

# CS 577 - Greedy

Marc Renault

Department of Computer Sciences  
University of Wisconsin – Madison

Fall 2023

TopHat Section 001 Join Code: 477366  
TopHat Section 002 Join Code: 560750



# GREEDY ALGORITHMS

## What is a Greedy Algorithm (GREEDY)?

- Typically, thought of as a *heuristic* that is locally optimal.
- Is GREEDY always the best? No, but a good place to start.
- This notion has yet to be fully formalized, and it often problem specific. *→ you treat every input as its the last input.*

① consider input,  
give some order to the input    ② Have a simple rule to  
process input in that order

## Definition from Priority Algorithms

A greedy algorithm is an algorithm that processes the input in a specified order. For each request in the input, the greedy algorithm processes it so as to minimize (resp. maximize) the objective, assuming that the request is the last request.]

For a given problem, there may be many greedy algorithms.

# Is GREEDY OPTIMAL?



## Not always: Bin Packing Problem

- Bins of size 1, and requests of size  $(0, 1]$ .
- Objective: Pack the items in the minimum number of bins.
- Greedy heuristic: FIRST FIT INCREASING (FFI)

Non-optimal example:

- $\sigma = \langle 1/2 - \varepsilon, 1/2 - \varepsilon, 1/2 + \varepsilon, 1/2 + \varepsilon \rangle$
  - FFI: 3 bins
  - OPT: 2 bins
- ↗ starts with smallest, goes to largest  
 keeps track of all bins.  
 } Better way.  
 ↗ optimal.  
 2 bins.

Techniques for showing that GREEDY is optimal:

- Always stays ahead
- Exchange argument

↗ show one input that returns wrong.  
 $\exists x \}$  there exists a wrong input.  
 one counterexample to show its  
 not optimal.

# most compatible GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

- sort input by finish time.
- take the first one
- keep taking the next most compatible with it and keep going on.

## Heuristic 4: Finish First

Schedule a compatible request with the smallest finish time.

## Optimal?

Counter-example? Let's try and prove it.

→ take input:  
sort by finish time  
add on the other ( $r_i$ ) requests.

# EXERCISE: FORMALIZE THE ALGORITHM (PSEUDOCODE)

HEURISTIC 4: FINISH FIRST

## Algorithm: FINISHFIRST

Let  $S$  be an initially empty set.

**while**  $\sigma$  is not empty **do**

Choose  $r_i \in \sigma$  with the smallest finish time (break ties arbitrarily).  $\rightarrow$  first grab.

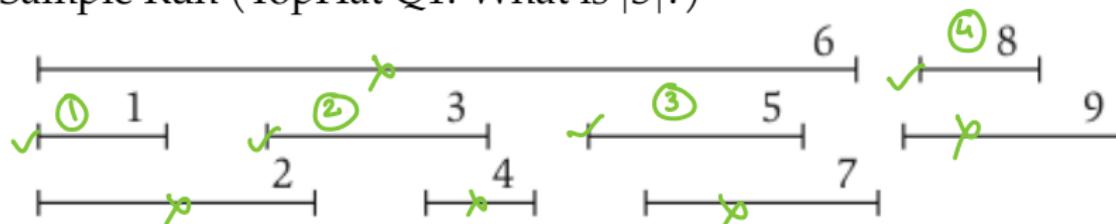
Add  $r_i$  to  $S$ .

Remove all incompatible requests in  $\underline{\sigma}$ .

**end**

**return**  $S$

Sample Run (TopHat Q1: What is  $|S|$ ?)



# ANALYSIS OF FINISHFIRST

## Observation 1

*Immediate from the definition of FINISHFIRST,  $S$  is compatible.*

Showing Optimality *prove no. of requests in both schedules are same.*

Let  $S^*$  be an optimal solution.

- We can show the strong claim that  $\underline{S} = \underline{S^*}$ . *Best case scenario.*
  - Can there be multiple  $S^*$ ? Yes.
  - Hence, we can show the weaker claim of  $|S| = |S^*|$  for this problem.
  - Technique: “Always stays ahead”
    - At every time step  $i$ ,  $|S_i| \geq |S_i^*|$ . } *Break up the processing to show each of them are optimal.*
- } *show optimality.*

## STAYS AHEAD ANALYSIS

$$K = \text{no. of jobs}$$

$$m = \text{optimal no. of jobs}$$

$$m \leq K$$

- all** • Label  $S = \langle i_1, \dots, i_k \rangle$  such that  $f_{i_u} < f_{i_v}$  for  $u < v$ . → use FF on this and match it with the solution.
- optimal** • Label  $S^* = \langle j_1, \dots, j_m \rangle$  such that  $f_{j_u} < f_{j_v}$  for  $u < v$ .

Ans: FF > solution  
for stays ahead.

### Lemma 1

For all  $i_r, j_r$  with  $r \leq k$ , we have  $f_{i_r} \leq f_{j_r}$  → optimal schedule

### Proof.

The proof is by induction. → we can go from our schedule to the optimal schedule. proof by induction:

- For  $r = 1$ , the claim is true as FINISHFIRST first selects the request with the earliest finish time. *Base case: r=1, only one request*
- Assume true for  $r - 1$ . *Inductive hypothesis: Assume true for (r-1), we have*
  - By the induction hypothesis, we have that  $f_{i_{r-1}} \leq f_{j_{r-1}}$ .  $f_{i_{r-1}} \leq f_{j_{r-1}}$
  - The only way for  $S$  to fall behind  $S^*$  would be for FINISHFIRST to choose a request  $q$  with  $f_q > f_{j_r}$ , but this is a contradiction.



# STAYS AHEAD ANALYSIS

→ our schedule produced by FF.

- Label  $S = \langle i_1, \dots, i_k \rangle$  such that  $f_{i_u} < f_{i_v}$  for  $u < v$ .
- Label  $S^* = \langle j_1, \dots, j_m \rangle$  such that  $f_{j_u} < f_{j_v}$  for  $u < v$ .  
↳ optimal schedule

## Lemma 1

→ some constant K

For all  $i_r, j_r$  with  $r \leq k$ , we have  $f_{i_r} \leq f_{j_r}$

finish time of i < finish time of j.

The optimality of FINISHFIRST, essentially, follows immediately from Lemma 1.

→ this statement basically proves the optimality of finish first.

# FINISHFIRST IS OPTIMAL

- Label  $S = \langle i_1, \dots, i_k \rangle$  such that  $f_{i_u} < f_{i_v}$  for  $u < v$ .  
 $\nearrow k \text{ elements}$
- Label  $S^* = \langle j_1, \dots, j_m \rangle$  such that  $f_{j_u} < f_{j_v}$  for  $u < v$ .  
 $\searrow m \text{ elements.}$

## Theorem 2

*FINISHFIRST produces an optimal schedule.*

$S = \text{more/equally optimal.}$

$m > k$  : no. of requests in  $S^*$  are greater than no. of requests in  $S$ .

## Proof.

By way of contradiction, assume that  $|S^*| > |S|$ . This implies that  $m > k$ . Lemma 1 shows that FINISHFIRST is ahead for all the  $k$  requests. That means it would be able to add the  $(k + 1)$ -st item of  $S^*$ . As it did not, this contradicts the definition of FINISHFIRST. □

$\nearrow$  assumed schedule is more optimal than FF.

need to implement this code in homework.

## IMPLEMENTATION AND RUNNING TIME

### Algorithm: FINISHFIRST

Let  $S$  be an initially empty set.

**while**  $\sigma$  is not empty **do**

    Choose  $r_i \in \sigma$  with the smallest finish time (break ties arbitrarily).

    Add  $r_i$  to  $S$ .

    Remove all incompatible request in  $\sigma$ .

**end**

**return**  $S$

} you need to sort or before FF.

$O(n \log n)$  priority queue  
or merge sort / heap sort.

$O(1)$   
 $O(n)$   
 $O(n \log n)$

### Implementation Details

- Choose request with smallest finish time:  
Before processing, sort requests:  $\underline{\underline{O(n \log n)}}$ .
- Remove incompatible requests: Advance in sorted order until a request with a compatible start time.

Overall:

$$\underbrace{O(n \log n)}_{\text{sorting}} + \underbrace{O(n)}_{\text{process requests.}} = \overbrace{O(n \log n)}^{\text{overall runtime.}}$$

# INTERVAL EXTENSIONS

①

- Online variant: Requests are presented in a specific order to the algorithm. At request  $i$ , the algorithm does not know  $n$  nor  $r_{i+1}, \dots, r_n$ .

②

- Add a value to the intervals (online/offline). Now objective is to maximize the total value of scheduled intervals.

③

- Scheduling all intervals: Interval Colouring Problem.
  - Unlimited resources and the algorithm must produce multiple compatible schedules that cover all the requests (without duplicates between the schedules).
  - Objective: Minimize the number of schedules.

stays ahead  
exchange argument.

another technique to show  
greedy algorithms are optimal.

## EXCHANGE ARGUMENT: MINIMIZE MAX LATENESS

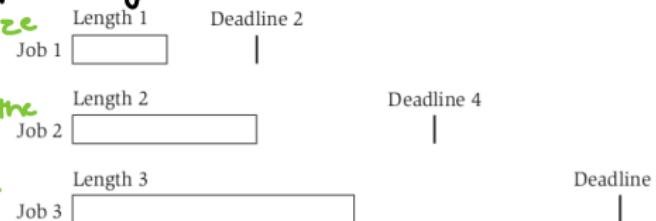
interval scheduling had a defined start and end time

## SCHEDULING PROBLEM: MINIMIZE MAX LATENESS

this one has a processing time and deadline.

we need to minimize  
max lateness

so, the job that is the  
most late has to be  
as small as possible.



-jobs can be started  
at any time, and  
they take  $i$  time  
to complete.

-one job running at  
a time

→ we need to schedule all the jobs

### Problem Definition

- $n$  jobs and a single machine that can process one job at a time
- For job  $i$ :
  - $t_i$  is the processing time,  $d_i$  is the deadline. → no lateness.
  - Lateness  $l_i = f_i - d_i$  if  $f_i > d_i$ ; 0 otherwise.
- Objective: Build a schedule for all the jobs that minimizes the max lateness.

TopHat Discussion 2: What greedy heuristic might work?

# GREEDY ALGORITHMS FOR MINIMIZING MAX LATENESS

method 1.

Heuristic 1: Increasing processing time.

Schedule jobs by increasing  $t_i$ .

choose the job with smallest processing time  $t_{\min}$ .

Optimal?

Counter-example: Jobs  $(t_i, d_i)$ :  $\{(1, 100), (10, 10)\}$

$$\Delta_i = t_i - d_i = 1 - 100 = -99 \quad \Delta_i = t_i - d_i = 10 - 10 = 0$$

start with job 1. - so you schedule it.



$$\text{total lateness} = 0 + 1 = 1$$

we can do better than this if we schedule them the opposite way.  
job 2, job 1.

# GREEDY ALGORITHMS FOR MINIMIZING MAX LATENESS

slack = amount of time you have between processing time and deadline.

start with smallest slack.

(jobs very close to their deadline).

→ processing time  
you have between deadline and  
finish time.

Heuristic 2: Increasing slack.

Schedule by increasing  $d_i - t_i$ . lateness =  $f_i - d_i$

Optimal?

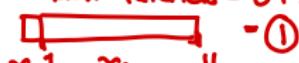
Counter-example:

	①	②	max lateness
lateness:	$f_i - d_i$ $= 11 - 2$ $= 9$	$f_i - d_i$ $= 10 - 10$ $= 0$	$= 9$

Jobs  $(t_i, d_i)$ :  $\{(1, 2), (10, 10)\}$

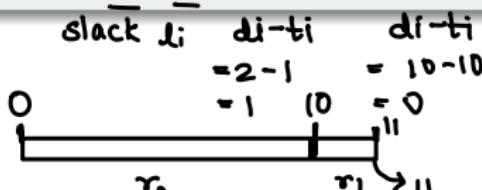
optimal would be!

$$\text{max lateness} = 0 + 1$$



$$d_1 = f_i - d_i = 1 - 2 = -1 \quad \textcircled{1}$$

$$d_2 = 11 - 10 = 1 \quad \textcircled{2}$$



# GREEDY ALGORITHMS FOR MINIMIZING MAX LATENESS

- schedule jobs based on increasing deadlines. (Optimal).

EDF

Heuristic 3: Earliest deadline first.

Schedule by increasing  $d_i$ .

Optimal?

Counter-example? Let's try and prove it.

# EXERCISE: FORMALIZE THE ALGORITHM (PSEUDOCODE)

HEURISTIC 3: EARLIEST DEADLINE FIRST.

## Algorithm: EDF

Let  $J$  be the set of jobs.

Let  $S$  be an initially empty list.

**while**  $J$  is not empty **do**

Choose  $j \in J$  with the smallest  $d_i$  (break ties arbitrarily).

Append  $j$  to  $S$ .

**end**

**return**  $S$

choose the tie breakers  
in jobs arbitrarily.

$\hookrightarrow$  earliest deadline.  
 $\hookrightarrow$  add  $j$  to  $S$ .  
(schedule)

Sample Run (TopHat Q1: What is max lateness?)

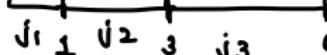
Job 1  Length 1      Deadline 2 |

$$= \text{lateness} = 0.$$

Job 2  Length 2      Deadline 4 |

lateness:  $0 + 0 + 0 = 0$

Job 3  Length 3      Deadline 6 |



# ANALYSIS OF EDF

## Observation 2

*There is an optimal schedule with no idle time.*



Let  $S^*$  be an optimal solution.

*taking later jobs and shift them earlier makes the lateness better.*

- Is it sufficient to show that  $|S| = |S^*|$ ? No. as we schedule all jobs, optimal solutions, cardinality has to be same
- Can there be multiple  $\underline{S}^*$ ? Yes.
- We need to show either  $\underline{S} = \underline{S}^*$ , or  $\underline{S} \equiv \underline{S}^*$  for max lateness.
- Technique: “Exchange Argument”
  - Start with an optimal solution  $S^*$  and transform it over a series of steps to something equivalent to  $S$  while maintaining optimality.
  - $S^* \equiv S_1 \equiv S_2 \equiv \dots \equiv S$  for max lateness.

## EXCHANGE ARGUMENT ANALYSIS

you need to define inversion.

### Definition 3

↳ 2 jobs out of order w.r.t deadline are considered inversion.

A schedule  $A$  has an inversion if there are jobs  $i$  and  $j$  with  $i$  scheduled before  $j$  and  $d_j < d_i$ .

~~$i$  scheduled before  $j$  but deadline of  $j$  is before  $i$ .~~

### Lemma 4

All schedules with no inversions and no idle time have the same lateness.

→ could be diff in order, but max lateness will be same.

This means we get a schedule equivalent to what earliest deadline first will provide.

- Only vary in jobs with the same deadline.
- Jobs with same deadline must sequential.
- Ordering of jobs with same deadline won't change lateness.



$$f_j^* = 10$$

## ANALYSIS OF EDF

*inversions are disorders in the list according to deadline.*

$$d_j < d_i$$

$$3 \leq 4$$

### Theorem 5

*There is an optimal schedule that has no inversions and no idle time.*

$$d_i - l_i = (f_i - d_i) - (f_j - d_j)$$



### Proof.

- If  $S^*$  has an inversion, then there is a pair of jobs  $i$  and  $j$  such that  $j$  is scheduled immediately after  $i$  and has  $d_j < d_i$ .
- We will swap  $i$  and  $j$  to create a new schedule  $S'$ . Note that  $S'$  has one less inversion than  $S^*$ .
- We need to show that  $S'$  has the same max lateness as  $S^*$ .

- Swapping  $i$  and  $j$  means that  $l'_j$  (lateness in  $S'$ ) is less than that in  $S^*$ .

$$= l_i = f_i - d_i \quad l_j = f_j - d_j$$

- Lateness of  $i$  may increase, but:

$$l'_i = f'_i - d_i = f^*_i - d_i \leq f^*_j - d_j = l^*_j.$$

$$10 - 4 \leq 10 - 3$$

- Let  $S^* := S'$  and repeat until no more inversions.

*proof by induction:  $S^* \rightarrow S'$  : reducing one inversion without increasing max lateness.  $\square$*

# EDF IS OPTIMAL

## Corollary 6

EDF produces an optimal schedule.

↳ no inversions and no idle time.

### Proof.

- EDF produces a schedule with no inversions and no idle time.
- From Theorem 5, there <sup>exists</sup> is an optimal schedule with no inversions and no idle time.
- Lemma 4 shows that these two schedules have the same max lateness.

↳ all optimal schedules have the same max lateness.

↳ sorting takes  $O(n \log n)$

Run time: Sort the jobs by deadline:  $O(n \log n)$ .



graph exploration  $\rightarrow$  creates a shortest path tree from the graph  $G$ .  
**DIJKSTRA'S**  $\rightarrow$  takes into account all the weights

### Algorithm: Dijkstra's

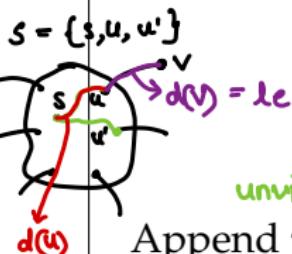
Let  $\underline{S}$  be the set of explored nodes.

For each  $u \in \underline{S}$ , we store a distance value  $\underline{d}(u)$ .

Initialize  $\underline{S} = \{s\}$  and  $\underline{d}(s) = 0$

**while**  $S \neq V$  do  $\rightarrow$  while all nodes are not visited, choose a node not visited.

Choose  $v \notin S$  with at least one incoming edge originating from a node in  $S$  with the smallest



$$\overline{d}'(v) = \min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$$

minimizer  
take the minimum path length from the set of path lengths obtained from all outgoing edges from visited nodes

Append  $v$  to  $S$  and define  $d(v) = d'(v)$ .

**end** (Dijkstra's takes the node that has smallest distance to  $S$  seen so far.)

How is it greedy?

TopHat 3: Which technique to prove optimality?

stays ahead.

# CORRECTNESS OF DIJKSTRA'S

Theorem 7

$\rightarrow S = \text{set of nodes explored/visited}$

Consider the  $S$  at any point in the execution of Dijkstra's. For each  $u \in S$ , the path  $P_u$  is a shortest  $s - u$  path.

$\hookrightarrow$  inductive hypothesis:

Proof.

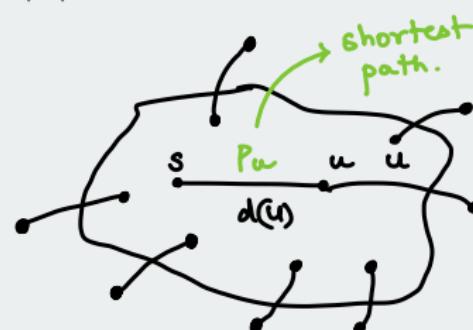
By induction on the size of  $S$ .

Base case : • For  $|S| = 1$ , the claim follows trivially as  $S = \{s\}$ .

• By the induction hypothesis, for  $|S| = k$ ,  $P_u$  is the shortest  $s - u$  path for all  $u \in S$ .

whatever we choose to add next,  
will be the shortest path

distance from  $s \rightarrow s$   
 $= 0$ .



# CORRECTNESS OF DIJKSTRA'S

show green path  
is impossible

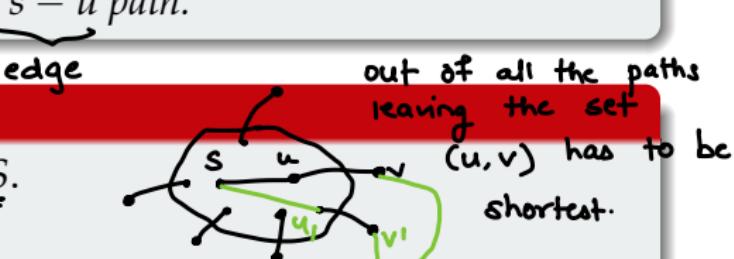
## Theorem 7

Consider the  $S$  at any point in the execution of Dijkstra's. For each  $u \in S$ , the path  $P_u$  is a shortest  $s - u$  path.

### Proof.

By induction on the size of  $\underline{S}$ .

