

---

# ALGORITHMS & DATA STRUCTURES

---

**CSCI 382**

**Erica Blum**

2023

Notes by Aliya Ghassaei

# Contents

<b>1</b>	<b>Asymptotic notation</b>	<b>5</b>
1.1	Definitions . . . . .	5
1.1.1	Big- $O$ . . . . .	5
1.1.2	Big- $\Omega$ . . . . .	5
1.1.3	Big- $\Theta$ . . . . .	5
1.1.4	Little- $o$ . . . . .	5
1.1.5	Little- $\omega$ . . . . .	5
1.2	Proofs strategies . . . . .	5
1.2.1	Proving from definition . . . . .	5
1.2.2	Proving using limit properties . . . . .	6
1.2.3	Helpful properties: . . . . .	6
1.2.4	Recurrences (substitution, recurrence trees, master theorem) . . . . .	6
1.3	Master method . . . . .	6
1.4	Recurrence trees . . . . .	7
1.4.1	Models of computation (comparison model, random access model) . . . . .	7
1.5	Solving problems by reducing to searching (choose + apply a data structure) . . . . .	7
<b>2</b>	<b>Interfaces &amp; implementations</b>	<b>7</b>
2.0.1	Priority queue interface . . . . .	7
2.1	Solving problems by reducing to sorting (choose + apply a sorting algorithm) . . . . .	7
2.2	Solving problems by designing a new algorithm . . . . .	8
<b>3</b>	<b>Introduction</b>	<b>9</b>
3.1	Vocabulary . . . . .	9
<b>4</b>	<b>Data Structures</b>	<b>9</b>
4.1	Sequences . . . . .	9
4.1.1	Static Operations . . . . .	9
4.1.2	Dynamic Operations . . . . .	10
4.2	Sets . . . . .	10
4.2.1	Static Operations . . . . .	10
4.2.2	Dynamic Operations . . . . .	10
4.2.3	Running times . . . . .	10
4.3	Direct access table . . . . .	10
4.3.1	Example . . . . .	10
4.4	Hash tables . . . . .	11
4.4.1	Running time . . . . .	11
4.4.2	Example . . . . .	11
4.4.3	Better example: universal family of hash functions . . . . .	11
4.5	Binary Trees . . . . .	11
<b>5</b>	<b>Running Time</b>	<b>12</b>
5.1	red black . . . . .	12

<b>6</b>	<b>Correctness proof</b>	<b>12</b>
6.1	Loop invariant method . . . . .	12
<b>7</b>	<b>Recursion</b>	<b>12</b>
<b>8</b>	<b>Limits</b>	<b>12</b>
8.1	Limit Laws . . . . .	12
<b>9</b>	<b>Sorting</b>	<b>12</b>
9.1	Permutation Sort . . . . .	12
9.2	Selection Sort . . . . .	13
9.3	Merge Sort . . . . .	13
<b>10</b>	<b>Binary Search Trees</b>	<b>13</b>
10.1	non-modifying operations . . . . .	14
10.2	modifying properties . . . . .	14
<b>11</b>	<b>priority queues and heaps</b>	<b>15</b>
11.1	Priority Queue Interface . . . . .	15
11.2	priority queue sort . . . . .	15
11.3	binary heaps . . . . .	15
11.3.1	infer tree from array . . . . .	15
<b>12</b>	<b>Cheat Sheets</b>	<b>16</b>
12.1	Running Times for DS . . . . .	16
12.2	orders . . . . .	16
<b>13</b>	<b>servers</b>	<b>16</b>
<b>14</b>	<b>Graph algorithms</b>	<b>17</b>
14.1	Definition . . . . .	17
14.2	Representing graphs . . . . .	17
14.3	Running time . . . . .	17
14.4	Breadth-first search . . . . .	17
14.5	Path problems . . . . .	17
<b>15</b>	<b>Rules</b>	<b>17</b>
15.1	Sum rules . . . . .	17
15.2	Log rules . . . . .	17
<b>16</b>	<b>Midterm review</b>	<b>18</b>
<b>17</b>	<b>Lecture 16</b>	<b>18</b>
17.1	Topological Sort . . . . .	18
17.2	Weighted Graphs . . . . .	18
17.3	weighted paths terminology . . . . .	18

<b>18 Lecture 17</b>	<b>19</b>
18.1 DAG Relaxation . . . . .	19
18.2 Bellman-Ford . . . . .	19
<b>19 Logical Equivalence</b>	<b>19</b>

# 1 Asymptotic notation

## 1.1 Definitions

### 1.1.1 Big- $O$

**Intuition:** If  $f(n) \in O(g(n))$  then  $f(n)$  grows no faster than  $g(n)$ .  $g$  is an asymptotic upper bound for  $f$ .

