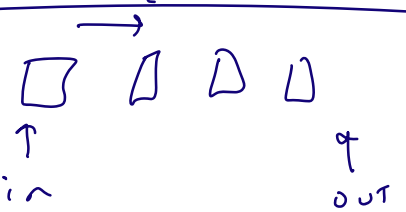
Stack → LIFO
Last In First Out

Stacks → insert "on top"
pop from the top.



push → adds to the top of stack
pop → removes and returns the top of stack.

Queues → FIFO
First In First Out



push → in of list
pop → from the out

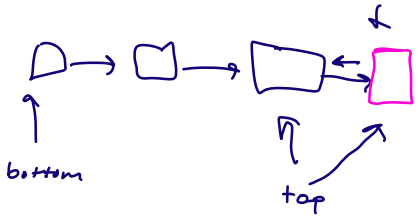
dllist → doubly node
sllist → singly node
llist → linked node

```
typedef struct {
    sllist *bottom;
    sllist *top;
} stack;

stack * stack_create() {
    return ((stack*) malloc(sizeof(stack)));
}

void push(stack *stack, void *item) {
    sllist_append(stack->top, item);
    stack->top = stack->top->next;
}
```

$\Theta(1)$

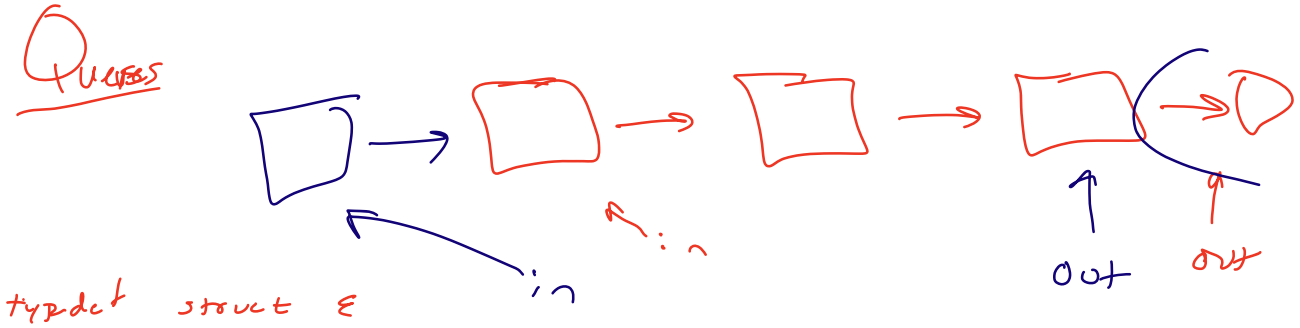



```
void *pop(stack *stack) {
    void *remo = stack->top;
    sllist *pos = stack->bottom;
    while (pos->next != NULL) {
        pos = pos->next;
    }
    stack->top = pos;
    return (remo);
}
```

$\Theta(n)$

```
sllist *pos;
for(pos = stack->bottom;
    pos->next != NULL;
    pos = pos->next) {;}
```

Queues




```
typedef struct {
  sllist *in;
  sllist *out;
} Queue;
```

$\Theta(1)$

```
void push(queue &q, void *item) {
  sllist *new = sslist_create(item);
  new->next = q->in;
  q->in = new;
}
```

$\Theta(n)$

```
void *pop(queue *q) {
  void *rem = q->out;
  sllist *pos = q->in;
  while (pos->next != NULL) {
    pos = pos->next;
  }
  q->out = pos;
  return rem;
}
```