- In step  $k + 1$ , we add  $v$ .
  - By definition,  $P_v$  is shortest path connected to  $S$  by one edge.
  - Since  $P_u$  is a shortest path to  $u$ ,  $P_v$  is the shortest path to  $v$  when considering only the nodes of  $S$ .
  - Moreover, there cannot be a shorter path to  $v$  passing through another node  $y \notin S$  else  $y$  that would be added at  $k + 1$ .



$v'$

$$d(u') + \ell_{u',v'} \geq d(u) + \ell_{u,v}$$



# DIJKSTRA'S OBSERVATIONS

## Algorithm: Dijkstra's

Let  $S$  be the set of explored nodes.

For each  $u \in S$ , we store a distance value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

Choose  $v \notin S$  with at least one incoming edge originating from a node in  $S$  with the smallest  $d'(v) =$

$$\min_{e=(u,v): u \in S} \{d(u) + \ell_e\}$$

Append  $v$  to  $S$  and define  $d(v) = d'(v)$ .

**end**

- Negative edge weights,

where does it fail?

if weight is -w, wrong path might

- TopHat 4: It is get added.

graph exploration,

what kind of exploration?

as you are exploring all neighbors of s

- Weighted (continuous) BFS

# IMPLEMENTATION AND RUN TIME OF DIJKSTRA'S

$$\begin{aligned} n &= \text{no. of nodes} \\ m &= \text{no. of edges} \end{aligned}$$

**Algorithm:** Dijkstra's

Let  $S$  be the set of explored nodes.

For each  $u \in S$ , we store a distance value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

    Choose  $v \notin S$  with at least one incoming edge originating from a node in  $S$  with the smallest  $d'(v) =$

$$\min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$$

    Append  $v$  to  $S$  and define  $d(v) = d'(v)$ .

**end**

    priority queue  
to get  $O(m \log n)$

worst case scenario  
to check every edge.

$$\begin{aligned} (n-1) \times m \\ = nm - m \\ = \cancel{(nm)} \end{aligned}$$

you need this.

$\rightarrow O(m \log n)?$

- TopHat 5:  
Number of iterations of the loop?
- Key Operations:

- Finding the min:  
Easy in  $O(m)$

- Overall:  
 $O(mn)$

- How can we get

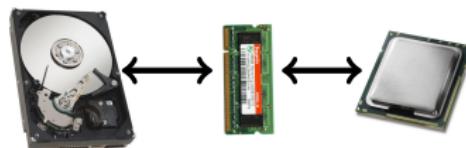
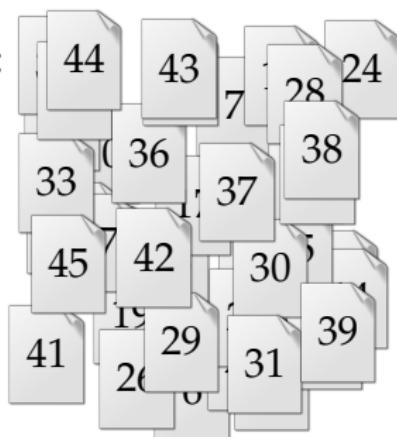
# PAGING

→ coding question (more greedy).

- run
- cannot time out
- autograder (10 mins).
- don't do bunch of linear searches.

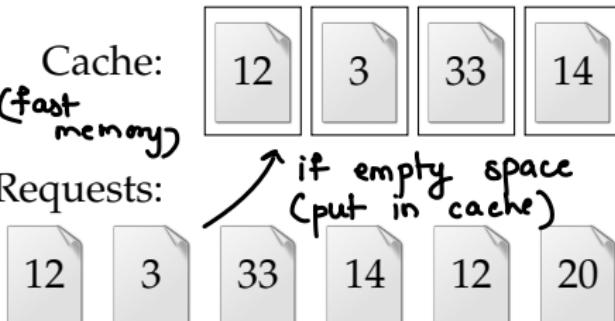
## PAGING PROBLEM

(slow  
memory)  
universe  
of  
memory



Cache:  
(fast memory)

### Requests:



→ if empty space  
(put in cache)

↳ this page is  
already in cache,  
so ignore.

- $\mathcal{U}$ : universe of pages ( $|\mathcal{U}| > k$ ).
  - Cache of size  $k$ .
  - Requests are to the pages of  $\mathcal{U}$ .
  - Goal: Minimize the number of page faults (requests to pages not in the cache).

eviction strategies:  
(you always serve the requests)

eviction strategies:  
(you always serve the requests)

# OFFLINE GREEDY ALGORITHM

## Farthest-in-Future (FF)

Evict the page whose next request is the furthest into the future.

Small Run:

- $\mathcal{U} = \{a, b, c\}$
  - $k = 2$
  - $\sigma = \langle p, f, f, f, f, f, f \rangle$
- request sequence*

cache: a b

page faults:  $1 \neq 4 = \text{total}$

- TopHat 6: How many faults in small run?  
4 faults.

TopHat 7: Which strategy to prove optimality?

exchange argument

- look at a decision that doesn't follow heuristic
- change once to match heuristic.

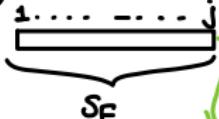
# PROVING FF OPTIMALITY

EXCHANGE ARGUMENT

$$\text{show: page faults at } S' \leq \text{page faults at } S$$

(schedule)

S:



$S$  should be identical to the SF (schedule with farthest in the future heuristic)

## Theorem 8

exchange is happening at  $(j+1)$  request. rest of the sequence, it could be doing anything.

Let  $S$  be a schedule for the  $n$  requests that make the same eviction decisions as  $S_{FF}$  for the first  $j$  items. Then, there is a schedule  $S'$  that makes the same eviction requests as  $S_{FF}$  for the first  $j + 1$  items with no more faults than  $S$ .

Kind of like

performing

Proof. proof by induction.

farthest in the future

(define later)

you go from  $s$  to  $s'$



- If on request  $j + 1$ ,  $S$  behaves as  $S_{FF}$ . Then define  $S'$  as  $S$  and the claim follows. kicked out diff pages to bring in the same page
- Otherwise, say  $S$  evicts  $u$  and  $S_{FF}$  evicts  $v$ . We will build  $S'$  by following  $S_{FF}$  for the first  $j + 1$  requests. Note that the number of faults are the same for  $S$  and  $S'$  up to  $j + 1$ , and the caches match except for  $u$  and  $v$ . page faults should be no more. you need to align the cache of  $S$  and  $S'$  at the end.

# PROVING FF OPTIMALITY

## EXCHANGE ARGUMENT

### Theorem 8

Let  $S$  be a schedule for the  $n$  requests that make the same eviction decisions as  $S_{FF}$  for the first  $j$  items. Then, there is a schedule  $S'$  that makes the same eviction requests as  $S_{FF}$  for the first  $j + 1$  items with no more faults than  $S$ .

### Proof.

- From  $j + 2$  onward,  $\underline{S'}$  follows  $S$  until either:
  - $S$  evicts  $v$ . In this case,  $S'$  evicts  $u$ . *→ page faults become equal cache equals.*
  - $S$  evicts  $g \neq v$  to bring  $u$  into the cache. In this case,  $S'$  evicts  $g$  and brings in  $v$ .
- Note: Since  $S_{FF}$  evicts  $v$  at  $j + 1$ ,  $u$  must be requested before  $v$ .
- In either case, both  $S$  and  $S'$  have a page fault, and afterwards their cache match.



# MST

(minimum spanning trees)

# MINIMUM SPANNING TREE PROBLEM

graph exploration

MST Problem *depends heavily on connectivity, is a tree as it won't have any cycles.*

Let  $G = (V, E)$  be a connected graph, where  $|V| = n$  and  $|E| = m$ . For each edge  $e$ ,  $c_e > 0$  is the cost of the edge.

- Find an edge set  $F \subseteq E$  with minimum cost that keeps the graph connected. That is,  $F$  should minimize  $\sum_{e \in F} c_e$ .

## Observation 3

*Let  $T = (V, F)$  be a minimum-cost solution to the problem described above. Then,  $T$  is a tree.*

## Proof.

- By the definition of the problem,  $T$  must be connected.
- By way of contradiction, assume that  $T$  has a cycle  $C$ . Remove any edge from  $C$  resulting in a graph  $T'$ .  $T'$  is still connect and has a cost less than  $T$ .



## ALGORITHM DESIGN

*exchange argument*

TopHat Discussion 3: What greedy heuristic might work?

### Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

### Prim's (1957) Algorithm

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

### Reverse-Delete (Kruskal's 1956) Algorithm

*removing  
start from most expensive edge, so as to not disconnect the graph.*

- Sort edges by cost from highest to lowest.
- Remove edges unless graph would become disconnected.

# ASSUME DISTINCT WEIGHTS

WLOG (WITHOUT LOSS OF GENERALITY)

≡ what you show in a proof holds true for every case.

sometimes its obvious, but sometimes you have to prove WLOG.

## Theorem 9

(HW Q2) If all edge weights in a connected graph are distinct, then  $G$  has a unique MST.

## Observation 4

All we need is a consistent tie-breaker when  $c_{e_1} = c_{e_2}$  for some pair of edges. I.e. based on the labels of the vertices of  $e_1 \cup e_2$ .

Assumption: all edge weights are distinct.

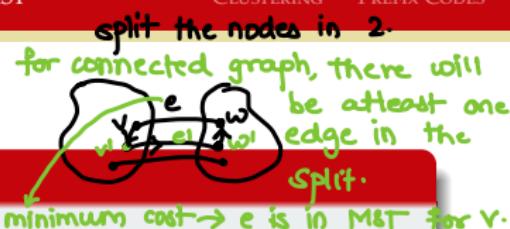
# ANALYZING MST HEURISTICS

**Lemma 10 "cut-property"**

$S$  is a proper subset of  $V$ .

Let  $S \subset V$  be a non-empty proper subset of the nodes, and let

$e = (v, w)$  be the minimum cost edge connecting  $S$  and  $V \setminus S$ . Then, every MST contains  $e$ .



## Proof.

By exchange argument:

- Let  $T$  be a spanning tree that does not contain  $e$ .
- Let  $e' = (v', w')$ , where  $e'$  is in  $P_{v,w} \in T$ ,  $v' \in S$ , and  $w' \in V \setminus S$ .
- Let  $T' = T \setminus e' \cup e$ . where we removed  $e'$  and added  $e$ .
- $T'$  is connected as  $e$  is a  $P_{v,w} \in T'$ .
- Since  $c_e < c_{e'}$ , cost of  $T'$  is less than  $T$ .



# KRUSKAL'S ALGORITHM IS OPTIMAL

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

→ for any cut in the graph, Kruskal chooses the lowest cost edge first.

Theorem 11

Kruskal's Algorithm produces an MST.

optimal as you keep choosing edge  $e$  which is minimum in comparison to other unchosen edges.

Proof.

to prove optimality

- Let  $e = (v, w)$  be the edge added at any step  $i$ .
- Since  $e$  does not create a cycle,  $v \in S$  and  $w \notin S$  (WLOG).
- As  $c_e$  is the minimum cost edge, the claim follows from Lemma 10.



# PRIM'S ALGORITHM IS OPTIMAL

## Prim's (1957) Algorithm

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

→ literally on implementation of cut property. cuts out all the high cost neighbors edges and chooses the cheapest edge.

Theorem 12  
Prim's Algorithm produces an MST.

## Proof.

- Immediate from Lemma 10.
- That is, Prim's algorithm does exactly what Lemma 10 describes.



# REVERSE-DELETE IS OPTIMAL

## Reverse-Delete (Kruskal's 1956) Algorithm

- Sort edges by cost from highest to lowest.
- Remove edges unless graph would become disconnected.

How should we prove that it produces an MST?

↙ can't use cut property to show optimality.

→ if you have a cycle, you remove the highest cost one.

# REVERSE-DELETE IS OPTIMAL

## Lemma 13

Let  $\underline{C}$  be any cycle in  $\underline{G}$ , and let  $e$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $G$ .



two node sets leftover.

### Proof. exchange argument

- Let  $T$  be a spanning tree that does contain  $e$ . *the most expensive edge.*
- Let  $S$  and  $V \setminus S$  be the nodes of the connected components after removing  $e$  from  $T$ .
- Let  $e'$  be an edge in  $C$  that connects  $S$  and  $V \setminus S$ . *As it is a cycle, you have one more edge going from  $S \rightarrow (V-S)$*
- Let  $T' = T \setminus e \cup e'$ . *without  $e$  but with  $e'$ .*
- $T'$  is connected as  $e'$  reconnects  $S$  and  $V \setminus S$ .
- Since  $c_e > c_{e'}$ , cost of  $T'$  is less than  $T$ .



# REVERSE-DELETE IS OPTIMAL

## Lemma 13

Let  $C$  be any cycle in  $G$ , and let  $\underline{e}$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $\underline{G}$ .

## Theorem 14

Reverse-Delete Algorithm produces an MST.

→ similar to the proof for Kruskal's

### Proof.

- Let  $e = (v, w)$  be an edge removed at any step  $i$ .
- By definition  $\underline{e}$ , belongs to a cycle  $C$ .
- As  $\underline{c}_e$  is the maximum cost edge of  $C$ , the claim follows from Lemma 13.



# IMPLEMENTING PRIM'S ALGORITHM

*→ Dijkstra's Algorithm, but different minimizer*  
**Prim's (1957) Algorithm**

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

## Key Operations

- Retrieve the minimum valued edge between  $S$  and  $V \setminus S$ .
- Prim's and Dijkstra's have nearly identical implementations (but different minimizers)!

*→ cut property*

## Priority Queue (min-heap)

- ExtractMin ( $O(1)$ ):  $n - 1$  times.
- ChangeKey ( $O(\log(n))$ ):  $m$  times.

*→ most efficient runtime: priority queue.*

$$\left. \begin{array}{l} \text{ExtractMin } O(1) \\ \text{ChangeKey } O(\log(n)) \end{array} \right\} O(m \log n)$$

Overall:  $O(m \log(n))$

# DIJKSTRA'S OBSERVATIONS

Algorithm: Dijkstra's  $\Rightarrow$  Prim's Algorithm.

Let  $S$  be the set of explored nodes.

For each  $u \in S$ , ~~we store a distance~~  
value  $d(u)$ .

Initialize  $S = \{s\}$  ~~and define~~

**while**  $S \neq V$  **do**

Choose  $v \notin S$  with at least one  
incoming edge originating  
from a node in  $S$  with the  
smallest ~~distance~~  $e^* =$   
 $\min_{e=(u,v):u \in S} \{ \text{distance } e \}$

Append  $v$  to  $S$  ~~and define~~  
 $d(v) = d(u)$ .

**end**

# IMPLEMENTING KRUSKAL'S ALGORITHM

## Kruskal's (1956) Algorithm

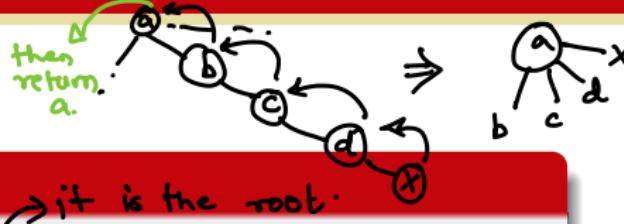
- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Key Operations

- Sorting the edges:  $O(m \log m)$  and, since  $m \leq n^2$ ,  $O(m \log n)$ . *from lowest cost to highest cost*
- Maintain sets of connected components that we merge.
- Initialize one set per node:  $O(n)$ . *sorting edges : preprocessing step* *basically cut property. (don't form cycle).*  
*each set initially will have 1 node each.*

- sets implemented with some sort of tree structure. == BST?*
- Union-Find Data Structure : disjoint set data structure.**
- Find(x): Finds the set containing  $x$ . ( $O(\log n)$ ) can be  $O(\alpha(n))$ . *Sets assumed to be organized in a tree. Name of a set is the root element of tree.*
  - Union(x,y): Joins two sets  $x$  and  $y$ . ( $O(1)$ )
- root of 1 tree*      *root of 2nd tree.*
- essentially constant.*

# UNION-FIND OPERATIONS



$\text{Find}(x): O(\log n)$

- If  $x.\text{parent}$  points to  $x$ , return  $x$ .
- Else  $\text{Find}(x.\text{parent}) \rightarrow$  keep going till you reach root, then return root.
- $O(\log n)$  requires balanced trees.  
↳ how to go from  $O(\log n) \rightarrow O(\alpha(n))$
- $O(\alpha(n))$  with path compression.

$\text{Union}(x, y): O(1)$

↳ you keep compressing the path as you go from node to root, to make finding easier

- (WLOG)  $x.\text{rank} \geq y.\text{rank}$ :  
 $y.\text{parent} = x$  } to unite, all you do is change the pointer.
- If  $x.\text{rank} = y.\text{rank}$ :  $\rightarrow$  if the rank of both is same  
 $x.\text{rank} := x.\text{rank} + 1 \rightarrow$  update the rank of  $x$ .
- By using rank, we maintain balanced sets if we start with balanced sets.



rank = height of tree.

# IMPLEMENTING KRUSKAL'S ALGORITHM

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Key Operations

- Sorting the edges:  $(O(m \log m))$  and, since  $m \leq n^2$ ,  $O(m \log n)$ .
- Maintain sets of connected components that we merge.
- Initialize one set per node:  $O(n)$ .

## Union-Find Data Structure

TH: How many Find and Unions?

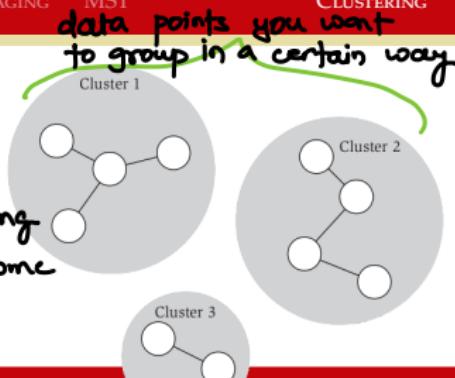
- $\text{Find}(x)$ : Finds the set containing  $x$ .
- $\text{Union}(x, y)$ : Joins two sets  $x$  and  $y$ .

*every edge  $m$  has  
2 $m$  points, so you  
find which set both  
endpoints are  
in.  
it is called  
2 $m$  times  
you process edges from  
lowest to highest*

## $k$ -CLUSTERING

goal: split  $n$  data points into  $k$  clusters.  
such that they have max spacing

if clusters are close, they are related in some way.



### Maximizing Spacing Problem

between clusters.

- A universe  $\underline{\mathcal{U}} := \{p_1, \dots, p_n\}$  of  $n$  objects. /  $n$  data points.
- Distance function  $\underline{d} : \underline{\mathcal{U}} \times \underline{\mathcal{U}} \rightarrow \mathbb{R}$  such that, for all  $p_i, p_j \in \mathcal{U}$ :

→ distance of a point to itself = 0.

$$\bullet d(p_i, p_i) = 0$$

→ distance from i to j is tve.

$$\bullet d(p_i, p_j) > 0 \quad \text{→ symmetry}$$

$o, o_1, o_2, \dots, o_n$  | on  $n$  objs  
 $k_1, k_2, k_3, \dots, k$  bars

we need to find partitioning  
such that it maximizes spacing.

- Objective: Partition  $\mathcal{U}$  into  $k$  non-empty groups  $\mathcal{C} := C_1, \dots, C_k$  with maximum spacing:

maximize  
spacing.

$$\text{maximize} : \min_{C_i, C_j \in \mathcal{C}} \min_{u \in C_i, v \in C_j} d(u, v)$$

distance b/w  
pairs.

distance between  
clusters.  
 $C_i, C_j$

# ALGORITHM DESIGN

TopHat Discussion 4: What greedy approach might work?  
↳ Kruskal's

run this till you have  $k$  connected components.

# ALGORITHM DESIGN

## Algorithm

- Build an MST.
- Remove  $k - 1$  largest edges.

according to Kruskal's, max cost edges will be left.  
k-Clusters at max spacing?      ↗ this will be the max  
distance between clusters.

- Start with a tree, remove  $k - 1$  edges: We get a forest of  $k$  trees.
- By definition largest edges are removed so max spacing.