**Definition 1.1.**  $O(g(n)) := \{f(n) : \exists c, n_0 > 0 \mid 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0\}$

### 1.1.2 Big- $\Omega$

**Intuition:** If  $f(n) \in \Omega(g(n))$ ,  $f(n)$  grows at least as fast as  $g(n)$ .  $f$  is a lower bound on  $g$ .

**Definition 1.2.**  $\Omega(g(n)) := \{f(n) : \exists c, n_0 > 0 \mid 0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0\}$

### 1.1.3 Big- $\Theta$

**Intuition:**  $f(n)$  grows at the same rate/within a constant factor of  $g(n)$

**Definition 1.3.**  $\Theta(g(n)) := \{f(n) : \exists c_1, c_2, n_0 > 0 \mid 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0\}$

**Theorem 1.1.** For any two functions  $f(n)$  and  $g(n)$ ,  $f(n) = \Theta(g(n)) \iff f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$

### 1.1.4 Little- $o$

**Definition 1.4.**  $o(g(n)) := \{f(n) : \text{for any } c > 0, \exists n_0 > 0 \mid 0 \leq f(n) < c \cdot g(n) \text{ for all } n \geq n_0\}$

### 1.1.5 Little- $\omega$

**Definition 1.5.**  $\omega(g(n)) := \{f(n) : \text{for any } c > 0, \exists n_0 > 0 \mid 0 \leq c \cdot g(n) < f(n) \text{ for all } n \geq n_0\}$

**Theorem 1.2.**  $f(n) \in \omega(g(n)) \iff g(n) \in o(f(n))$

## 1.2 Proofs strategies

### 1.2.1 Proving from definition

This means that you'd use the definition to prove that a function  $g$  is in some complexity class.

**Steps:**

- 1.

**Example:**

### 1.2.2 Proving using limit properties

Use the following properties:

### 1.2.3 Helpful properties:

**Transitivity:**

$$\begin{aligned} f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) &\implies f(n) = \Theta(h(n)) \\ f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) &\implies f(n) = O(h(n)) \\ f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) &\implies f(n) = \Omega(h(n)) \\ f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) &\implies f(n) = o(h(n)) \\ f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) &\implies f(n) = \omega(h(n)) \end{aligned}$$

**Reflexivity:**

$$\begin{aligned} f(n) &= \Theta(f(n)) \\ f(n) &= O(f(n)) \\ f(n) &= \Omega(f(n)) \end{aligned}$$

**Symetry:**

$$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$$

**Transpose symetry:**

$$\begin{aligned} f(n) = O(g(n)) &\iff g(n) = \Omega(f(n)) \\ f(n) = o(g(n)) &\iff g(n) = \omega(f(n)) \end{aligned}$$

**Limit laws:**

$$\begin{aligned} \lim_{n \rightarrow \infty} f(n)/g(n) \in [0, \infty) &\Rightarrow f = O(g) \\ \lim_{n \rightarrow \infty} f(n)/g(n) \in (0, \infty] &\Rightarrow f = \Omega(g) \\ \lim_{n \rightarrow \infty} f(n)/g(n) \in (0, \infty) &\Rightarrow f = \Theta(g) \\ \lim_{n \rightarrow \infty} f(n)/g(n) = 0 &\Rightarrow f = o(g) \\ \lim_{n \rightarrow \infty} f(n)/g(n) = \infty &\Rightarrow f = \omega(g) \end{aligned}$$

**More:**

$$f_1 = O(g_1) \text{ and } f_2 = O(g_2) \implies f_1 + f_2 = O(\max(g_1, g_2))$$

### 1.2.4 Recurrences (substitution, recurrence trees, master theorem)

## 1.3 Master method

Apply Master theorem to solve recurrences of the form  $T(n) = aT(n/b) + f(n)$  where  $a \geq 1$ ,  $b > 1$ , and that  $f(n)$  is asymptotically nonnegative.