## TopHat Q10: Which MST algorithm?

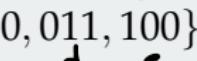
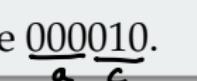
Kruskal's ( $O(m \log n)$  which is  $O(n^2 \log n)$  for clustering):

- Merge sets from lowest to most expensive edges.
- Stop when we have  $k$  sets.

# BINARY ENCODING

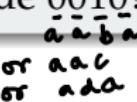
## Fixed-Width Encoding

*you want to represent this as 0's and 1's.*

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^k$ .  
 $\gamma(S) := \{000, 001, 010, 011, 100\}$ .  

- Ex. ASCII
- TopHat Q11: Decode 000010.  


*wishes*

## Variable-Width Encoding

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^*$ .  
 $\gamma(S) := \{0, 1, 10, 01, 11\}$ .  
*→ minimizing bits used*
- TopHat Q12: How many ways to decode 0010?  


# UNIQUE VARIABLE-WIDTH ENCODINGS

## Prefix Codes

Encoding of  $S$  such that no encoding of a symbol in  $S$  is a prefix of another.

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^*$ .  
 $\gamma(S) := \{11, 01, 001, 000, 100\}$ .
- 0010 invalid sequence
- TopHat 13: Decode 1101.

→ read symbols till it matches with your prefix coding.

# UNIQUE VARIABLE-WIDTH ENCODINGS

## Prefix Codes

Encoding of  $S$  such that no encoding of a symbol in  $S$  is a prefix of another.

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^*$ .  
 $\gamma(S) := \{11, 01, 001, 000, 100\}$ .

## Easy Decoding

Scan left to right, once an encoding is matched, output symbol.

## Optimal Prefix Codes

*→ we need to come up with an optimal prefix code.*

- For a set of symbols  $S$ , let  $f_x$  denote the frequency of  $x$  in the text to be encoded.
- Average bits  $\text{ABL}(\gamma) := \sum_{x \in S} f_x \cdot |\gamma(x)|$ .
- Goal: Find  $\underline{\gamma}$  that minimizes  $\text{ABL}$ . } greedy solution.

# DIVIDE AND CONQUER

→ recursion.

but use iterative approach  
as recursion takes more memory and time.

# DIVIDE AND CONQUER (DC)

for this, iterative is more optimal than recursive.

because recursive needs

memory allocation that slows it down while iterative only has to increment the counter.

## Overview

- Split problem into smaller sub-problems.
- Solve (usually recurse on) the smaller sub-problems.
- Use the output from the smaller sub-problems to build the solution.  
eg: Merge sort.

to solve larger problems

## Tendencies of DC

- Naturally recursive solutions
- Solving complexities often involve recurrences.
- Often used to improve efficiency of efficient solutions, e.g.  $O(n^2) \rightarrow O(n \log n)$ . To improve runtime
- Used in conjunction with other techniques.

figuring out runtime involves recurrences.

# SEARCHING

## Linear Search

- Brute force approach: check every item in order.
- Time complexity:  $O(n)$  . linear time.

## Divide and Conquer Approach

- Binary Search <sup>has to be sorted</sup> useful if you will be doing multiple searches.
- Complexity:  $O(\log n)$  <sup>faster.</sup>

# SORTING

$\rightarrow n$  items to put them in order.

Ordering some (multi)set of  $n$  items.

## Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$  } time complexity of brute force

Simple Sorts  $\rightarrow$  taking an unordered list and removing inversions.  
 $n$  items, you invert every pair, so number of inversions  $= n^2$

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$  ↘ time complexity.

## Efficient Sorts

- Divide & Conquer: Quick Sort ( $O(n^2)$ ), Merge Sort ( $O(\underline{n} \log \underline{n})$ )

# SORTING

Ordering some (multi)set of  $n$  items.

## Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

## Efficient Sorts

- Divide & Conquer: Quick Sort ( $O(n^2)$ ), Merge Sort ( $O(n \log n)$ )

## Trick Sorts

$\rightarrow$  CS 400

$k = \max$  key size of what you are sorting.

good  
long as  
you know  
the key  
size  
(should  
be small).

- Radix Sort ( $O(n \lceil \log k \rceil)$ ), Counting Sort ( $O(n + k)$ )
- $k$  is the maximum key size.
- TopHat 5: What value of  $k$  would make both sorts have unbounded time complexity no better than Merge Sort?  $\Omega(n \log n)$

the radix/  
counting sort  
would not be  
useful.

**Stable sort:**

**MERGESORT** — when you get the sorted list back, if two items are equal, the relative position of these items doesn't change.

**Algorithm: MERGESORT**

**Input :** A list  $A$  of  $n$  comparable items.

**Output:** A sorted list  $A$ .

**if**  $|A| = 1$  **then return**  $A \rightarrow$  Base case  $T(1) \leq c$

*split list in two (half)*  $\left[ \begin{array}{l} A_1 := \text{MERGESORT}(\text{Front-half of } A) \\ A_2 := \text{MERGESORT}(\text{Back-half of } A) \end{array} \right] \text{recursive } \Rightarrow 2 \times T\left(\frac{n}{2}\right)$   
**call**  
**return**  $\text{MERGE}(A_1, A_2)$   $O(n)$  — calling the other method.  
*merge them into the  $n$ -length list that's sorted.*

**Algorithm: MERGE**

**Input :** Two lists of comparable items:  $A$  and  $B$ .

**Output:** A merged list.

Initialize  $S$  to an empty list.

**while** either  $A$  or  $B$  is not empty **do**

  | Pop and append  $\min\{\text{front of } A, \text{front of } B\}$  to  $S$ .

**end**

**return**  $S$

*to maintain stability in the while loop!*

*if front of  $A$  = front of  $B$ ,*

*you prioritize element in  $A$ .*

*front of  $A$  or  $B$  whichever is smaller.*

*But you prioritize elements in  $A$  to ensure stability.*

TopHat 6: What is the complexity of MERGE?  $O(n)$

*at most items in both lists combined.*  
*↳ while loop:*

for normal induction, you only assume  $k$  and try to prove  $(k+1)$ . But for strong induction, you assume  $1, \dots, (k-2), (k-1), k$  and prove  $(k+1)$ .

### Algorithm: MERGESORT

$$K \rightarrow K/2 \rightarrow K/4 \rightarrow K/8$$

**Input :** A list  $A$  of  $n$  comparable items.

**Output:** A sorted list  $A$ .

**if**  $|A| = 1$  **then return**  $A$

$A_1 := \text{MERGESORT}(\text{Front-half of } A)$  } inductive hypothesis -  $A_1, A_2$  return  
 $A_2 := \text{MERGESORT}(\text{Back-half of } A)$  } to be sorted.  
**return**  $\text{MERGE}(A_1, A_2)$

you need all of these  
to be correct, so strong  
induction.

**Assumption:** Previous one to the base case, all of them are  
**Program Correctness:** correct with strong induction, your proof becomes coherent.

Because

① Soundness: List  $A$  is sorted after call to MERGESORT.

Proof: By strong induction on list length:  $\rightarrow$  as recursion takes you one step back.

**Base case:**  $k = 1$ : List is sorted.

**Inductive step:** By ind hyp,  $A_1$  and  $A_2$  are sorted, and, then, by definition, MERGE will produce a sorted list.

**inductive hypothesis:** our recursive calls give us the correct answer.  
 $A_1, A_2$  are sorted.

# MERGESORT

---

## Algorithm: MERGESORT

---

**Input :** A list  $A$  of  $n$  comparable items.

**Output:** A sorted list  $A$ .

**if**  $|A| = 1$  **then return**  $A$  ]  $\tau(1)$

$A_1 := \text{MERGESORT}(\text{Front-half of } A)$  ]  $2\tau(n/2)$

$A_2 := \text{MERGESORT}(\text{Back-half of } A)$

**return**  $\text{MERGE}(A_1, A_2)$

$\uparrow O(n)$

---

## Run time Considerations:

- Cost to  $\text{MERGE}$ :  $O(n)$ .  
*(call merge procedure.)*

- Recurrences: 2 calls to MERGESORT with lists half the size.

① Identify recursive calls.

② Know how big each recursive calls are.

# MERGESORT RECURRENCE

*worst time*

*Base case.*

*don't use Big(O) for recurrence relation*

$$\left\{ \begin{array}{l} T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c \\ \text{recursive calls.} \\ \text{cost to call merge.} \end{array} \right. \quad \text{upper bound}$$

## Notes

*length of n = odd (uneven split).*

- More precise:  $T(n) \leq T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + cn$
- Usually, we can asymptotically ignore floor and ceilings.
- Essentially, we are assuming  $n$  is a power of 2.
- Alternate form:  $T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + O(n); T(1) \leq O(1)$

## Methods

- main*
- Unwind / Recurrence Tree
  - Guess
  - Master Theorem
  - Nuclear Bomb Theorem / Master Master Theorem

# UNWIND MERGESORT RECURRENCE

Start with input and unwind your way to base case.

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\ &\leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\ &\leq 2\left(2\left(2T\left(\frac{n}{8}\right) + c\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \end{aligned}$$

⋮

→ pattern

$$\begin{aligned} &\leq 2^k T\left(\frac{n}{2^k}\right) + kcn \quad \xrightarrow{\text{recurrence ends at } T(1)} \\ &= nT(1) + cn \log(n) \\ &= \cancel{cn} + cn \log n \\ &= O(n \log(n)) \end{aligned}$$

dominant term,  
runtime.

Base case!

$$1 = \frac{n}{2^k}$$

$$\iff 2^k = n$$

$$\iff k = \underbrace{\log_2(n)}$$

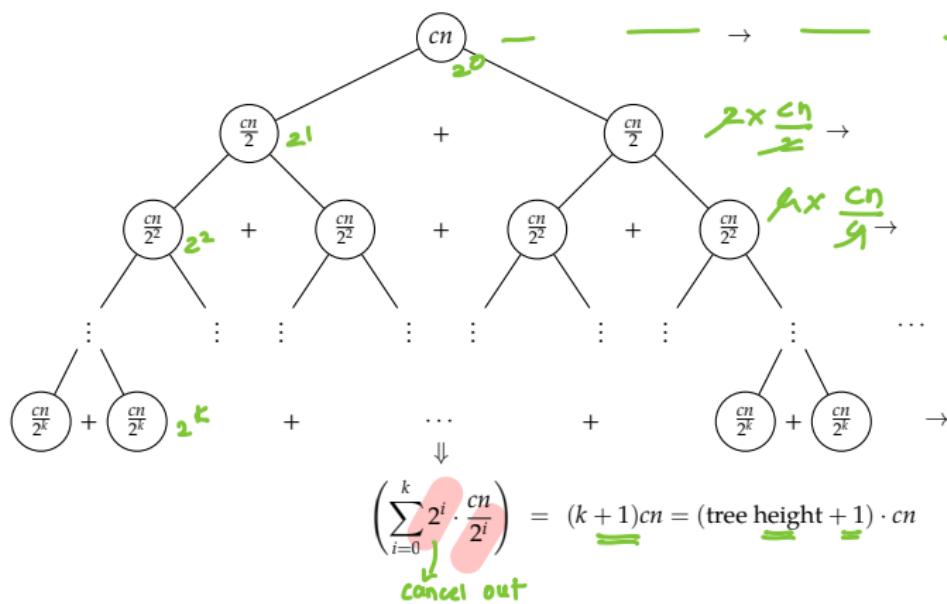
recurrence ends  
at  $T(1)$

$$\begin{aligned} &2^{\log_2(n)} T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2(n) \times c \times n \\ &= n \times T\left(\frac{n}{n}\right) + \log_2(n) \times c \times n \\ &= n \cdot T(1) + cn \log(n) \end{aligned}$$

## RECURSION TREE METHOD

$x = \text{tree height}$

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$



each layer has total cost of  $\text{c}_n$ .

$$\pi = \log n$$

$$= cn \lg n + cn$$

$$= O(n \cdot \lg n)$$

<sup>1</sup>Based on: <http://www.texample.net/tikz/examples/merge-sort-recursion-tree/>

# PROVE RECURRENCE BY STRONG INDUCTION

*start induction  
at n = 2.*

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn \leq \underbrace{cn \lg n + cn}_{\text{come up with a function that is Big-OH of what you are trying to show.}}; T(1) \leq c$$

**Base Case:**  $n = 2$ .

$$\begin{aligned} T(2) &= 2 \cdot T(1) + 2c \leq 4c \\ &= c \cdot 2 \lg 2 + 2c \end{aligned}$$

**Inductive step:**

$$\begin{aligned} T(k) &= 2 \cdot T(k/2) + ck \\ &\leq 2 \left( \frac{ck}{2} \lg \frac{k}{2} + \frac{ck}{2} \right) + ck \\ &= ck \lg(k/2) + 2ck \\ &= ck \lg k - ck + 2ck \\ &= ck \lg k + ck \end{aligned}$$

$$\therefore O(n \log n)$$

# GENERALIZED RECURRENCE

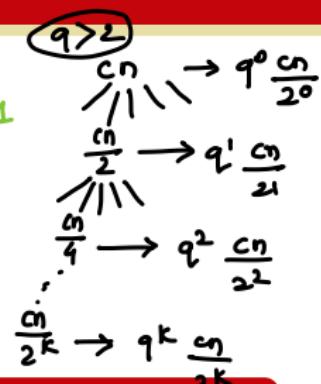
↳ for merge sort.

$$r^{\log n} = n^{\log r}$$

$$r > 1, n > 1$$

$$T(n) \leq q \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

$\overline{q}$   
coefficient = no. of recursive calls.



Case  $q > 2$

$$O(n^{\lg q})$$

$$T(n) = \sum_{i=0}^k q^i \frac{cn}{2^i} = cn \sum_{i=0}^k \left(\frac{q}{2}\right)^i$$

GP:  $r = q/2$

$$= cn \left( \frac{r^{k+1} - 1}{r - 1} \right)$$

$$\leq cn \left( \frac{(r^{\lfloor \lg q \rfloor + 1} - 1)}{r - 1} \right)$$

infinite GP sum.

Case  $q = 2$  → merge sort.

$$O(n \log n)$$

Case  $q = 1$

$$O(n)$$

recurrence tree.

A recurrence tree diagram for  $q = 1$ . The root node is labeled  $0$ . It branches into  $1$  child, which is also labeled  $0$ . This pattern repeats, with every node labeled  $0$ . A green bracket labeled "tree." groups all nodes under the root.

$$= \frac{r \cdot cn}{r - 1} r^{\log n}$$

$$= O(n \cdot n^{\log 1})$$

$$= O(n \cdot n^{\log(1/2)})$$

$$= O(n \cdot n^{\log 2 - 1})$$

$$= O(n^{\log 2})$$

→ implement for week 5.  
runtime solution.

# INVERSION COUNT

---

# COUNTING INVERSIONS

Inversion  $\rightarrow$  pair of items out of order.

Given a list  $A$  of comparable items. An inversion is a pair of items  $(a_i, a_j)$  such that  $a_i > a_j$  and  $i < j$ , where  $i$  and  $j$  are the index of the items in  $A$ .

$\rightarrow a_i$  occurs before  $a_j$  in the list.

## Inversion Count

Count the number of inversions in a list  $A$ , containing  $n$  comparable items.



### Exercise – Teams of 2 or 3

- Solve the problem in  $\Theta(n^2)$ .
- Solve the problem in  $O(n \log n)$ .
- Prove correctness and complexity.

$\rightarrow$  check all pairs. if  $B \leq A$ , entire  $A$  is inversion.  
 $\rightarrow$  sorting. Based on merge. if you are picking elements from  $B$ , you know there is an inversion.

## PART 2: GIVE AN $O(n \log n)$ SOLUTION.

$\curvearrowright$  modified mergesort.

---

**Algorithm:** COUNTSORT

---

**Input :** A list  $A$  of  $n$  comparable items.

**Output:** A sorted array and the number of inversions.

**if**  $|A| = 1$  **then return**  $(A, 0)$   $\rightarrow$  base case.

$(A_1, c_1) := \text{COUNTSORT}(\text{Front-half of } A) -$  ] proof of correctness:  
 $(A_2, c_2) := \text{COUNTSORT}(\text{Back-half of } A) -$  induction (strong).  
 $(A, c) := \text{MERGECOUNT}(A_1, A_2)$  ] modified merge

**return**  $(A, c + c_1 + c_2)$

---

## PART 2: GIVE AN $O(n \log n)$ SOLUTION.

### Algorithm: MERGECOUNT

**Input :** Two lists of comparable items:  $A$  and  $B$ .

**Output:** A merged list and the count of inversions.

Initialize  $S$  to an empty list and  $c := 0$ .

**while either  $A$  or  $B$  is not empty do**

Pop and append  $\min\{\text{front of } A, \text{front of } B\}$  to  $S$ .

**if Appended item is from  $B$  then**

$c := c + |A|$ .       $=$

**end**       $\hookrightarrow$  All items in  $A$  are inversions.

**end**

**return**  $(S, c)$

### Analysis

$\rightarrow$  mergecount works  $\rightarrow$  show this.

- Correctness: Need to show that the inversions are counted.
- Complexity: Same recurrence as MERGESORT.

# LINEAR TIME SELECTION

Talmost quick sort.

# LINEAR TIME SELECTION

Given an unordered array of  $n$  comparable items, find the  $k^{\text{th}}$  value if the list were to be sorted.

## Problem

Find the  $\underline{k^{\text{th}}}$  value in an unsorted array  $A$  of  $n$  numbers if  $A$  were sorted.

$n \log n$   $\xrightarrow[\text{sort then find}]{}$   $n$  (without sorting).  
linear time.

### Algorithm: QUICKSELECT

$$1 \leq k \leq n$$

Input : A array  $A[1..n]$  and an int  $k$ .

Output: The  $k^{\text{th}}$  element of  $A$  if  $A$  were sorted.

if  $n = 1$  then return  $A[1]$  : base case

Choose a pivot  $A[p] \rightarrow$  arbitrary pivot.

$r := \text{PARTITION}(A[1..n], p)$

if  $k < r$  then (left)

| return  $\text{QUICKSELECT}(A[1..r - 1], k)$

else if  $k > r$  then (right) .

| return  $\text{QUICKSELECT}(A[r + 1..n], k - r)$

else

| return  $A[r] \rightarrow$  return pivot.

end

items in the list smaller than pivot will be on the left and all items larger than pivot will be on the right.

# QUICKSELECT RECURRENCE

$$T(n) \leq \max_{1 \leq r \leq n} \max \{ T(r-1), T(n-r) \} + cn$$

↗ worst case, max/min item in array.  
 ↗ worst case.  
cn  
 cost of single call. to quickselcet.

## Algorithm: QUICKSELECT

**Input :** A array  $A[1..n]$  and an int  $k$ .

**Output:** The  $k$ th element of  $A$ .

if  $n = 1$  then return  $A[1]$

Choose a pivot  $A[p]$

$r := \text{PARTITION}(A[1..n], p)$  →  $r$  tells us where the partition is.  
so, we can base our reasoning around  $r$ .

if  $k < r$  then

return  $\text{QUICKSELECT}(A[1..r-1], k)$   
 $r-1$  items.

① if  $r = k$ ,  
we found the answer

else if  $k > r$  then

return  $\text{QUICKSELECT}(A[r+1..n], k-r)$

②  $k < r$   
look at front part of list.

else

return  $A[r]$  ①

$n - (r+1) + 1$

$n - r - 1 + 1$   
 $(n-r)$  items.

excluding  $(r)$

excluding first half of list.

end

# ~~QUICKSELECT~~ RECURRENCE

~~Quicksort~~:

$$T(n) \leq \max_{1 \leq r \leq n} \max\{T(r - 1), T(n - r)\} + cn$$

**Algorithm:** ~~QUICK~~  
~~SORT~~

**Input :** A array  $A[1..n]$  and an int  $k$ .

**Output:** The  $k$ th element of  $A$ .

**if**  $n = 1$  **then return**  $A[1]$

Choose a pivot  $A[p]$

~~Merge~~ PARTITION( $A[1..n], p$ )

~~if~~  $k < r$  **then**

**return** QUICKSELECT( $A[1..r - 1], k$ )

~~else if~~  $k > r$  **then**

**return** QUICKSELECT( $A[r + 1..n], k$ )

~~else~~

**return**  $A[p]$

**end**

# QUICKSELECT RECURRENCE

*→ pivot selection is making it  $n^2$ .*

$$T(n) \leq \max_{1 \leq r \leq n} \max\{T(r-1), T(n-r)\} + cn$$

$$\leq \max_{1 \leq \ell \leq n-1} T(\ell) + cn$$

$$\leq T(n-1) + cn = T(n-2) + c(n-1) + cn$$

$$\in O(n^2)$$

$$= \sum_{i=1}^n i \quad \left. \right\} n^2.$$

$$= \frac{i(i+1)}{2} = i^2 = n^2.$$

# MEDIAN OF MEDIAN

## Algorithm: MOMSELECT

**Input :** A array  $A[1..n]$  and an int  $k$ .

**Output:** The  $k$ th element of  $A$ .

**if**  $n$  is small **then** Solve by brute force.  $\text{①}$



$m := \lceil n/5 \rceil \rightarrow$  Break list into chunks of 5.  
for  $i := 1$  to  $m$  do

**②** \* trying to find the middle/halfway point.

figuring out the end partition.  $M[i] :=$  brute force find median of  $A[5i - 4..5i]$   $O(n)$ .  
 $\rightarrow M$  array consists of medians of groups of 5.