**Master Theorem:** fix this later

**Theorem 1.3. Case 1:** If  $f(n) = O(n^{\log_b a - \epsilon})$  for some constant  $a > 0$ , then  $T(n) = n \log^b a$ .

*Intuition:* When  $f(n)$  is polynomially less than the watershed function, the cost of leaves dominates the cost of the rest of the tree.

**Case 2:** If  $f(n) = O(n \log^b a \log^k n)$  for some constant  $k > 0$ , then  $T(n) = n \log^b a \log^{k+1} n$ .

*Intuition:* There are  $\log n$  levels, each with a cost of  $n \log^b a \log^k n$ .

*Simple (common) case:*  $k = 0 \Rightarrow f(n) = n \log^b a \Rightarrow T(n) = n \log^b a \log n$ .

**Case 3:** If  $f(n) = n \log^b a +$  for some constant  $a > 0$ , and  $f(n)$  satisfies the regularity condition (below), then  $T(n) = f(n)$ .

*Regularity condition:*  $a f(n/b)^c \leq f(n)$  for some constant  $c < 1$  and all large enough  $n$ . (Usually true for polynomial  $f(n)$ .)

*Intuition:* When  $f(n)$  is polynomially greater than the watershed  $f(n)$ , the cost of the root dominates the cost of the leaves.

## 1.4 Recurrence trees

If of the form  $T(n) = aT(n/b) + f(n)$ , the each node has  $a$  branches and the starting  $a$  branches will be  $n/b$ . Then next layer will be  $n/b^2$  because you are plugging  $n/b$  for  $n$  in  $T(n)$ .

The first layer is  $f(n)$  time to combine, next layer will be  $a/$  the denominator in that branch all times  $f(n)$ , basically the watershed function when you plug the new  $n$  in

keep going till you get 1. depth =  $\log n$  base whatever  $n$  was divided by. make it into a sum

1. given recurrence relation, prove that it's in some class
2. method: (recurrence tree) -i guess, -i substitution method
3. look on recurrence sheet
4. ref sheet will have mast theorem, properties or relations for exponents and logarithms, summations

### 1.4.1 Models of computation (comparison model, random access model)

## 1.5 Solving problems by reducing to searching (choose + apply a data structure)

## 2 Interfaces & implementations

**Interfaces:** set, sequence, priority queue

**Implementations:** arrays, sorted arrays, linked lists, hash tables, BSTs, red-black trees, min/max heap

### 2.0.1 Priority queue interface

1. heap

## 2.1 Solving problems by reducing to sorting (choose + apply a sorting algorithm)

1. Mergesort Insertion sort Selection sort Heapsort

## 2.2 Solving problems by designing a new algorithm

1. Brute force “Decrease and conquer” Divide and conquer



## 3 Introduction

### 3.1 Vocabulary

1. interface:
2. implementation
3. array-based (example: array/list)
4. static: region of memory stays static
5. pointer-based (ex: linked list)
6. dynamic

## 4 Data Structures

sequence data structure operations in O

Implementation	<b>build(X)</b>	<b>get_at/set_at</b>	<b>insert_first/delete_first</b>	insert/delete last	insert at/delete at
array	$n$	1	$n$	$n$	$n$
linked list	$n$	$n$	1	$n$	$n$
dynamic array	$n$	1	$n$	$1_{(a)}$	$n$
binary tree	$n \log n$	$h$	$h$	$h$	$h$

Set data structure operations in O

Implementation	<b>build(X)</b>	<b>find(k)</b>	<b>insert(x)/delete(x)</b>	<b>find.min/max()</b>	<b>find.prev/next</b>
array	$n$	$n$	$n$	$n$	$n$
Sorted array	$n \log n$	$n$	$n$	1	$\log n$
direct access array	$u$	1	1	$u$	$u$
Hash table	$n_{(e)}$	$1_{(e)}$	$1_{(a)(e)}$	$n$	$n$
binary tree	$n$	$h$	$h$	$h$	$h$

### 4.1 Sequences

Given a collection of elements called **X**, store elements  $x_0, \dots, x_{n-1}$

#### 4.1.1 Static Operations

1. **build(X)** - builds the sequence from **X**
2. **iter\_seq()** output items in order
3. **len()** - returns the number of elements in **X**
4. **get\_at(i)** - returns item at index  $i$
5. **set\_at(i, x)** - insert  $x$  at location  $i$