**②** mom := MOMSELECT( $M[1..m], [m/2]$ )  $\leftarrow T(n/5)$

$r := \text{PARTITION}(A[1..n], \text{mom}) O(n) \rightarrow$  recursive call on  $M$ : median of those medians.

**quick select.** if  $k < r$  then  
| return MOMSELECT( $A[1..r - 1], k$ )  
else if  $k > r$  then  
| return MOMSELECT( $A[r + 1..n], k - r$ )  
else  
| return  $A[r]$   
end

$\left. \right\} T(7n/10)$

# MomSelect ANALYSIS

## MomSelect Pivot

- greater and less than  $> \lfloor [n/5]/2 \rfloor - 1 \approx n/10$  medians.
  - Therefore, MomSelect Pivot is greater and less than  $3n/10$  items.
  - So, worst-case partition size is  $\underline{\underline{7n/10}}$ .
- $= m$   
 $M = n/10$   
  
 $m = n/5$

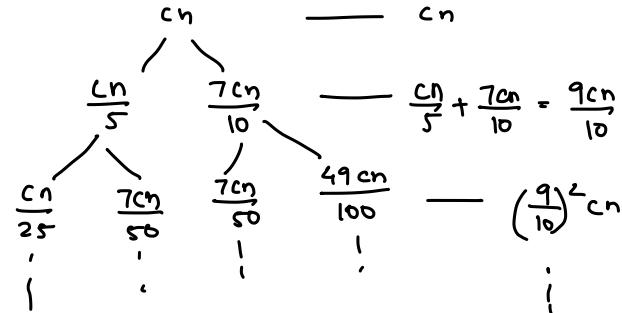
Recurrence:

$$T(n) \leq \underbrace{T(n/5)}_{\substack{\text{recursive call} \\ \downarrow \\ \text{mom} \\ \text{call} \\ \text{recursion}}} + \underbrace{T(7n/10)}_{\substack{\text{one side. : worst case.} \\ \downarrow \\ \text{quick} \\ \text{select} \\ \text{recursion}}} + cn \in O(n)$$

$\overline{T}$   
 linear time cost  
 to do a single call.  
 (work per call)

To prove Momselect is  $O(n)$ :

Recurrence Tree:



$$\begin{aligned} T(n) &\leq \sum_{i=0}^k \left(\frac{9}{10}\right)^i c_n \\ &\leq c_n \sum_{i=0}^{\infty} \left(\frac{9}{10}\right)^i = c_n \times \frac{\frac{9}{10}}{1 - \frac{9}{10}} = c_n \times \frac{9}{10} \times \frac{10}{1} = 9c_n \leq 10c_n. \\ &\leq 10c_n \leq O(n) \end{aligned}$$

sum it  
infinite  
GP

$$= \frac{a}{1-r}$$

# INTEGER MULTIPLICATION

# INTEGER MULTIPLICATION

→ multiplying 2 integers.

Partial Product Method:

$$\begin{array}{r}
 & 1100 \\
 & \times 1101 \\
 \hline
 12 & 0001100 \\
 \times 13 & 0000000 \\
 \hline
 036 & 0110000 \\
 +12\cancel{0} & 1100000 \\
 \hline
 156 & 10011100
 \end{array}$$

→ minimize no. of bitwise operations needed

Problem

same length.

Multiple two  $n$ -length binary numbers  $x$  and  $y$ , counting every bitwise operation.

$$\boxed{2n^2} = n^2$$

TopHat 8: What is the complexity of the partial product method?  $O(n^2)$ .

DIVIDE & CONQUER V1 *→ best way to start is splitting into two.*  
 Eg:  $\frac{1101}{n=4} = 11 \times 2^2 + 01$

High and low bits

$$\text{Q2: } 6582 = 65 \times 10^2 + 82$$

Consider  $x = x_1 \cdot 2^{n/2} + x_0$  and  $y = y_1 \cdot 2^{n/2} + y_0$ .  
 Base 2.

$$\begin{aligned} xy &= (\underbrace{x_1 \cdot 2^{n/2} + x_0}_{\text{1}})(\underbrace{y_1 \cdot 2^{n/2} + y_0}_{\text{2}}) \\ &= x_1 y_1 \cdot 2^n + (\underbrace{x_1 y_0 + x_0 y_1}_{\text{3}}) \cdot 2^{n/2} + x_0 y_0 \quad \text{Answer.} \end{aligned}$$

- How many recursive calls? 4.  $n/2 = \text{size of all recursive calls.}$
- Cost per call?  $O(n)$
- What is the size of the recursive calls?  $n/2$ .
- What is the recurrence?

$$T(n) \leq \hat{4}T(\overline{n/2}) + \hat{cn} = O\left(\underline{\underline{n^{\lg 4}}}\right) = O\left(\underline{\underline{n^2}}\right)$$

*no. of calls. size*      *cost per call.*      *reduce no. of recursive calls.*

# DIVIDE & CONQUER V2

## High and low bits

Consider  $x = x_1 \cdot 2^{n/2} + x_0$  and  $y = y_1 \cdot 2^{n/2} + y_0$ .

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0 \end{aligned}$$

Hint:  $(x_1 + x_0)(y_1 + y_0) = x_1y_1 + \underline{x_1y_0 + x_0y_1} + x_0y_0$ .  
combine these terms.

Exercise: Design an algorithm with 3 Recursive Calls

- Recursions:
  - $p := \text{intMult}(x_1 + x_0, y_1 + y_0)$
  - $x_1y_1 := \text{intMult}(x_1, y_1)$
  - $x_0y_0 := \text{intMult}(x_0, y_0)$
- Combine: Return  $x_1y_1 \cdot 2^n + (p - \underline{x_1y_1} - \underline{x_0y_0}) \cdot 2^{n/2} + x_0y_0$
- Recurrence:  $T(n) \leq 3T(n/2) + O(n) = O(n^{\lg 3}) = O(n^{1.59})$

→ more geometric.

# CLOSEST PAIRS

---

---

# FINDING THE CLOSEST PAIR OF POINTS

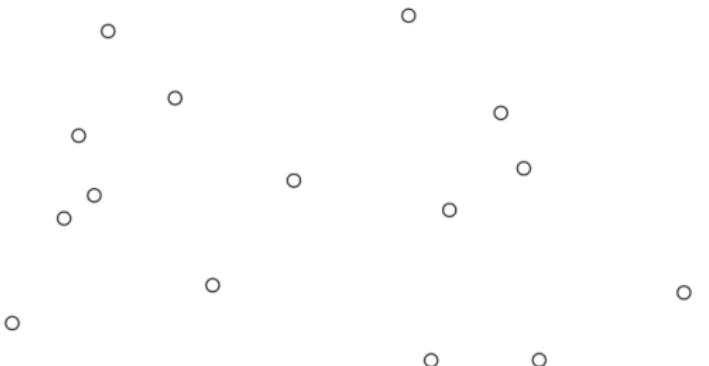
Euclidian Plane:

2D space:



1D version

(closest points on  
the line)



→ goal : find pair of points closest to each other

## Problem

Given a set of  $n$  points,  $\mathcal{P} = \{p_1, p_2, \dots, p_n\}$ , in the plane. Find the closest pair. That is, solve  $\arg \min_{(p_i, p_j) \in \mathcal{P}} \{d(p_i, p_j)\}$ , where  $d(\cdot, \cdot)$  is the Euclidean distance.

→ calculate all distances, take min.

What is the  $O(n^2)$  solution?

## 2-D CLOSEST PAIR

DIVIDE AND CONQUER

① Divide: Split point set (in half?).

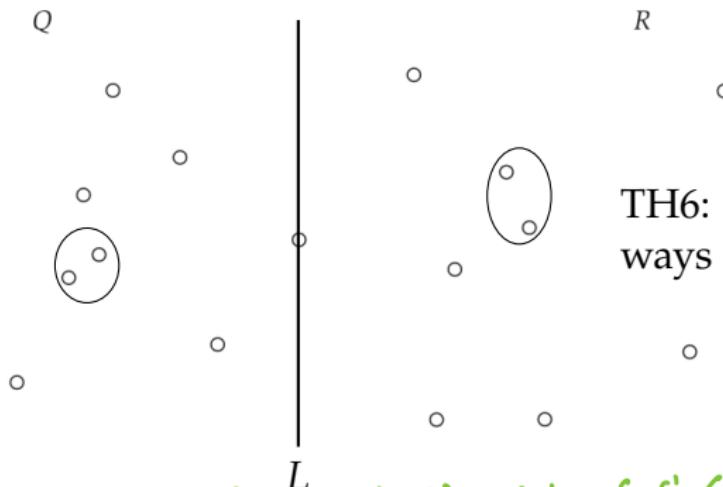
② Conquer: Find closest pair in each partition. } <sup>recursion.</sup>

③ Combine: Merge the solutions.

→ kind of like merge sort.

### 3. COMBINE THE SOLUTIONS.

$S$  is sorted based on  $y$  co-ordinates.



TH6: Are one of these always the minimum of  $\mathcal{P}$ ?

if there exists a set of points  $s, s' \in S$  that happen to be closer than  $\delta$ , then these points are within 15 positions of each other; if you consider points in  $S$  sorted by  $y$  co-ordinate.

#### Lemma 3

Let  $S$  be the set of points within  $\delta$  of  $L$ . If there exists a  $s, s' \in S$  and  $d(s, s') < \delta$ , then  $s$  and  $s'$  are within 15 positions of each other in  $S_y$ .

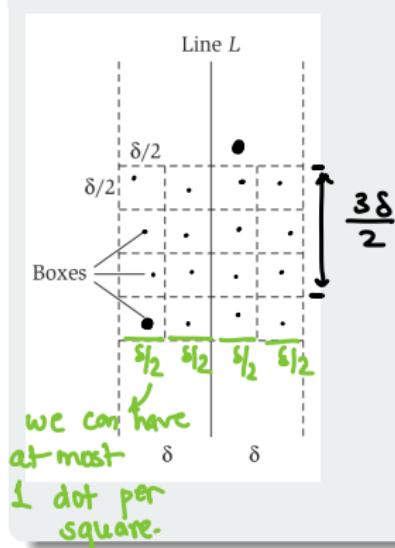
### 3. COMBINE THE SOLUTIONS.

*prove Lemma 8-*

#### Lemma 3

Let  $S$  be the set of points within  $\delta$  of  $L$ . If there exists a  $s, s' \in S$  and  $d(s, s') < \delta$ , then  $s$  and  $s'$  are within 15 positions of each other in  $S_y$ .

#### Proof.



- Partition  $\delta$ -space around  $L$  into  $\delta/2$  squares.
- At most 1 point per square else contradicts definition of  $\delta$ .
- By way of contradiction, say  $d(s, s') < \delta$  and  $s$  and  $s'$  separated by 16 positions.
- By counting argument,  $s$  and  $s'$  are separated by 3 rows which is at least  $3\delta/2$ . □

# COMPLETING THE ANALYSIS

## Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

## Runtime of the Algorithm

- Sorting by  $x$  and by  $y$  ( $O(n \log n)$ ).
- How many recursive calls? 2.
- What is the size of the recursive calls?  $n/2$ .
- Work per call: check points in  $S$ .
  - $15 \cdot |S| = O(n)$ . } checking boundary condition in linear time.
- What is the recurrence?

$$T(n) \leq 2T(n/2) + cn = \underbrace{O(n \log n)}_{\text{runtime}} .$$

# MAX SUBARRAY

---

# MAX SUBARRAY

## Problem

Given an array  $A$  of integers, find the (non-empty) contiguous subarray of  $A$  of maximum sum.

touching each other.

↳ there may be -ve values.

## Exercise – Teams of 3 or so

need  
to  
go  
from  
quadratic

to  
 $n \log n$ .

- Solve the problem in  $\Theta(n^2)$ .
- Solve the problem in  $O(n \log n)$ .

• Prove correctness and complexity.

# PART 1: GIVE A $\Theta(n^2)$ SOLUTION.



## Algorithm: CHECKALLSUBARRAY

**Input :** Array  $A$  of  $n$  ints.

**Output:** Max subarray in  $A$ .

Let  $M$  be an empty array

**for**  $i := 1$  to  $\text{len}(A)$  **do**

**for**  $j := i$  to  $\text{len}(A)$  **do**

**if**  $\text{sum}(A[i..j]) > \text{sum}(M)$  **then** :

$M := A[i..j]$

**end**

**end**

**end**

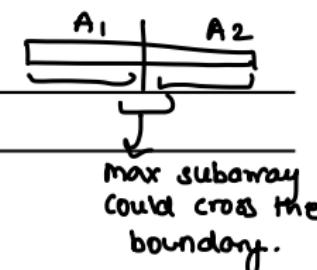
**return**  $M$

## Analysis

- Correct: Checks all possible contiguous subarrays.
- Complexity:
  - Re-calculating the sum will make it  $O(n^3)$ . Key is to calculate the sum as you iterate.
  - For each  $i$ , check  $n - i + 1$  ends. Overall:

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} = \Theta(n^2)$$

something like mergesort recurrence.  
**PART 2: GIVE AN  $O(n \log n)$  SOLUTION.**




---

### Algorithm: MAXSUBARRAY

---

**Input :** Array  $A$  of  $n$  ints.

**Output:** Max subarray in  $A$ .

**if**  $|A| = 1$  **then return**  $A[1]$   $\rightarrow$  Base case.

$A_1 := \text{MAXSUBARRAY}(\text{Front-half of } A)$

$A_2 := \text{MAXSUBARRAY}(\text{Back-half of } A)$

$M := \text{MIDMAXSUBARRAY}(A) \rightarrow$  checking middle ground.

**return** Array with max sum of  $\{A_1, A_2, M\}$

---

### Algorithm: MIDMAXSUBARRAY

---

**Input :** Array  $A$  of  $n$  ints.

**Output:** Max subarray that crosses midpoint  $A$ .

$m :=$  mid-point of  $A$



$m$  (midpoint)  
 $L$  combine  $(L, R)$  .

$L :=$  max subarray in  $A[i, m - 1]$  for  $i = m - 1 \rightarrow 1$

$R :=$  max subarray in  $A[m, j]$  for  $j = m \rightarrow n$

**return**  $L \cup R$  // subarray formed by combining  $L$  and  $R$ .

---

## PART 2: GIVE AN $O(n \log n)$ SOLUTION.

### Algorithm: MAXSUBARRAY

**Input :** Array  $A$  of  $n$  ints.

**Output:** Max subarray in  $A$ .

**if**  $|A| = 1$  **then return**  $A[1]$

$A_1 := \text{MAXSUBARRAY}(\text{Front-half of } A)$  ] size  $(\frac{m}{2})$

$A_2 := \text{MAXSUBARRAY}(\text{Back-half of } A)$  ] size  $(\frac{m}{2})$

$M := \text{MIDMAXSUBARRAY}(A) \leftarrow O(n)$

**return** Array with max sum of  $\{A_1, A_2, M\}$

### Analysis

*→ strong induction.*

- Correctness: By induction,  $A_1$  and  $A_2$  are max for subarray and  $M$  is max mid-crossing array.
- Complexity: Same recurrence as MERGESORT.

# CS 577 - Dynamic Programming



Marc Renault

Department of Computer Sciences  
University of Wisconsin – Madison

Fall 2023

TopHat Section 001 Join Code: 477366

TopHat Section 002 Join Code: 560750



**WISCONSIN**  
UNIVERSITY OF WISCONSIN-MADISON

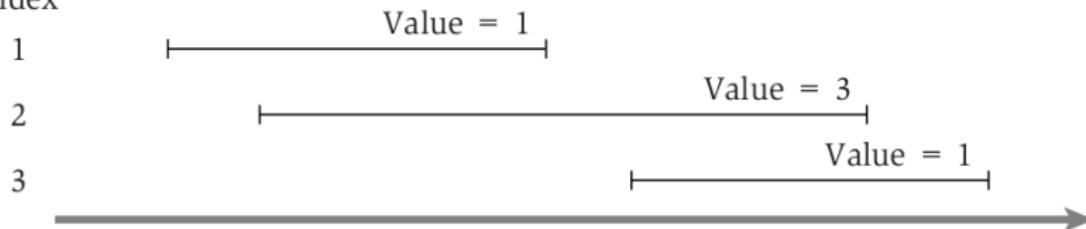
[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN\\*](#)[LS\\*](#)[RNA\\*](#)

# WEIGHTED INTERVAL SCHEDULING

→ greedy interval scheduling is a special case of DP interval scheduling.

# WEIGHTED INTERVAL SCHEDULING

Index



## Problem Definition

- Requests:  $\sigma = \{r_1, \dots, r_n\}$
- A request  $r_i = (s_i, f_i, v_i)$ , where  $s_i$  is the start time,  $f_i$  is the finish time, and  $v_i$  is the value.
- Objective: Produce a *compatible* schedule  $S$  that has maximum value. *no. of intervals doesn't matter, but you need the max value.*
- Compatible schedule  $S$ :  $\forall r_i, r_j \in S, f_i \leq s_j \vee f_j \leq s_i$ .

TH1: What is the value of the FF heuristic? 2.

1, 3. value = 2

TH2: What is the optimal value? 3. 2 value = 3

sorted  $\sigma = \langle r_1, r_2, r_3, \dots, r_j \rangle$

## RECURSIVE SOLUTION

*you're basically going one step back every time till you get to the 1st request.*

### Recursive Procedure

*sort them by finish time*

subprobs:  
possibilities  
et how  
you can  
get to the  
optimal sol.

① Assume  $\sigma$  ordered by finish time (asc).

*what this recursion*

② Find the optimal value in sorted  $\sigma$  of first  $j$  items: *is about*.

- ① Find largest  $i < j$  such that  $f_i \leq s_j$ . *finish time of i is less than start time of j.*
- ②  $\text{OPT}(j) = \max(\text{OPT}(j-1), \text{OPT}(i) + v_j)$

*dichotomy: Any given input is either in the solution or not in the solution.*

### Proof of optimality.

*Strong induction on input size.*

By strong induction on  $j$ .

Base cases:  $j = 0$  or  $j = 1$ : Only 1 possible optimal solution.

Inductive step:  *$\sigma$  has only one request / no requests.*

- By ind hyp, we have opt for  $j-1$  and opt for  $i$ .
- FF assures the dichotomy that the last interval is either in the solution or not.  
*as you need max value.*
- Take the max of whether or not a given interval is included.

# CONSIDER THE RECURSION

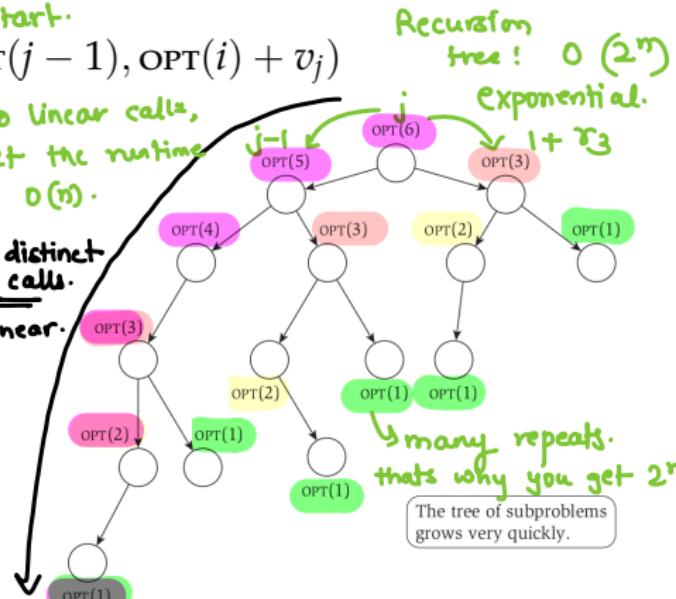
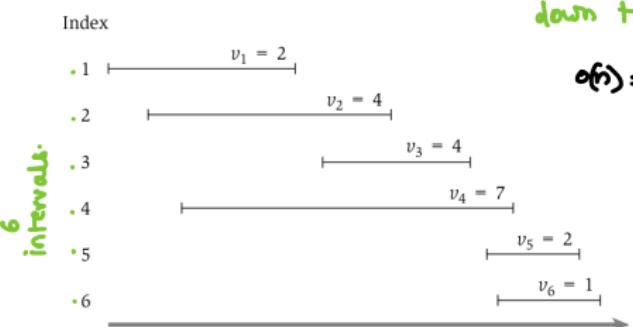
$$\sigma = \langle r_1, r_2, r_4, r_5, r_6 \rangle$$

*start.*

$$\text{OPT}(j) = \max(\text{OPT}(j-1), \text{OPT}(i) + v_j)$$

if you do linear calls,  
you can get the runtime  
down to  $O(n)$ .

$\text{of}(n) = \underline{\text{distinct}} \text{ calls.}$   
linear.



TH3: What is the asymptotic number of recursive calls with  $n$  jobs?  $O(2^n)$

# MEMOIZING THE RECURSION

→ iterative method for recursion : you remember the previous values and store them to look-up as use for later instead of recalculating

## Memoization

- Not a typo.
- Coined in 1989 by Donald Michie.
- Derived from latin “memorandum”, meaning “to be remembered”.

## Basic Technique

- Calculate once: store the value in array and retrieve for future calls.
- Can be implemented recursively, but tends to be more natural as an iterative process.

→ smaller runtime.

# DYNAMIC PROGRAM SOLUTION

## Algorithm: WEIGHTINTDP

Sort  $\sigma$  by finish time → create an array of  $(n+1)$  items.

$m[0] := 0 \rightarrow 0^{\text{th}}$  index = 0.

for  $j = 1$  to  $n$  do

    Find index  $i$

$m[j] = \max(m[j - 1], m[i] + v_j)$

end

↳ storing and looking up tables.

(recursice table look-ups,

We want: instead of recursive function calls).

## DP Solutions

- DP algorithms are formulaic.
- We understand how loops work.

(Bellman equation) NO Pseudocode.

- Definitions required for algorithm to work

- Description of matrix

- Bellman Equation

- Location of solution, order to populate the matrix

# DYNAMIC PROGRAM SOLUTION

Definitions required for algorithm to work

- $\sigma$  sorted by finish time, ascending order.
- For a given job at index  $j$ ,  $i_j < j$  is the largest index such that  $f_i \leq s_j$ .  $\rightarrow$  define what  $i$  is

$\rightarrow$  preprocessing of input

what index

Description of matrix

Dot : non-implicit params to your recursive solution.

= are we trying to calculate the max value for (j) in this case.

dimensions (depends on degrees of freedom).

- 1D array  $M$ , where  $M[j]$  is the maximum value of a compatible schedule for the first  $j$  items in sorted  $\sigma$ .  
Initialize  $M[1] = v_1$ .  $\rightarrow$  initialize base case  $\rightarrow$  fill it up.

Bellman Equation

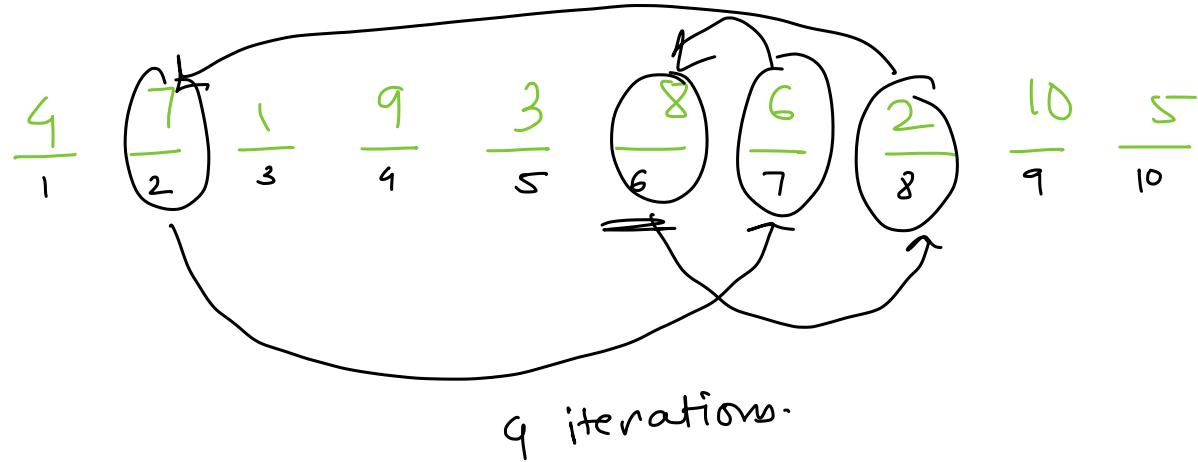
$\rightarrow i_j < 1 ; M[i,j] = 0$

- $M[j]$  =  $\max\{M[j - 1], M[i_j] + v_j\}$

Solution, order to populate

- The maximum value of a compatible schedule for the  $n$  jobs is found at  $M[n]$ .  $\rightarrow$  last cell. Populate from 2 to  $n$ .

8<sup>th</sup> element:



recursive solution :  $O(2^n)$ .

## ANALYZE THE ALGORITHM

### DP Solution

*some time*

- $\sigma$  sorted by finish time, ascending order.
- For a given job at index  $j$ ,  $i < j$  is the largest index such that  $f_i \leq s_j$ . *figure out the i's*
- Bellman Equation:  $m[j] = \max(m[j - 1], m[i] + v_j)$   
*Bellman eq.*

### Runtime

*lookups, addition, math.  
constant time.*

- Preprocessing:
  - Sorting jobs:  $O(n \log n)$ .
- Populate the matrix:
  - Number of cells:  $O(n^2)$  *n log n*
  - Cost per cell: Finding  $i$ :  $O(n)$  linear search,  $O(\log n)$  binary search *O(n<sup>2</sup>)*

Overall:  $O(n^2)$  linear search,  $O(n \log n)$  binary search

## ANALYZE THE ALGORITHM



### DP Solution

- $\sigma$  sorted by finish time, ascending order.
- For a given job at index  $j$ ,  $i < j$  is the largest index such that  $f_i \leq s_j$ .
- Bellman Equation:  $m[j] = \max(m[j - 1], m[i] + v_j)$

calculating max value = figuring out how many intervals in the solution..  
What about the schedule  $S$ ? ↗ get this using solution matrix.

Trace back from the optimal value: ↗ reverse engineer / Backtrack  
from the Bellman eq. optimal solution.

- Job  $j$  is part of the optimal schedule from 1 to  $j$  iff  $v_j + \text{OPT}(i) \geq \text{OPT}(j - 1)$

$= =$   
because you choose max in Bellman.

# BASIC DP OUTLINE

## Algorithm Template

- Preprocessing of data  $\rightarrow$  sorting
- Populate the matrix:
  - Iterate over the cells in the correct order.
  - Understand the work done per cell.

$\hookrightarrow$  storing / lookup.

## Algorithm Guidelines

- ① There are only a polynomial number of subproblems.
- ② The solution to the larger problem can be efficiently calculated from the subproblems.  $\rightarrow$  work done per cell has to be efficient.
- ③ Natural ordering of the subproblems from “smallest” to “largest”. iterate through subproblems : smallest  $\rightarrow$  largest.

[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN\\*](#)[LS\\*](#)[RNA\\*](#)

# LONGEST INCREASING SUBSEQUENCE $\equiv$

# LONGEST INCREASING SUBSEQUENCE

## Problem

- Given an integer array  $A[1..n]$ .
- Find the longest increasing subsequence. That is, let  $i$  be a sequence of indexes, we have  $A[i_k] < A[i_{k+1}]$  for all  $k$ .

## Subsequence

- For a sequence  $A$ , a subsequence  $S$  is a subset of  $A$  that maintains the same relative order.
- Ex: I like watching the puddles gather rain.
  - puddles: subsequence, substring (contiguous)
  - late train: subsequence, not substring (not contiguous)

$\hookrightarrow$  strictly less.  
 you need to have a strictly growing subarray in the array.

TH1: For an array of length  $n$ , how many subsequences?  $2^n$

$\hookrightarrow$  each element will either be in a subsequence or not.  $\rightarrow 2^n$   
 $\hookrightarrow$  2 possibilities.

DP → coming up with a recursive program → refactor to Dynamic Programming

## RECURSIVE APPROACH

dichotomy = {skip, take}

↓  
to make it more

Algorithm: LIS

will the elements be in subsequence  
or not. (max will give you longest).

Input : Integer  $k$ , and array of integers  $A[1..n]$ .

Output: Return length of LIS where every value  $> k$ .

if  $n = 0$  then return 0 } Base case.

else if  $A[1] \leq k$  then

return LIS( $k, A[2..n]$ ) } you take the first element out of the array  
and recurse from second element to n.

else  $\rightarrow A[1] > k \rightarrow A[1]$  dichotomy.

skip := LIS( $k, A[2..n]$ ) → skip  $A[1]$

take := LIS( $A[1], A[2..n]$ ) + 1 → if we use  $A[1]$ , all the other values  
have to be strictly greater than  $A[1]$ .

return max{skip, take} : dichotomy. you make  $A[1]$  to be your  $k$ .

end

as you're including  $A[1]$ , your subsequence has  $A[1]$ .

TH2: For an array  $A[1..n]$ , how would you find the length of the LIS using the LIS( $\cdot$ ) algorithm? LIS( $-\infty, A[1..n]$ )

setting  $k$  to be  
any value smaller  
than arrays elements.

to track  
the length.

# RECURSIVE APPROACH

---

## Algorithm: LIS

---

**Input :** Integer  $k$ , and array of integers  $A[1..n]$ .

**Output:** Return length of LIS where every value  $> k$ .

**if**  $n = 0$  **then return** 0

**else if**  $A[1] \leq k$  **then**

  | **return** LIS( $k, A[2..n]$ )

**else**

  |  $skip :=$ LIS( $k, A[2..n]$ )

  |  $take :=$ LIS( $A[1], A[2..n]$ ) + 1

  | **return** max{ $skip, take$ }

**end**

---



TH3: Run time of the algorithm for a length  $n$  array?  $O(\underline{2^n})$

TH4: How many distinct recursive calls for a length  $n$  array?

$O(n^2)$   $\rightarrow$  2 calls per element,  $n$  elements.  $= n^2$

# DYNAMIC PROGRAM FOR LIS

dimensions for array: parameters to your recursive code.

in this case: 2 non-implicit parameters.  
(2 parameters in the recursive call that are constantly changing).

## Description of matrix

2D array  $L$ , where  $L[i, j]$  is the maximum LIS of  $A[j..n]$  with every item  $> A[i], i < j$ .

current k value

answer to  
the problem.

rule for problem  $k < j$

## Bellman Equation

Recursive fn calls

↓  
Recursive lookups in the table.

Base case.

$$L[i, j] = \begin{cases} 0, & \text{if } j > n \\ L[i, j+1], & \text{if } A[i] \geq A[j] \\ \max\{L[i, j+1], L[j, j+1] + 1\}, & \text{otherwise} \end{cases}$$

skip
take  $A[i]$ 
-populating  $n^2$  cells in array.

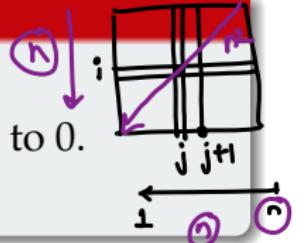
$\rightarrow K \geq \text{next element in array.}$ 
 $\rightarrow$  dichotomy.
 $\rightarrow$  const time  $O(1)$

## Solution and populating $L$

- Solution in  $L[0][1]$ ; add  $A[0] = -\infty$ .

- Populate  $j$  from  $n$  to  $1$ ;  $i$  from  $0$  to  $j-1$  or  $j-1$  to  $0$ .

- Run time:  $O(n^2)$  ↳ end of array → solution.  
↳ 2D array



# DYNAMIC PROGRAMMING FOR GAMES

## Games

- Some number of players (1 to many).
- Set of rules with some objective.
- Huge domain, started by Von Neumann, that spans many fields such as Economics, Math, Biology, and Computer Science.

game theory.

→ how to win?

## DP for Games

In many games, DP is a natural paradigm for an optimal strategy.

# COINS IN A LINE

## Players

Two players:



Alice  
(Player A)



Bob  
(Player B)

tie: neither one gets the value.

either  
↓                      or  
      0 0 0 0 . . . . 0

## Rules

assume n is always even.

- $n$  (even) coins in a line; each coin has a value.
- Starting with Alice, each player will pick a coin from the head or the tail.
- Winner: Player with the max value at the end; winning player keeps the coins.

# NATURAL DICHOTOMY



for Alice.

## Head or Tail?

- Two players: Assume that Bob will play optimally.
  - For Alice's kth turn:
    - Coin array:  $C[i..j]$
    - $\text{AliceOpt}(c[i..j]) := \max\{c[i] + \text{BobOpt}(c[i+1..j]), c[j] + \text{BobOpt}(c[i..j-1])\}$
    - $\text{BobOpt}(c[i..j]) := \min\{\text{Head}, \text{tail}\}$
- rational player.  
you need to express BobOpt in terms of AliceOpt.

Bob can maximize his profit by minimizing Alice's profit.

TH1: How many dimensions for DP array? 2

→ two parameters. : 2D array.



$i$  could be from  $1 \rightarrow n$   
 $(n)^2$  →  $j$  could be from  $1 \rightarrow n$

we are concerned about  $\frac{n^2}{2}$  side of array.

# HEAD OR TAIL DP

## DP Description

- 2D array  $M$ : each cell represents Alice's optimal payout.
- $M[i, j]$  is the maximum value possible for Alice when choosing from  $c[i..j]$ , assuming Bob plays optimally.

- Bellman Equation:

$$M[i, j] = \max\{c[i] + \min\{M[i+2, j], M[i+1, j-1]\}, c[j] + \min\{M[i+1, j-1], M[i, j-2]\}\}$$

$M[i, i] \stackrel{\textcircled{1}}{=} c[i]$  for all  $i$ .  $\rightarrow$  diagonal.  $\stackrel{\textcircled{2}}{=}$  take coin with larger val.  $\rightarrow$  tail.

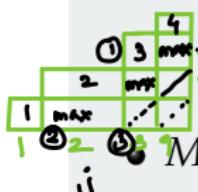
- $M[i, j] \stackrel{\textcircled{3}}{=} \max\{c[i], c[j]\}$  for  $i = j - 1$ . only two coins left.

- Populate  $i$  from  $n - 2$  to  $1$ ;  $j$  from  $n$  to  $3$  for  $i < j - 1$ .

- Solution:  $M[1, n]$  Alice's payout.

- Runtime:  $O(n^2)$  populating 2D array.

- Proof of correctness: Strong induction on the cell population order.



Bobopt(c[i+1, ..., j])

head

tail

Bobopt(c[i, ..., j-1])

tail.

→ use Bellman  
equation.

inductive hyp:

recursive lookups

return the correct value.

[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN\\*](#)[LS\\*](#)[RNA\\*](#)

# MAX SUBARRAY

---

# MAX SUBARRAY

## Problem

Given an array  $A$  of integers, find the (non-empty) contiguous subarray of  $A$  of maximum sum.  $\rightarrow$  integers could be -ve.

## Exercise – Teams of 3 or so

- Solve the problem in  $\Theta(n^2)$ .
- Solve the problem in  $O(n \log n)$ .
- Prove correctness and complexity.

*divide and conquer*

## PART 2: GIVE AN $O(n \log n)$ SOLUTION.

---

### Algorithm: MAXSUBARRAY

---

**Input :** Array  $A$  of  $n$  ints.

**Output:** Max subarray in  $A$ .

**if**  $|A| = 1$  **then return**  $A[1]$

$A_1 := \text{MAXSUBARRAY}(\text{Front-half of } A)$

$A_2 := \text{MAXSUBARRAY}(\text{Back-half of } A)$

$M := \text{MIDMAXSUBARRAY}(A)$

**return** *Array with max sum of  $\{A_1, A_2, M\}$*

---

### Algorithm: MIDMAXSUBARRAY

---

**Input :** Array  $A$  of  $n$  ints.

**Output:** Max subarray that crosses midpoint  $A$ .

$m := \text{mid-point of } A$

$L := \text{max subarray in } A[i, m - 1] \text{ for } i = m - 1 \rightarrow 1$

$R := \text{max subarray in } A[m, j] \text{ for } j = m \rightarrow n$

**return**  $L \cup R //$  subarray formed by combining  $L$  and  $R$ .

---

## PART 2: GIVE AN $O(n \log n)$ SOLUTION.

---

### Algorithm: MAXSUBARRAY

---

**Input :** Array  $A$  of  $n$  ints.

**Output:** Max subarray in  $A$ .

**if**  $|A| = 1$  **then return**  $A[1]$

$A_1 := \text{MAXSUBARRAY}(\text{Front-half of } A)$

$A_2 := \text{MAXSUBARRAY}(\text{Back-half of } A)$

$M := \text{MIDMAXSUBARRAY}(A)$

**return** *Array with max sum of  $\{A_1, A_2, M\}$*

---

### Analysis

- Correctness: By induction,  $A_1$  and  $A_2$  are max for subarray and  $M$  is max mid-crossing array.
- Complexity: Same recurrence as MERGESORT.

process the list once (include it, not include it).  
 & then you start a new subarray.

## PART 3: GIVE AN $O(n)$ SOLUTION.

### DP Solution

- 1D array  $s$ , where  $s[i]$  contains the value of the max subarray ending at  $i$ . ( $O(n)$  cells)  
 (populate the matrix = linear.)
- Bellman equation:  $s[i] = \max(\underbrace{s[i-1] + A[i]}_{\text{extend by 1}}, \underbrace{A[i]}_{\text{start new one}})$ . ( $O(1)$  time)
- Solutions is:  $\max_j\{s[j]\}$ . ( $O(n)$  time)  
 cell that has the max value (post processing).

$$O(2n) = O(n)$$

But we need the subarray not the value!

- Use a parallel array that memoizes the starting index of the subarray ending at  $i$ :

$$\text{start}[i] = \begin{cases} \text{start}[i-1] & \text{if } s[i-1] + a[i] > a[i] \\ i & \text{otherwise} \end{cases}$$

- Or, trace back from max value at index  $j$  until  $s[i] = A[i]$ .

[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN\\*](#)[LS\\*](#)[RNA\\*](#)

# SUBSET AND KNAAPSACK

---

# SUBSET PROBLEM

↳ no. in a set.

goal: find a subset of those no. that add up to some other number.

## Problem Definition ↳ scheduling problem.

- A single machine that we can use for time W.
- A set of jobs: 1, 2, ..., n.
- Each job has a run time: w<sub>1</sub>, w<sub>2</sub>, ..., w<sub>n</sub>.
- What is the subset S of jobs to run that maximizes  
 $\sum_{i \in S} w_i \leq W$ ?      adding individual job runtime to match w.

## Greedy Heuristics

↳ largest to smallest weight arrangement.

- Decreasing weights: {W/2 + 1, W/2, W/2}
- Increasing weights: {1, W/2, W/2}

(set is not ordered)

## DYNAMIC PROGRAMMING APPROACH

dichotomy (exclude, not exclude).

going through list of items to figure out whether or not to exclude an item  
1D Approach

last item not in solution.

- if  $n \notin S$ , then  $v[n] = v[n - 1]$  } exclude last, recursive call on  $(n-1)$  items.
- if  $n \in S$ , then  $v[n] = ?$  } include last item:
  - Accepting  $n$  does automatically exclude other items.