### 4.1.2 Dynamic Operations

Dynamic sequences use static sequence operations in addition to these operations:

1. `insert_at(i, x)` - inserts item  $x$  at location  $i$
2. `delete_at(i)` - deletes and returns item at location  $i$

## 4.2 Sets

### 4.2.1 Static Operations

1. `build(X)` - builds set
  - with array implementation, `build(X)`  $\in O(n)$  where  $n = |X|$
2. `len()` - returns  $|X|$
3. `find(k)` - returns item with key  $k$ 
  - with array implementation, `find(k)`  $\in O(n)$  where  $n = |X|$
4. `find_min()` - returns item with smallest key
5. `find_max()` - returns item with largest key
6. `find_next(k)` - return next smallest item from  $k$  (?)
7. `find_prev(k)` - return next largest item from  $k$  (?)
8. with array, min, max, prev, next all  $\in O(n)$

### 4.2.2 Dynamic Operations

1. `insert_at(x, i)` - insert element  $x$  at index  $i$
2. `delete_at(i)` - delete and return element at  $i$
3. with array, insert/delete  $\in O(n)$

### 4.2.3 Running times

1. with sorted array, `find(k)`, `prev`, `next`  $\in O(\log n)$ , insert/delete in  $n$ , find min/max in 1, build is  $n \log n$

## 4.3 Direct access table

### 4.3.1 Example

1. items are Reed students
2. “keyspace” is all possible 8-digit ID numbers
3. index by Reed ID makes `find(k)`  $\in O(1)$ , but wastes space

## 4.4 Hash tables

Before: sorting and searching using only comparisons on items.

Now: use keys in more complex ways arrow direct access table

```
h : {ids} -> {0, ... , m-1}
# assume m = Theta(m)
```

store items as determined by  $h(\text{key})$

cannot be a bijection

collision: things in ids that map to same idem in codomain

solution: instead of storing item in table, store pointer to another data structure

“Chaining” to some other data structure

```
def hash.find(k):
    digest = h(k)
    c = get(h.direst) -> go find k?
```

### 4.4.1 Running time

ideally: all chains are constant size so then all operations on chains are constant size

chose  $m, h$  such that all chains end up with about  $n/m$  approx  $O(1)$  things in them

### 4.4.2 Example

modular division:  $h(k) = k \bmod(m)$ .

Problem: only as even as keys themselves

if too even, might get mapped to same thing

### 4.4.3 Better example: universal family of hash functions

example of one family:

defien  $h_{ab}(k) = (((ak + b) \bmod p) \bmod m)$  where  $p$  is a large prime and  $a, b$  chosen randomly from 0 to  $p-1$

Formally, define family  $\mathcal{H}(p, m) = \{h_{ab}(k) \mid a, b \in [0 \dots p-1] \text{ and } a \neq 0\}$  desired property: for some  $h \in \mathcal{H}$ ,

Pr that any pairs of keys collide  $[h(k_1) = h(k_2)] \leq \frac{1}{m}$  for all  $k_1, k_2$  such that  $k_1 \neq k_2$

parallel concept for sorting comparison model

lower bound Omega ( $n \log n$ )

## 4.5 Binary Trees

traversal order: flatten tree **Operations:**

1. **find.first()**: finds the first node in traversal order. go left until you can't anymore
2. **find.last()**: finds last node in traversal order. goes right until you can't anymore

## 5 Running Time

### 5.1 red black

root black

every leaf black

if node is red, both children are black

for every node, all paths from node to each descendant have same number of black nodes

## 6 Correctness proof

### 6.1 Loop invariant method

**Sketch:**

1. Define a loop invariant  $L$
2. **Initialization:** prove  $L$  holds before 1st iteration
3. **Maintenance:** if  $L$  holds before loop, still true before next one
4. **Termination:** show that loop terminates and that  $L$  still holds

## 7 Recursion

running time for recursive algo "solving a recurrence" - substitution method - recurrence tree - master theorem

## 8 Limits

### 8.1 Limit Laws

## 9 Sorting

Sorting Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

### 9.1 Permutation Sort

```
permutation_sort(A):  
    generate every possible permutation of A  
    check if A is sorted  
    continue until you find a sorted permutation
```