① consider no. of items  
② capacity we have left over.

2D solution.

Need to consider more

To solve  $v[n]$ , we need to consider: same dichotomy with the constraint of how much processing time we have.

$n \notin S$

- the best solution with  $n - 1$  previous items restricted by  $W$ , and if we look at  $(n-1)$  items, how much  $w$  can we use.

$n \in S$

- the best solution with  $n - 1$  previous items restricted by  $W - w_n$  we minus the processing time of the  $n^{\text{th}}$  job.

# DYNAMIC PROGRAMMING APPROACH

## 2D Approach

- 2D Matrix  $v$ :

- $i$ : Item indices from 0 to  $n$ .
  - $w$ : Max weight from 0 to  $W$ .
  - $v[i, w]$  is the subset of the first  $i$  items of maximum sum  $\leq w$ .
- processing time*  
we are allowed to use  
items not sorted
- max sum doesn't go past the capacity constraint.

- Indicator:  $x_{i,w} := 0$  if  $w_i > w$  and 1 otherwise.
- Bellman Equation:

$$v[i, w] = \max(v[i - 1, w], x_{i,w} \cdot (v[i - 1, w - w_i] + w_i))$$

*Indicator*

- Base case.*
- $v[0, w] := 0$  for all  $w$  and  $v[i, 0] := 0$  for all  $i$
  - Solution value:  $v[n, W]$ .

TH7: Running time to populate the matrix:  $O(nW)$

TH8: Is this polynomial? No, *pseudo-polynomial* because of  $W$  which is unbounded.

unary encoding

$V : \mathbb{N}$

5 ones

↳ on  $W$  will make it polynomial.

max sum doesn't go past the capacity constraint.

last item's weight is more or less than the processing time available.

↳ then set indicator variable to 0.

*exclude i*      *include i*

include weight of i

$W = 2^n$   
 $\hookrightarrow n$  bits to encode.

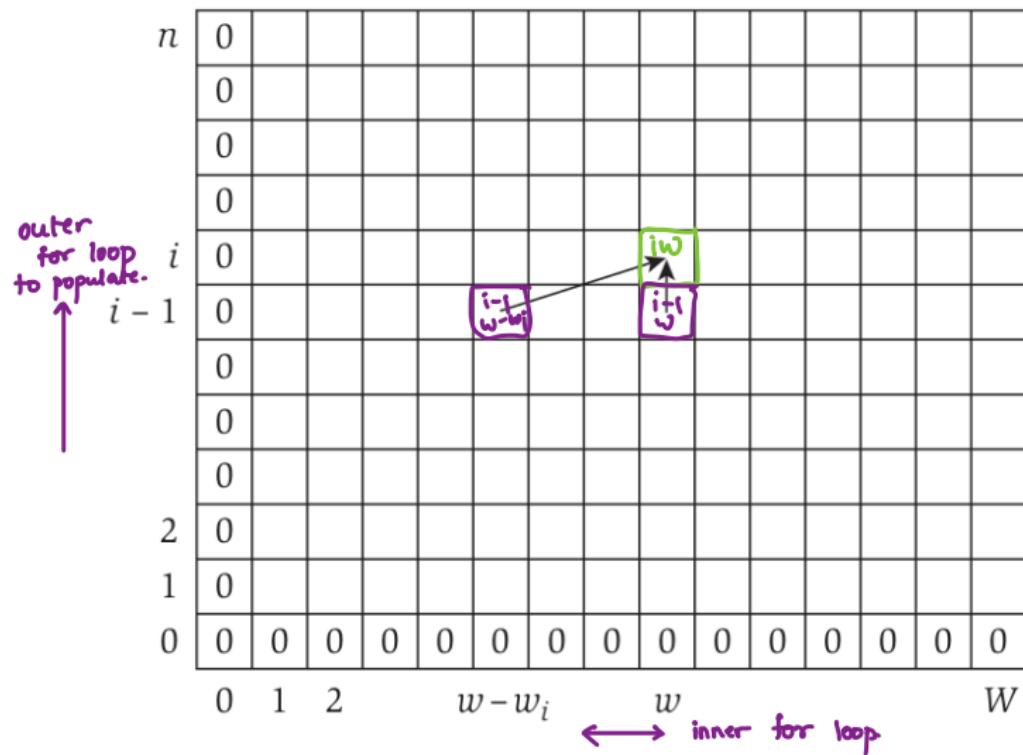
↳ subset of the first  $n$  items

no limit to how much  $w$  will be.

$\log_2(w)$

# SUBSET VISUALIZATION

Matrix Visualization:



# SUBSET VISUALIZATION

we need the previous row filled in to calculate the next row.

Example Run:

$$w_i = w_1 = 2 .$$

$$W = \underline{6}, \text{ items } w_1 = \underline{2}, w_2 = \underline{2}, w_3 = \underline{3}$$

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Initial values

3							
2							
1							
i = ①	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

$$\underline{6} = W$$

Filling in values for  $i = 1$

$w_i = w_2 = 2$							
$w_1 + w_2$							
i = ②	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for  $i = 2$

$w_i = w_3 = 3$							
$w_1 + w_3 = 2+3=5$							
$w_2 + w_3 = 5$							
i = ③	0	0	2	3	4	5	5
2	0	0	2	4	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for  $i = 3$

$w_3$  is in the solution  
 $w_2$  is not in the solution.  
 $w_1$  is part of the solution

you have a solution matrix for the subset prob, what sets will be a part of the solution?  
**DYNAMIC PROGRAMMING APPROACH**

↳ you backtrack the Bellman equation.

## 2D Approach

- 2D Matrix  $v$ :
  - $i$ : Item indices from 0 to  $n$ .
  - $w$ : Max weight from 0 to  $W$ .
  - $v[i, w]$  is the subset of the first  $i$  items of maximum sum  $\leq w$ .
- Indicator:  $x_{i,w} := 0$  if  $w_i > w$  and 1 otherwise.
- Bellman Equation:

$$v[i, w] = \max(v[i - 1, w], x_{i,w} \cdot (v[i - 1, w - w_i] + w_i))$$

- $v[0, w] := 0$  for all  $w$  and  $v[i, 0] := 0$  for all  $i$
- Solution value:  $v[n, W]$ .

How can we recover the subset itself?

TH9: Running time of recovery of subset:  $O(n)$

## KNAPSACK EXTENSION

→ generalized version  
of subset problem.

when weight = value,  
that will be the  
subset problem.



### Problem Definition

- You are a thief with a knapsack that can carry W weight of goods.
- A set of items: 1, 2, ..., n. → from which you can steal.
- Each item has a weight: w<sub>1</sub>, w<sub>2</sub>, ..., w<sub>n</sub>. → minimize the weight. ↳ weight can't exceed knapsack weight
- Each item has a value: v<sub>1</sub>, v<sub>2</sub>, ..., v<sub>n</sub>. → maximize the value
- What is the subset S of items to steal that maximizes  $\sum_{i \in S} v_i$  with the constraint that  $\sum_{i \in S} w_i \leq W$ ?

# EXERCISE: SOLVE THIS WITH DP IN $O(nW)$ .

## DP Solution

- 2D Matrix:
  - $i$ : Item indices from 0 to  $n$ .
  - $w$ : Max weight from 0 to  $W$ .
  - $v[i, w]$  is the subset of the first  $i$  items of maximum total value with a sum of weights  $\leq w$ .
- Indicator:  $x_{i,w} := 0$  if  $w_i > w$  and 1 otherwise.
- Bellman Equation:

$$v[i, w] = \max(v[i - 1, w], x_{i,w} \cdot (v[i - 1, \underbrace{w - w_i}_{\text{ensures capacity constraint.}}] + v_i))$$

- $v[0, w] := 0$  for all  $w$  and  $v[i, 0] := 0$  for all  $i$
- Solution value:  $v[n, W]$ .

[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN\\*](#)[LS\\*](#)[RNA\\*](#)

# EDIT DISTANCE

---

# EDIT DISTANCE

↳ how many changes does it take to go from one string to another string.  
**Problem**

Minimum number of letter

- insertions: adding a letter,
- deletions: removing a letter,
- substitutions: replacing a letter

} use these operations to  
go from  $a \rightarrow b$

to change string  $A[1..m]$  to string  $B[1..n]$ .

$m$  characters

$n$  characters.

Ex: TUESDAY  $\rightarrow$  THUESDAY  $\rightarrow$  THURSDAY

add H.

replace R

} 2 edit distances away.

Or, align and count mismatched letters

TUESDAY  
THURSDAY

some add

# RECURSIVE APPROACH

-Deletion:  $\text{Edit}(i, j) = \text{Edit}(i-1, j) + 1$

one edit made, you keep track of the edits.  
 removed one character in A so,  $(i-1)$  characters left

→ trichotomy - take minimum.

## Smaller Subproblems

$A \rightarrow B$

- Let  $A[1..m]$  and  $B[1..n]$  be the 2 input strings.
- What is the edit distance for  $\underline{A[1..i]}$  and  $\underline{B[1..j]}$ :

- i, j represent how far you are in the string.
- Insertion:  $\text{Edit}(i, j) = \text{Edit}(i, j-1) + 1$ . → insert a character in A at corresponds to the jth index at B.
  - Deletion:  $\text{Edit}(i, j) = \text{Edit}(i-1, j) + 1$ .
  - Substitution:  $\text{Edit}(i, j) = \text{Edit}(i-1, j-1) + A[i] \neq B[j]$
  - $i = 0$ :  $\text{Edit}(i, j) = j$ . → j insertion i<sup>th</sup> and j<sup>th</sup> you edit only if characters match they don't match.
  - $j = 0$ :  $\text{Edit}(i, j) = i$ .

substitution: if  $A[i] \neq B[j]$  make edit (+1)  
 if  $A[i] = B[j]$ , dont make edit (+0)

↳ i deletion so you take that to match A with B if they match, then dont do anything.

# DYNAMIC PROGRAM FOR EDIT DISTANCE

Description of matrix  $\rightarrow$  edits in A, edits in B  
 $i, j \rightarrow 2D$  matrix

2D array  $E$ , where  $E[i, j]$  is the edit distance for  $A[1..i]$  and  $B[1..j]$ .

## Bellman Equation

$$E[i, j] = \begin{cases} i, & \text{if } j = 0 \\ j, & \text{if } i = 0 \\ \min\{\underbrace{E[i, j - 1] + 1}_{\text{insertion}}, \underbrace{E[i - 1, j] + 1}_{\text{deletion}}, \underbrace{E[i - 1, j - 1] + A[i] \neq B[j]}_{\text{substitution}}\}, & \text{otherwise} \end{cases}$$

2 Base cases

we are filling  $m \times n$  cells in constant time each. so,  $O(mn)$  runtime.

indication

## Solution and populating L

- Solution in  $E[m, n]$  *fill in all the base cases before populating.*
- Set  $E[0, j] = j$ ;  $E[i, 0] = i$ ; populate from 1 to  $n$ , 1 to  $m$ .
- Run time:  $O(mn)$

## SPACE SAVINGS

↳ when you have space constraints. *you don't have entire solution matrix in memory.*

Bellman Equation  $\Rightarrow$  for edit distance

$$E[i, j] = \begin{cases} i, & \text{if } j = 0 \\ j, & \text{if } i = 0 \\ \min\{E[\underline{i}, \underline{j - 1}] + 1, E[\underline{i - 1}, \underline{j}] + 1, \\ & E[\underline{i - 1}, \underline{j - 1}] + A[i] \neq B[j]\}, & \text{otherwise} \end{cases}$$

*i, i-1, j, j-1*  
*you only need the previous row to calculate the next row.*

## How much space do we need?

- Notice that  $E[i][j]$  depends on  $E[i, j - 1]$ ,  $E[i - 1, j]$ , and  $E[\underline{i - 1}, \underline{j - 1}]$ .
- We only need previous and current row of matrix for calculations.

# SHORTEST PATH

↳ dynamic programming  
for shortest path.

→ find shortest path from  $s$  to all the other nodes in the graph.

## SHORTEST PATH

GOING NEGATIVE

→ weight of the edge can be -ve.

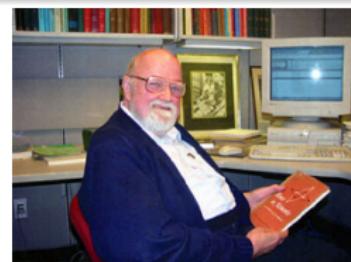
**Problem Definition** but no -ve weighted cycles in the graph.

We have a directed graph  $G = (V, E)$ , where  $|V| = n$  and  $|E| = m$  and a node  $s$  that has a path to every other node in  $V$ . For each edge  $e = (i, j)$ ,  $c_{ij}$  is the weight of the edge, and there are no cycles with negative weight.

- What is the shortest path from  $s$  to each other node?



Richard Bellman



L R Ford Jr.

# SHORTEST PATH

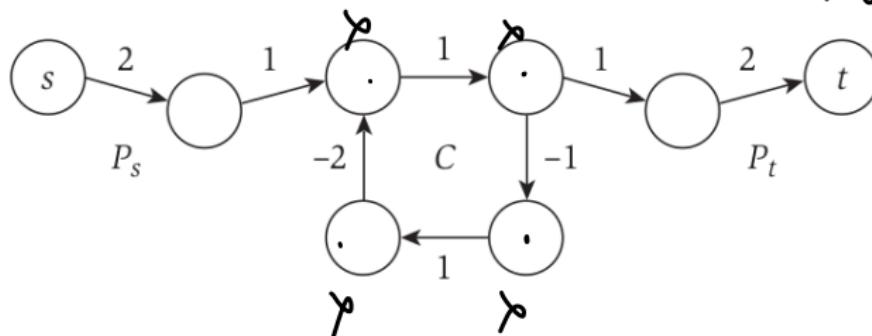
GOING NEGATIVE

## Problem Definition

We have a directed graph  $G = (V, E)$ , where  $|V| = n$  and  $|E| = m$  and a node  $s$  that has a path to every other node in  $V$ . For each edge  $e = (i, j)$ ,  $c_{ij}$  is the weight of the edge, and there are no cycles with negative weight.

- What is the shortest path from  $s$  to each other node?

Why no negative cycles? *if you have a -ve weighted cycle, then the shortest path =  $-\infty$ . (you keep going around that cycle to reduce the distance).*



→ why no -ve edge weights: for a minimizer, we can argue that  
**DIJKSTRA'S** there is no other path outside the nodes we saw that can have a shorter path.

### Algorithm: Dijkstra's

Let  $S$  be the set of explored nodes.

For each  $u \in S$ , we store a distance value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

Choose  $v \notin S$  with at least one incoming edge originating from a node in  $S$  with the smallest

$$d'(v) = \min_{e=(u,v): u \in S} d(u) + \ell_e$$

Append  $v$  to  $S$  and define  $d(v) = d'(v)$ .

**end**

**return**  $S$



# DIJKSTRA'S

if you boost every edge by a value, to make it all -ve, then compare, the length of the path will be a function of the no of hops. the more hops, the higher the length.

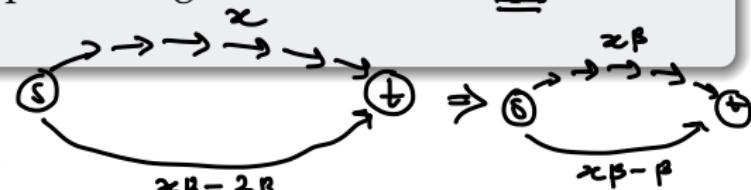
## Negative Problem

- Lose guarantee that minimum edge between  $S$  and  $V \setminus S$  is part of minimum path.

Why not just boost all edges by max negative value plus a bit ( $\beta$ )?

- A path with  $\underline{x}$  edges: Cost increases  $x \cdot \underline{\beta}$ . *boosting  $\underline{x}$  by a value.*
- Solution in new graph is not guaranteed to be optimal in original graph.

upper graph is  
shortest path,  
after boosting doesn't remain  
the shortest path.



## BELLMAN-FORD

② shortest path will have  $(n-1)$  edges.  
as there are  $n$  vertices.

Observation 1 → there will be a tree cost going around those cycles  $\rightarrow$  not shortest.

If  $G$  has no negative cycles, then there exists a shortest path from  $s$  to  $t$  that is simple, and has at most  $n^{(2)} - 1$  edges.

↳ not going through cycles. ↳ one upper limit to no. of edges.

Dynamic Program  $\frac{\text{no. of edges}}{(n-1) \text{ or } m.}$   $\frac{\text{vertices } (n)}$

- 2D matrix  $M$  of # edges in path  $\times$  vertices.

finds shortest paths from every node in the graph to some node  $t$ .

- $M[i][v]$  is the shortest path from  $v$  to  $t$  using  $\leq i$  edges.
- Solution:  $M[n-1][s]$
- Dichotomy:
  - Use  $\leq i-1$  edges.
  - Use  $\leq i$  edges.
 ↳ shortest path weight from  $s \rightarrow t$ .  
 use the edge or dont use the edge.  
 $\leq (i-1)$  edges.  $\leq i$  edges. should we or add another edge to our shortest path.

$$M[i][v] = \min\left\{ M[i-1][v], \min_{w \in V} \{ M[i-1][w] + c_{vw} \} \right\},$$

where  $c_{vw} = \infty$  if no edge from  $v$  to  $w$ .

# BELLMAN-FORD ANALYSIS

$$M[i][v] = \min\{M[i-1][v], \min_{w \in V}\{M[i-1][w] + c_{vw}\}\}$$

Worst Case:  $n$  nodes

- # of Cells:  $O(n^2)$ .  $\xrightarrow{\text{no. of edges in a path} = (i-1)}$  }  $n^2$
- Cost per cell:  $O(\underline{n})$ .
- Overall:  $\underbrace{n^2 \times n}_{n^2 \times \underline{n}} = \underline{n^3}$

Worst Case:  $n$  nodes,  $m$  edges

- For each node  $v$ , we only need to consider outgoing edges to  $w$  (denoted by  $\eta_v$ ).  $\xrightarrow{\text{outgoing edges}}$
- For every node  $v$ , we need to do this calculation for  $0 \leq i \leq n - 1$  lengths.  $\xrightarrow{\text{n}^2\text{-ish}}$
- Overall:  $O(n \sum_{v \in V} \eta_v) = O(\underline{mn})$ .

# BELLMAN-FORD ANALYSIS

→ space saving version of shortest path

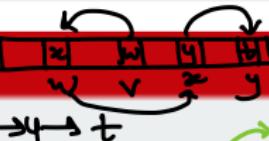
$$M[i][v] = \min\{M[i-1][v], \min_{w \in V}\{M[i-1][w] + c_{vw}\}\}$$

Worst Case:  $n$  nodes,  $m$  edges

- Overall:  $O(n \sum_{v \in V} \eta_v) = O(mn)$ .

Space Saving:  $O(n)$ .

first:



Keep updating first[v] as you go through the Bellman eq.

- To build row  $i$ :  $v \rightarrow w \rightarrow x \rightarrow y \rightarrow t$
- you could back track, but its more complex. • We only need  $i - 1$  values for each node.  $M[v] = \min\{M[v], \min_{w \in V}\{M[w] + c_{vw}\}\}$  for each  $i$ .
- Recovery of actual path: An additional array first[v] that maintains the first hop from  $v$  to  $t$ .

keep track of the very first hop from node  $v$  along its shortest path to  $t$ .

# NEGATIVE CYCLES

Observation 2 → used to figure out if you have -ve cycles.

If there is a negative cycle along the path from  $s$  to  $t$ , then the shortest path is  $\underline{\underline{-\infty}}$ .

Observation 3

→ keep running Bellman Ford for paths of length  $n$ .

$M[i][v] = M[n-1][v]$  for all  $i > n-1$  and all nodes  $v$  if there are no negative cycles on the paths to  $t$ .

if paths beyond  $(n-1)$  get shorter, you know there is a -ve cycle.

paths won't change.

if you have no negative cycles.

Augmented Graph for Negative Cycle Finding

- Add a node  $t$  with an incoming edge from all other nodes with cost 0.
- Run Bellman-Ford from any node  $s$  to  $t$  until number of edges  $n$ .  
 $M[n][v] < M[n-1][v]$  → -ve cycle found.
- If, for some  $v$ ,  $M[n][v] \neq M[n-1][v]$ , then there is a negative cycle.

# CS 577 - Network Flow

Marc Renault

Department of Computer Sciences  
University of Wisconsin – Madison

Fall 2023

TopHat Section 001 Join Code: 477366  
TopHat Section 002 Join Code: 560750



**WISCONSIN**  
UNIVERSITY OF WISCONSIN-MADISON

# NETWORK FLOW

↙  
flow network  
and its properties.  
(graph theory)

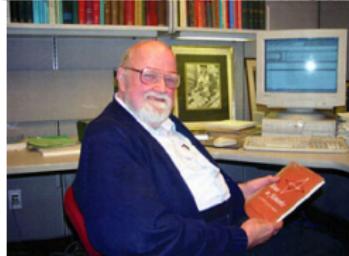
→ solve a lot of  
problems / application  
of graph theory.

# NETWORK FLOW

## Flow Problems

- Flow Network / Transportation Networks: Connected directed graph with water flowing / traffic moving through it.
  - Edges have limited capacity.
  - Nodes act as switches directing the flow.
  - Many, many problems can be cast as flow problems.
- (stuff moving through the graph)

### Ford-Fulkerson Method (1956)

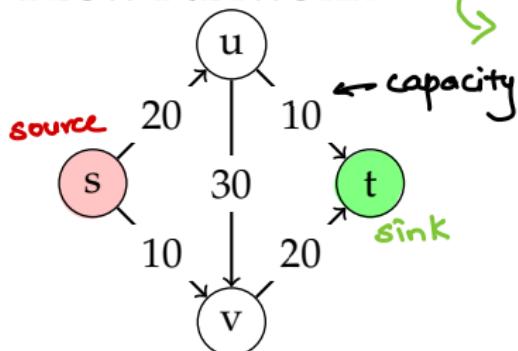


L R Ford Jr.



D. R. Fulkerson

# FLOW NETWORK



*flow along the edge vs capacity*  
*current flow* ↗ *max flow* ↘

## Basic Flow Network

- Directed graph  $G = (V, E)$ .
- Each edge  $e$  has  $c_e \geq 0$ .
- Source  $s \in V$  and sink  $t \in V$ .
- Internal node  $\underline{V \setminus \{s, t\}}$ .

## Defining Flow

*any node that is  
not the source or sink*

- Flow starts at  $\underline{s}$  and exits at  $\underline{t}$ .
- Flow function:  $f : E \rightarrow R^+$ ;  $f(e)$  is the flow across edge  $e$ .

*what you have to find.*

*function* *from edges to some real value.*

*Flow Conditions:*

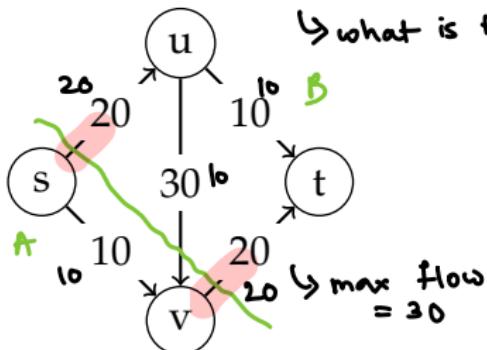
- i Capacity: For each  $e \in E$ ,  $0 \leq f(e) \leq c_e$ .
- ii Conservation: For each  $v \in V \setminus \{s, t\}$ ,

*for every internal node.* 
$$\sum_{e \text{ into } v} f(e) = f^{\text{in}}(v) = f^{\text{out}}(v) = \sum_{e \text{ out of } v} f(e)$$

- Flow value  $v(f) = f^{\text{out}}(s) = f^{\text{in}}(t)$ .

*flow should be at least 0 or at most capacity.*

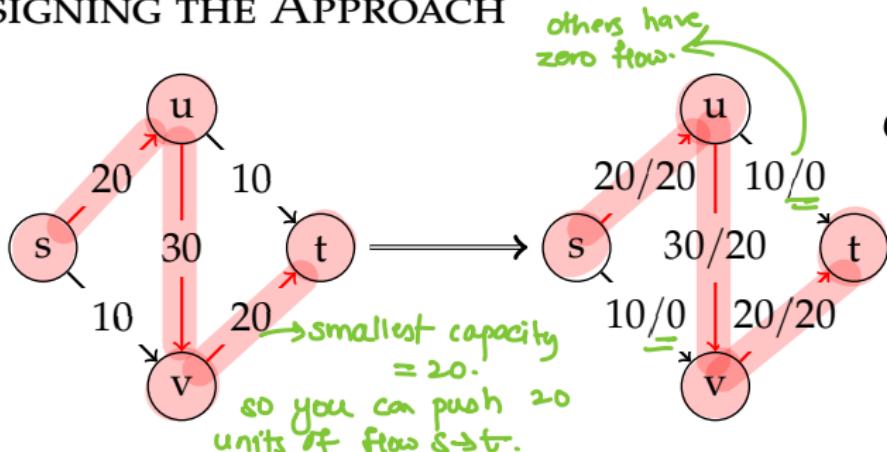
# MAXIMUM-FLOW PROBLEM



## Alternate View: Min-Cut

- A Cut: Partition of  $V$  into sets  $(\underline{A}, \underline{B})$  with  $s \in \underline{A}$  and  $t \in \underline{B}$ .
- Flow from  $s$  to  $t$  must cross the set  $A$  to  $B$ .
- Cut capacity:  $c(A, B) = \sum_{e \text{ out of } A} c_e$  } sum of outgoing edges from  $A$
- Minimum-cut of  $G$ : The cut  $(A^*, B^*)$  that minimizes  $c(A^*, B^*)$  for  $G$ .  
↳ smallest capacity cut.
- The min-cut and max-flow are the same value for any flow network.

## DESIGNING THE APPROACH



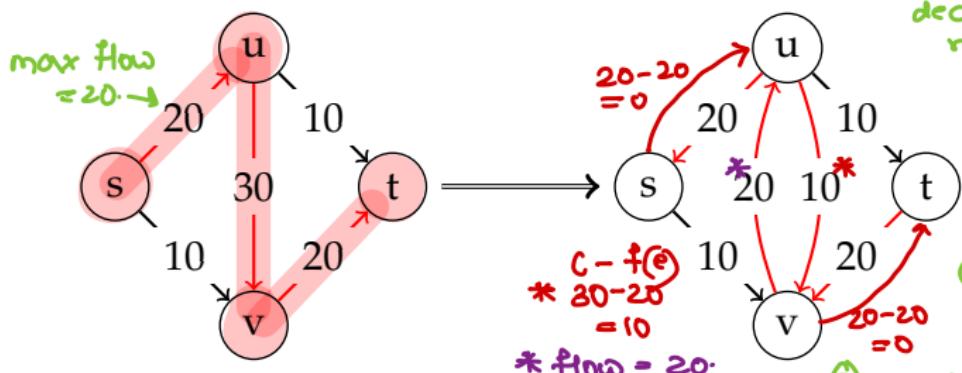
- ① start off with your flow fixn = 0 for all edges.
- ② grab some path  $s \rightarrow t$  and see which is the one with smallest capacity.
- ③ push flow equal to the smallest capacity  $s \rightarrow t$ .

### Basic Greedy Approach

- Initialize  $f(e) = 0$  for all edges.
- While there is a path from  $s$  to  $t$  with available capacity, push flow equal to the minimum available capacity along path.
- We need a mechanism to reverse flow...

function on the org flow network and the current flow fn.

## RESIDUAL GRAPH



↳ you make your greedy decision on the residual graph.

- ① everytime you make a greedy decision, you get a flow fn.
  - ② flow fn gives us a new residual graph

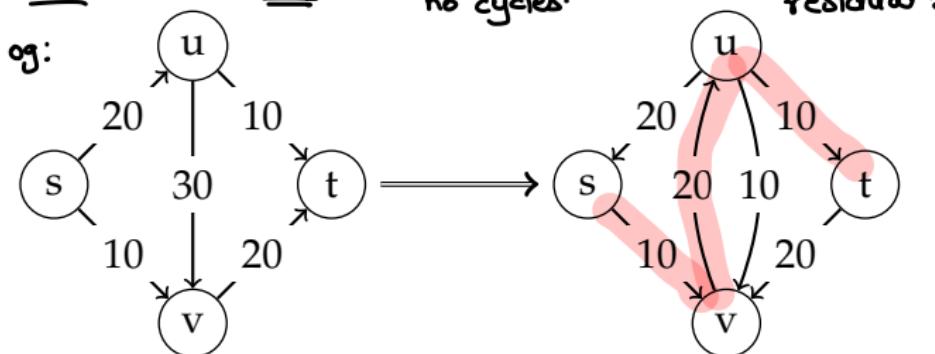
③ you find paths from set  
in this new graph.

## Residual Graph

Given a flow network  $G$  and a flow  $f$  on  $G$ , we define the residual graph  $\underline{G}_f$ :

- Same nodes as  $\underline{G}$ .  $\rightarrow$  for every edge in  $\underline{G}$ .
  - For edge  $(\underline{u}, \underline{v})$  in  $E$ :  $\rightarrow$   $\underline{\text{eg capacity}} - (\text{current flow})$ .
    - $\rightarrow$  og direction of  $e$   $\rightarrow$  Add edge  $(\underline{u}, \underline{v})$  with capacity  $c_e - f(e)$ . } residual graph has two edges.
    - $\rightarrow$  reverse edge.  $\rightarrow$  Add edge  $(\underline{v}, \underline{u})$  with capacity  $f(e)$ .  $\rightarrow$  flow going across the edge.

path from  $s \rightarrow t$  that has room for more flow  
**AUGMENTING PATH** (*simple path  $s \rightarrow t$   
no cycles*)



## Augmenting Path

- A simple directed path from  $s$  to  $t$ .
- BOTTLENECK( $P, G_f$ ): Minimum residual capacity on augmenting path  $P$ .

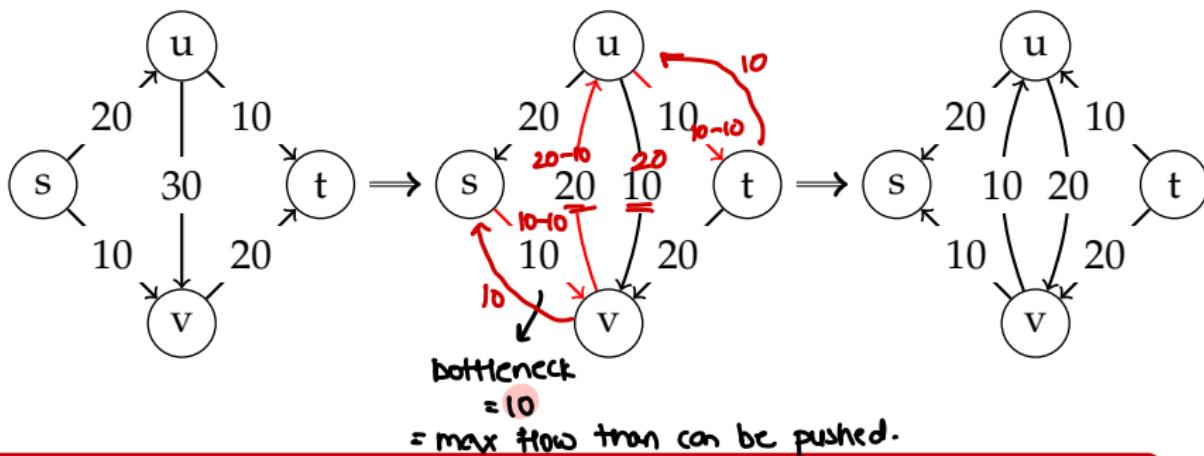
## TopHat 3

List the nodes (separated by commas, i.e.  $s,u,t$ ) of an augmenting path in the example residual graph.

# AUGMENTING PATH

*new residual graph:*

*final graph:*

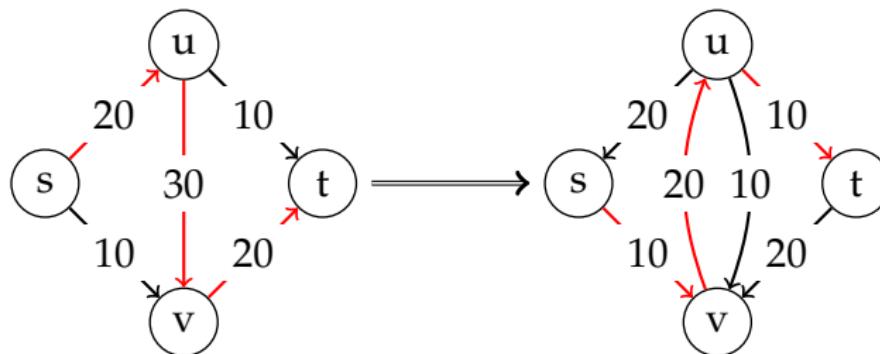


## Increasing the Flow along Augmenting Path

- Push  $\text{BOTTLENECK}(P, G_f) = q$  along path  $P$ :
  - Pushing  $q$  along a directed edge in  $G$ , increase flow by  $q$ .
  - Pushing  $q$  in opposite direction of edge in  $G$ , decreases flow by  $q$ .

# DESIGNING THE APPROACH

How to choose the augmenting path.



## Refined Greedy Approach

- Initialize  $f(e) = 0$  for all edges.
  - While  $G_f$  contains an augmenting path  $P$ :
    - Update flow  $f$  by  $\text{BOTTLENECK}(P, G_f)$  along  $P$ . *↑ from the og graph with the flow fn.*
- ↳ Keep doing this till you find an augmenting path.*

# ANALYZING THE ALGORITHM

## CONSTANT INCREASE AND TERMINATION

### Observation 1

If all capacities are integers, then all  $\underline{f}(e)$ , residual capacities, and  $v(f)$  are integers at every iteration.

### Refined Greedy Approach

- Initialize  $f(e) = 0$  for all edges.
- While  $G_f$  contains an augmenting path  $P$ :
  - Update flow  $f$  by  $\text{BOTTLENECK}(P, G_f)$  along  $P$ .

### TopHat 4

What technique should we use to prove the observation?  
induction

# ANALYZING THE ALGORITHM

CONSTANT INCREASE AND TERMINATION

↳ constant increase.

## Observation 1

If all capacities are integers, then all  $f(e)$ , residual capacities, and  $v(f)$  are integers at every iteration.

→ Augmenting path has to start at source and end at sink, plus they have some residual capacity, so at least one unit of flow can be pushed, but we push the bottleneck flow, so our flow goes up by the bottleneck val.

## Refined Greedy Approach

- Initialize  $f(e) = 0$  for all edges.
- While  $G_f$  contains an augmenting path  $P$ :
  - Update flow  $f$  by  $\text{BOTTLENECK}(P, G_f)$  along  $P$ .

Lemma 1 – claim. (proving an invariant.)

↳ whenever we increase the flow  $f_m$  by the bottleneck value, then our  $v(f')$   $>$   $v(f)$ , where  $v(f') = v(f) + \text{BOTTLENECK}(P, G_f)$  for an augmenting path  $P$  in  $G_f$ .  
 its more than the previous one strictly increasing.

## Proof.

By definition of  $P$ , first edge of  $p$  is an out edge from  $s$  that we increase by  $\text{BOTTLENECK}(P, G_f) = q$ . By the law of conservation, this will give  $q$  more flow. □

# ANALYZING THE ALGORITHM

CONSTANT INCREASE AND TERMINATION  
 termination.

## Observation 1

If all capacities are integers, then all  $f(e)$ , residual capacities, and  $v(f)$  are integers at every iteration.



$\uparrow C = \sum_e \text{capacities along outgoing edges}$

## Refined Greedy Approach

- Initialize  $f(e) = 0$  for all edges.
- While  $G_f$  contains an augmenting path  $P$ :
  - Update flow  $f$  by  $\text{BOTTLENECK}(P, G_f)$  along  $P$ .

## Theorem 2

Ford-Fulkerson

Let  $C = \sum_{e \text{ out of } s} c_e$ , the FF method terminates in at most  $C$  iterations.

Proof.  $\rightarrow$  with every iteration, the flow will strictly increase, so its using up at least one of the  $C$  units leaving  $s$ . So, after  $C$  iterations, we get termination.

From Lemma 1, the flow strictly increases at each iteration.

Hence, the residual capacity out of  $s$  decreases by at least 1 at each iteration.



# ANALYZING THE ALGORITHM

## RUNTIME

Observation 2 ( $m \geq n/2$ )

Since  $G$  is connected,  
 $m \geq \underline{n - 1}$ . Hence,  
 $O(\underline{m + n}) = O(\underline{m})$ .

*you basically need  
to bound this*

### Theorem 3

Suppose all capacities are integers. Then, runtime of  $O(\underline{mC})$ .

*no. of edges      these many iterations to push max flow*

### TopHat 7

Is this a polynomial bound? No, it is pseudo-polynomial.

*you dont know the bound of C. (its unbounded).*

# ANALYZING THE ALGORITHM

## RUNTIME

### Observation 2

Since  $G$  is connected,  
 $m \geq n - 1$ . Hence,  
 $O(m + n) = O(m)$ .

### Refined Greedy Approach

- Initialize  $f(e) = 0$  for all edges.
- While  $G_f$  contains an augmenting path  $P$ :
  - Update flow  $f$  by  $\text{BOTTLENECK}(P, G_f)$  along  $P$ .

### Theorem 3

Suppose all capacities are integers. Then, runtime of  $O(mC)$ .

### Proof.

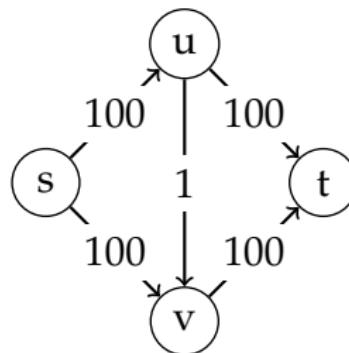
*→ go through while loop at most C times.*

- Theorem 2: termination happens in at most  $C$  iterations.
- Work per iteration: Overall:  $O(m)$ 
  - ① Find an augmenting path: BFS or DFS:  $O(m + n)$  .
  - ② Update flow along path  $P$ :  $O(n)$ . *→ traversal through graph.*
  - ③ Build new  $G_f$ :  $O(m)$ . *→ edmonds Karp algo :  $O(m^2n)$*

*steps ②, ③ same time. □*

choose paths that have large bottlenecks.

## CHOOSING GOOD AUGMENTING PATHS



parameterize the residual graph with  $\Delta$ .

$G_f(\Delta)$  restricts the edges to be considered in the residual graph

### Idea

- Choose paths with large bottlenecks.
- Let  $G_f(\Delta)$  be a residual graph with edges of residual capacity  $\geq \Delta$  or at least  $\Delta$ .

### Scaled Version of the FF method.

- Initialize  $f(e) = 0$  for all edges.
- Initialize  $\Delta := \max_i (2^i)$  such that  $2^i \leq \max_e \text{out of } s(c_e)$ .
- While  $\Delta \geq 1$ :
  - While  $G_f(\Delta)$  contains an augmenting path  $P$ :
  - Update flow  $f$  by BOTTLENECK( $P, G_f(\Delta)$ ) along  $P$ .
  - Set  $\Delta := \Delta/2$ .

two white loops.

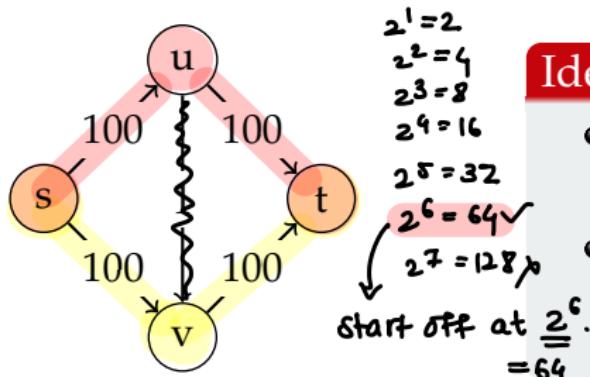
going through values of  $\Delta$ .

largest edge leaving  $s$ .

log ff

for every iteration, we shrink  $\Delta$  by a factor of 2.