## 9.2 Selection Sort

```
selection_sort(A, i):  
    1. finds biggest item in A[:i]  
    2. swaps biggest thing with thing at A[i]  
    3. recurse on A[:i-1]
```

## 9.3 Merge Sort

```
if n = 0, 1: done  
otherwise:  
    split array into Left, Right  
    MergeSort(Left), MergeSort(Right)  
    Merge(Left, Right)
```

1. specificity: array A, p starting point, r ending point
2. if  $p \geq r$  (if p equal to r, then only one element and return)
3. convention:  $A[a:b] = []$  if b less than A
4. a:b means including both end points
5. define midpoint  $q = (p + r)/2$  (floor of the average)
6. Left =  $A[p:q]$
7. Right  $A[q+1:r]$
8. MergeSort(A, p, q)
9. MergeSort(A, q+1, r)
10. Merge(A, p, q, r)
  1. comparisons in constant time?
    - (a) mostly comparing integers
    - (b) assume  $i \leq j$  in constant time
    - (c) might be more complex
    - (d)

## 10 Binary Search Trees

$O(h)$  time where h is height

usually use these for sets

example: set us cs profs

name/id

adam/560

charlie/703

erica/998

greg/997

jim/100

include bst from notes

properties: for any  $X$ ,  $\text{keys in } x.\text{left} \leq x.\text{key} \leq \text{keys } x.\text{right}$

can also store a sequence in this way

ex 2: grocery list

apples, bananas, cereal, dish soap, eggs

order doesn't matter

bst property: for any  $L[i]$ , all items in  $L[i].\text{left}$  appear before  $L[i]$

all items in right subtree appear after  $L[i]$  is balanced, height is at most  $O(\log n)$

when balanced, called red-black trees

## 10.1 non-modifying operations

1. find min - all the down left
2. find max - all the way down right
3. kind  $k$  - compare at every node
- 4.

## 10.2 modifying properties

1. insert item: traverse, then insert as a leaf
2. delete: find successor and swap 16 with 17, then remove old leaf (find a safe leaf to remove)
- 3.

balanced if even under dynamic operations it maintains height  $O \log N$   
red-black tree

1. dummy nodes
2. imagine single dummy node "tree ends"
3. stop drawing them

loop invariants

different cases for red black trees

## 11 priority queues and heaps

### 11.1 Priority Queue Interface

new: delete max - remove item with highest priority  
find max = find highest priority item  
supports subset of set interface operations  
optimized for finding min or max, we will focus on max priority  
also has insert and build

### 11.2 priority queue sort

algo in slides  
simple because you make the ds do it

### 11.3 binary heaps

#### 11.3.1 infer tree from array

require: completely full in upper levels (until last level)  
binary trees with  $n$  things -> arrays of length  $n$   
bottom level should be left justified  
different than traversal ordering function  
root is index 0  
finding the left child of  $i$  is just at index  $2i+1$   
 $\text{right}(i) = 2i+2$   
 $\text{parent}(i) = \text{floor}((i-1)/2)$   
max heap properties:

1. at node  $l$ , the value of thing at position  $l$  is going to be at least the value its children
- 2.

tree does not live in memory, just array  
we want higher importance nodes to be closer to the top  
bottom layer is less than all elements above it  
example 1 from slides: not a max heap: 15 is a node that is not greater than or equal to both of its children

inserting( $x$ )  
-append  $x$  to array: whole function is in  $O(\log n)$   
-fix problems in heap  
fixing: swap up with parents until max heap rule is followed:  
check that val of parent  $\geq$  val node, if no, swap and keep going  
delete-max():  
this deletes the root!  
1. swap root with item at node number  $n-1$ , then delete thing at  $n-1$   
the fix downwards:

- start at root
  - swap it with the greater key in its children, then continue
- this is  $O(\log n)$  time

## 12 Cheat Sheets

### 12.1 Running Times for DS

Operations $O(\cdot)$				
Set Data Structure	Container	Static	Dynamic	Order
array	AX	ALA	248	
sorted array	AL	ALB	008	

### 12.2 orders

## 13 servers

1. servers and clients
2. each server: some capacity
3. each client: some workload
4. assign so that no server goes over its capacity

implementation

1. initialize(X) where X is set of servers
2. connect(client, workload)
3. disconnect(client)
4. clients-of(server): return list of clients connected to that server

how to do:

idea

use priority queue for clients and prioritize biggest

priority queue for servers so that you can find best fit the best

notes on actual:

server has set of clients where key is client id and value is workload

priority queue storing the servers

store based on priority, which is available capacity

structures:

pq P: (server.id, available capacity)

set S: (maps server id to set of clients and pointer to si in P) set C: maps client id to connected server and workload (if not stored as client object)



## 14 Graph algorithms

### 14.1 Definition

1.  $G = (V, E)$  where for all  $(u, v) \in E$ ,  $u \neq v$  and no duplicate edges (for this class)
2.  $Adj^+(u) := \{v \in V \mid (u, v) \in E\}$  (outgoing neighbors of node  $u$ )
3. incoming neighbors of  $u$   $Adj^-(u) = \{v \in V \mid (v, u) \in E\}$
4. out degree( $u$ ) = size of  $adj^+$
5. in degree( $u$ ) = size  $adj^-$

### 14.2 Representing graphs

matrix or adj list

storing:

top level set adj of vertices

lower level: adj lists  $Adj(u)$  either set or sequence (default to outgoing

can be direct access array

)

### 14.3 Running time

1. linear means  $O(V + E)$  (size of those sets)  
doing something with each edge and each vertex
2.  $|E| = O(V^2)$  for simple graphs  
most edges you can have is fully connected graph
3. undirected:  $|E| \leq \frac{V(V-1)}{2}$  (proved on homework)
4. directed :  $|E| \leq V(V-1)$  (V size chose 2)

### 14.4 Breadth-first search

### 14.5 Path problems

## 15 Rules

### 15.1 Sum rules

### 15.2 Log rules

Product rule	$\log_b(MN) = \log_b(M) + \log_b(N)$
Quotient rule	$\log_b\left(\frac{M}{N}\right) = \log_b(M) - \log_b(N)$
Power rule	$\log_b(M^p) = p \cdot \log_b(M)$
Base switch rule	$\log_b(c) = \frac{1}{\log_c(b)}$
Base change rule	$\log_b(x) = \frac{\log_c(x)}{\log_c(b)}$

## 16 Midterm review

## 17 Lecture 16

### 17.1 Topological Sort

"finishing order" - order in which is finishes visiting (recursive calls pop all the way up)  
DAG meaning?

**Definition 17.1.** Topological order of  $G = (V, E)$  : ordering of vertices such that every  $(u, v)$  in  $E$  satisfies that  $u$  is before  $v$  in the order (in context of directed graphs)

Find topological ordering of graph: Edges = (A,C) (C, E) (A, B) (B, G) (B, E) (C, D) (E, D) (E, F)

check: ACBEDFG

other: ABGCEDF

order for F and G don't matter

Story problem: teams

Solution sketch:

1. Given info, what is the graph? vertices = set of employees, edges are links
2. goal: a topological ordering satisfies the constraint
3. find topological ordering or prove none exists
4. prove none exists: get sequence, check if topological
5. (reverse of visiting order is a topological sort)
6. solution exists iff  $G$  is a DAG
7. alternatively show a cycle so there is no ordering
8. or show graph is acyclic -> topological ordering exists

can solve by cycle detection problem

### 17.2 Weighted Graphs

unweighted: distance is num edges in shortest paths

add a weight function from set of edges to integers

### 17.3 weighted paths terminology

why infimum? has to do with range of weights we're trying to consider

problem with negative weights and cycles

confused...

get nice graph

## 18 Lecture 17

assume G is (undirected) graph satisfies def 1 (show that G must also satisfy def 2)

weighted graphs given shortest paths weights, can compute shortest paths tree in linear time ( $O(V + E)$ )

### 18.1 DAG Relaxation

high level:

for each vertex, maintain estimate which is upper bound as true shortest distance distance

repeatedly improve

refining estimates: true shortest-path weight between any pair of vertices u, v is upper bounded by weight of shortest path from u to v that has a specific last edge (x, v)

proposing path through x as possible path

see slide for formal definition

update estimate to use edge if edge exists to make it shorter

### 18.2 Bellman-Ford

## 19 Logical Equivalence

Contrapositive	$P \implies Q$	$\neg Q \implies \neg P$
----------------	----------------	--------------------------