# CHOOSING GOOD AUGMENTING PATHS



## Idea

- Choose paths with large bottlenecks.
- Let  $G_f(\Delta)$  be a residual graph with edges of residual capacity  $\geq \Delta$ .

all your edges in the residual graph should have min capacity of 64.

## Scaled Version

- Initialize  $f(e) = 0$  for all edges.
- Initialize  $\Delta := \max_i (2^i)$  such that  $2^i \leq \max_e \text{out of } s(c_e)$ .
- While  $\Delta \geq 1$ :
  - While  $G_f(\Delta)$  contains an augmenting path  $P$ :
    - Update flow  $f$  by  $\text{BOTTLENECK}(P, G_f(\Delta))$  along  $P$ .
  - Set  $\Delta := \Delta/2$ .

→ duality between  
min cut and max flow.

# MINIMUM CUT

---

# MAX-FLOW AND MIN-CUT

## Recall Cut

- A Cut: Partition of  $V$  into sets  $(A, B)$  with  $s \in A$  and  $t \in B$ .
- Cut capacity:  $c(A, B) = \sum_{e \text{ out of } A} c_e$ .  
*↳ outgoing edges leaving A*

# MAX-FLOW AND MIN-CUT

**Lemma 4** -claim.

value of flow =  $\frac{\text{amt of flow going from source to sink}}$

Let  $f$  be any  $s - t$  flow and  $(A, B)$  be any  $s - t$  cut. Then,

$$\begin{array}{l} \xrightarrow{\text{flow fn.}} v(f) = f^{\text{out}}(A) - f^{\text{in}}(A) = f^{\text{in}}(B) - f^{\text{out}}(B). \\ \text{↳ value can be found} \end{array}$$

Proof. for Lemma 4.

- By definition,  $f^{\text{out}}(A) = f^{\text{in}}(B)$  and  $f^{\text{in}}(A) = f^{\text{out}}(B)$ .
- By definition,  $v(f) = f^{\text{out}}(s)$

conservation constraint.

for all internal nodes, flow in

has to be equal

to flow out

sum across  
all nodes in A.

$$= f^{\text{out}}(s) - f^{\text{in}}(s)$$

Flow into source = 0.

$$= \sum_{v \in A} (f^{\text{out}}(v) - f^{\text{in}}(v))$$

for all internal nodes in

(out - in) = v(f)

its 0, but for source its non-zero sum

- Last line follows since  $\sum_{v \in A \setminus \{s\}} (f^{\text{out}}(v) - f^{\text{in}}(v)) = 0$ .

$$\sum_{v \in A} (f^{\text{out}}(v) - f^{\text{in}}(v)) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = f^{\text{out}}(A) - f^{\text{in}}(A).$$

nodes → edges.      e out of A      e into A

□

# MAX-FLOW AND MIN-CUT

## Lemma 4

Let  $f$  be any  $s - t$  flow and  $(A, B)$  be any  $s - t$  cut. Then,

$$v(f) = f^{\text{out}}(A) - f^{\text{in}}(A) = f^{\text{in}}(B) - f^{\text{out}}(B).$$

*→ cut capacity should be an upper bound on the max flow that can run through the graph.*

Let  $f$  be any  $s - t$  flow and  $(A, B)$  be any  $s - t$  cut. Then,

$$v(f) \leq c(A, B).$$

## Proof.



cut capacity of flow going from  $A \rightarrow B$  will be an upper bound on max flow.

$$\begin{aligned} v(f) &= f^{\text{out}}(A) - f^{\text{in}}(A) \leq f^{\text{out}}(A) = \sum_{e \text{ out of } A} f(e) \\ &\leq \sum_{e \text{ out of } A} c_e = c(A, B) \end{aligned}$$

valid flow fact: flow along an edge cannot go beyond capacity.  
 cut capacity.  
 at most capacity of edge.

□

# MAX-FLOW EQUALS MIN-CUT

Theorem 6 → FF method is optimal.

If  $f$  is a  $s - t$  flow such that there is no  $s - t$  path in  $G_f$ , then there is an  $s - t$  cut  $(A^*, B^*)$  in  $G$  for which  $v(f) = c(A^*, B^*)$ .

→ we have a flow fn, where if we create the residual graph, based on the Proof. flow fn, then there will be no paths from  $s \rightarrow t$  in that residual graph.

- Let  $A^*$  be the

$$\text{Let } B^* = V \setminus A^*$$

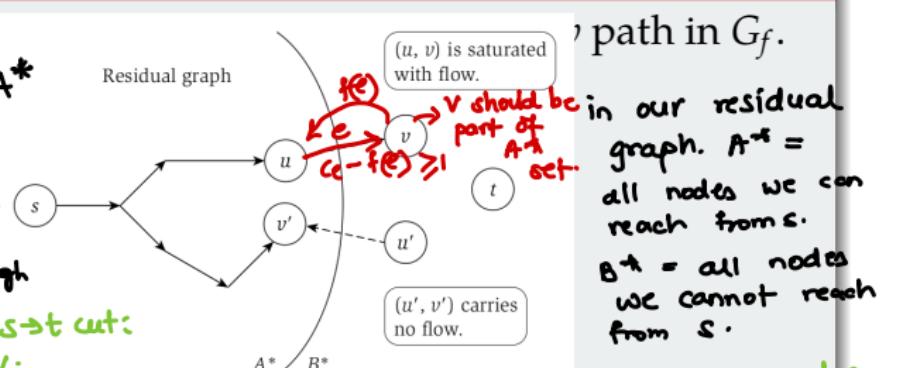
FF keeps going until there are no augmenting  $s \rightarrow t$  paths. (you can't push any more flow through it).

- $(A^*, B^*)$  is an  $s \rightarrow t$  cut:

- Partition of  $V$ .
- $s \in A^*$ ,  $t \in B^*$

- Consider  $e = (u, v)$ : Claim  $f(e) = c_e$ .

- If not, then  $s - v$  path in  $G_f$  which contradicts definition of  $A^*$  and  $B^*$ , if  $f(e) < c_e$



path in  $G_f$ .

in our residual graph.  $A^*$  = all nodes we can reach from  $s$ .

$B^*$  = all nodes we cannot reach from  $s$ .

you show all outgoing edges are saturated, incoming edges have no flow.

# MAX-FLOW EQUALS MIN-CUT

## Theorem 6

If  $f$  is a  $s - t$  flow such that there is no  $s - t$  path in  $G_f$ , then there is an  $s - t$  cut  $(A^*, B^*)$  in  $G$  for which  $v(f) = c(A^*, B^*)$ .

Corollary 7 the cut we described in theorem 6 = min cut.

Let  $f$  be flow from  $G_f$  with no  $s - t$  path. Then,  $v(f) = c(A^*, B^*)$  for minimum cut  $(A^*, B^*)$ .

Proof.  $\rightarrow$  contradiction.

$\rightarrow$  some other flow  $f'$  greater than  $f$ .

- By way of contradiction, assume  $v(f') > v(f)$ . This implies that  $v(f') > c(A^*, B^*)$  which contradicts Lemma 5.  $\rightarrow$  any cut is an upper bound to  $f$ .
- By way of contradiction, assume  $c(A, B) < c(A^*, B^*)$ . This implies that  $c(A, B) < v(f)$  which contradicts Lemma 5.  $\rightarrow$  any flow  $f$  is at least as large as  $c(A, B)$ .



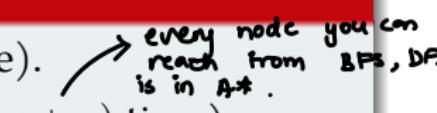
# FINDING THE MIN-CUT

## Theorem 9

Given a maximum flow  $f$ , an  $s - t$  cut of minimum capacity can be found in  $O(m)$  time.

↳ linear time.

## Proof.

- Construct residual graph  $G_f$  ( $O(m)$  time). 
- BFS or DFS from  $s$  to determine  $A^*$  ( $O(\underline{m} + n)$  time).
- $B^* = V \setminus A^*$  ( $O(\underline{n})$  time). 

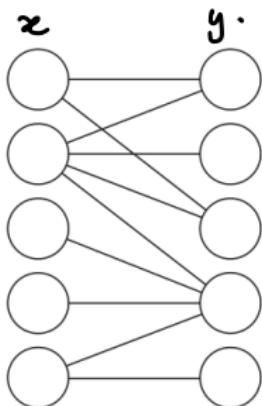


# BIPARTITE MATCHING

$\overleftarrow{\text{L}}$  problem reductions.

$A \rightarrow B$  problem.

# BIPARTITE MATCHING PROBLEM



*→ node appears only once in the edge.*

## Definition

- Bipartite Graph  $G = (V = X \cup Y, E)$ .
- All edges go between  $X$  and  $Y$ .
- Matching:  $M \subseteq E$  s.t. a node appears in only one edge,
- Goal: Find largest matching (cardinality).

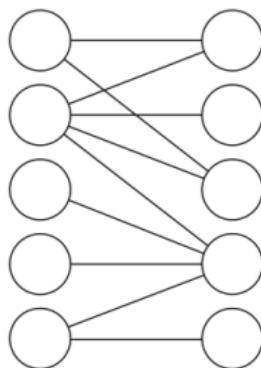
*take the og problem → encode it to a Network Flow problem → then solve.*

## Reduction to Max-Flow Problem

*solution of NF corresponds to the og problem.*

- Goal: Create a flow network based on the original problem.
- The solution to the flow network must correspond to the original problem.
- The reduction should be efficient.  
*→ polynomial time.*

# BIPARTITE MATCHING PROBLEM



## Definition

- Bipartite Graph  $G = (V = X \cup Y, E)$ .
- All edges go between  $X$  and  $Y$ .
- Matching:  $M \subseteq E$  s.t. a node appears in only one edge.
- Goal: Find largest matching (cardinality).

① source ② sink ③ direction ④ capacity.

## Reduction to Max-Flow Problem

*↳ network flow problem.*

✓ How can the problem be encoded in a graph?  
*given the input instance.*

✓ Source/sink: Are they naturally in the graph encoding, or do additional nodes and edges have to be added?

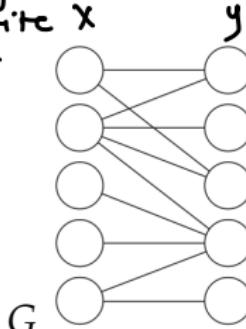
• For each edge: What is the direction? Is it bi-directional?  
 What is the capacity?

*↳ NF are directed graphs.*

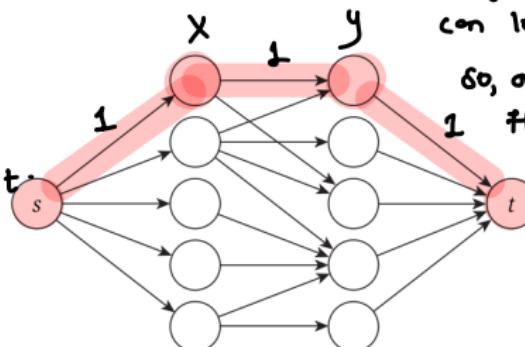
*Can you find source or sink or add them?*

# BIPARTITE MATCHING TO FLOW NETWORK

max flow going from  $x \rightarrow y$   
 $\Rightarrow$  max edges selected in  
 the bipartite  $X$  matching..



Add  $s$  and  $t$ .



only 1 unit of flow can come into  $x$  and only one unit of flow can leave  $y$ . So, only one unit of 1 flow between  $x$  and  $y$ .

- Add source connected to all X.
- Add sink connected to all Y.
- Original edges go from X to Y.
- Capacity of all edges is 1.

you can send at least 1 unit of flow to every node in  $X$ .

$X$  = internal node, so it has to leave  $X$ .

$X$  sends flow to  $y$ . (connected).

$y$  sends flow to  $t$ .

# BIPARTITE MATCHING TO FLOW NETWORK

## Theorem 10

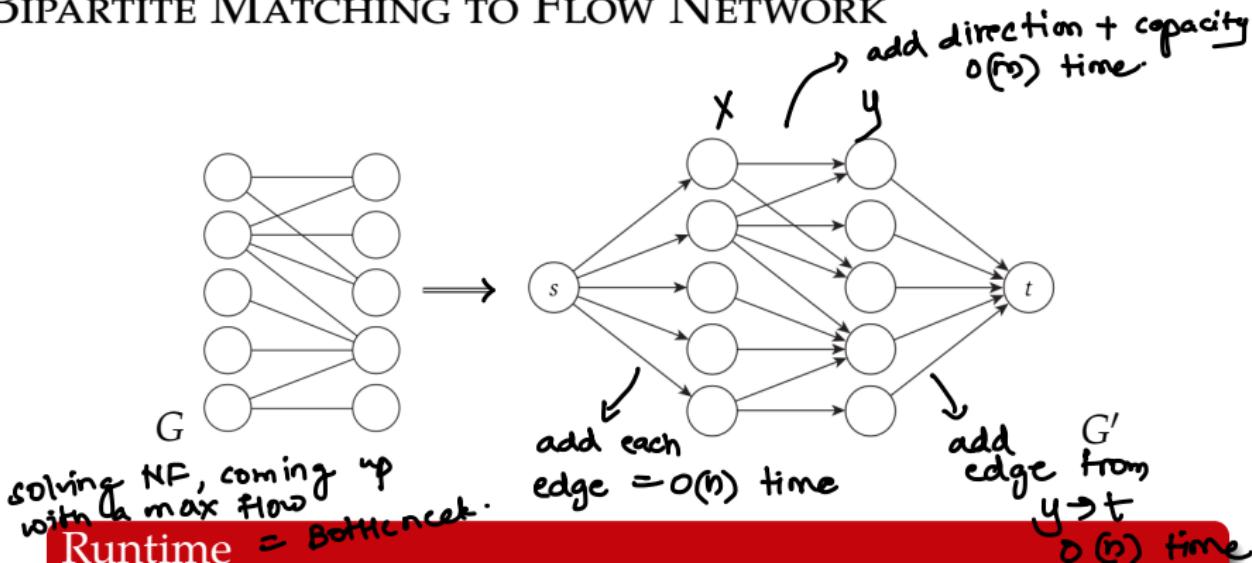
$|M^*|$  in  $G$  is equal to the max-flow of  $G'$ , and the edges carrying the flow correspond to the edges in the maximum matching.

$\rightarrow$  max flow in  $G'$  for reduction is equal to the max matching  
Proof. for the bipartite problem.

- $s$  can send at most 1 unit of flow to each node in  $X$ .
- Since  $f^{\text{in}} = f^{\text{out}}$  for internal nodes,  $Y$  nodes can have at most 1 flow from 1 node in  $X$ .



# BIPARTITE MATCHING TO FLOW NETWORK



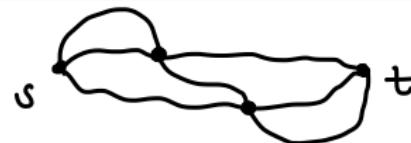
- Assume  $n = |X| = |Y|, m = |E|$ .

- Overall:  $O(mn)$ .

- Basic FF method bound:  $O(mC)$ , where  $C = \frac{n}{m}$ . *n edges st capacity 1.  
1 m = n*

# EDGE-DISJOINT PATHS

# EDGE-DISJOINT PATHS



## Problem

Given a graph  $G = (V, E)$  and two distinguished nodes  $s$  and  $t$ , find the number of edge-disjoint paths from  $s$  to  $t$ .

### Flow Network

- Directed Graph:
  - $s$  is the source and  $t$  is the sink.
  - Add capacity of 1 to every edge.

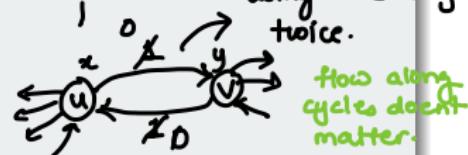
- Undirected Graph:

- For each undirected edge  $(u, v)$ , convert to 2 directed edges  $(u, v)$  and  $(v, u)$ .
- Apply directed graph transformation.

*paths that don't share any edges. set to 0.*

$$\begin{cases} f^{\text{out}}(v) = y+1 & | \\ f^{\text{in}}(v) = y+1 & | \\ f^{\text{out}}(u) = x+1 & | \\ f^{\text{in}}(u) = x+1 & | \\ f^{\text{out}}(y) = x & | \\ f^{\text{in}}(y) = x & | \end{cases}$$

*using this edge twice.*



# EDGE-DISJOINT PATHS ANALYSIS

*reduction = encoding into a new problem*

**Observation 3** *solution to the NF problem contains hints to the eg. prob.* for actual sol.

If there are  $k$  edge-disjoint paths in  $G$  from  $s - t$ , then the max-flow is  $k$  in  $G'$ .

## Runtime

- Basic FF method:  $O(mC) = O(\underline{m}\underline{n})$ .



## Path Decomposition

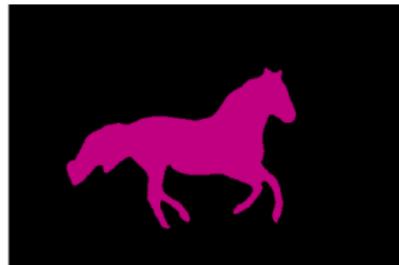
- Let  $f$  be a max-flow for this problem. How can we recover the  $k$  edge-disjoint paths?
- DFS from  $s$  in  $f$  along edges  $e$ , where  $\underline{f}(e) = 1$ :  
  - Find a simple path  $P$  from  $s$  to  $t$ : set flow to  $\underline{0}$  along  $\underline{P}$ ; continue DFS from  $s$ .
  - Find a path  $P$  with a cycle  $C$  before reaching  $t$ : set flow to  $\underline{0}$  along  $\underline{C}$ ; continue DFS from start of cycle.

*to not use edges again*

# IMAGE SEGMENTATION

# IMAGE SEGMENTATION

*↗ figure out least separation penalty value.*



## Problem

Let  $P$  be the set of pixels in an image. We would like to separate  $P$  into set  $A$  and  $B$ , where  $A$  are the foreground pixels and  $B$  are the background pixels. *pixel  $i$  is adjacent to  $j$*

For pixel  $i$ :

- $a_i > 0$  is the likelihood of  $i$  being in the foreground.
- $b_i > 0$  is the likelihood of  $i$  being in the background.
- For each adjacent pixel  $j$ :  $p_{ij} = p_{ji}$  is a symmetric separation penalty, paid when  $i$  and  $j$  are not both  $\in A$  or  $\in B$

# IMAGE SEGMENTATION

## Problem

Let  $P$  be the set of pixels in an image.

For pixel  $i$ :

- $a_i > 0$  is the likelihood of  $i$  being in the foreground.
- $b_i > 0$  is the likelihood of  $i$  being in the background.
- For each adjacent pixel  $j$ :  $p_{ij} = p_{ji}$  is a separation penalty paid when  $i$  and  $j$  are not both  $\in A$  or  $\in B$ .

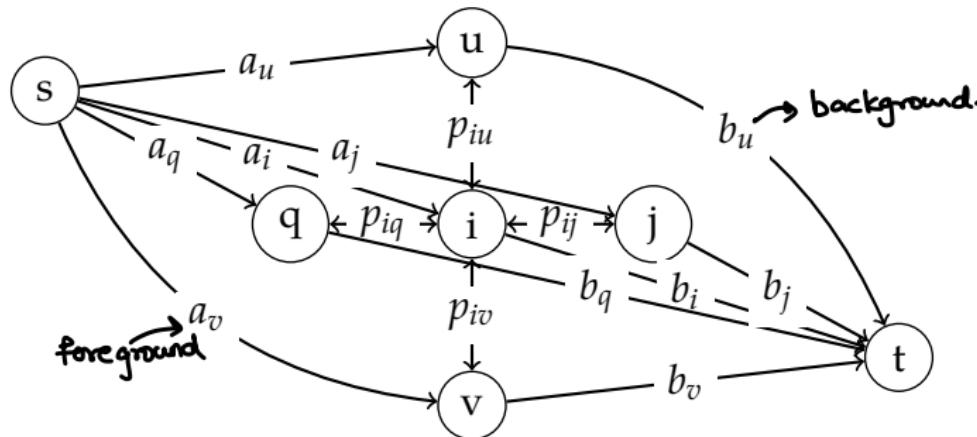
## Goal

(minimize this)  
subtract separation  
penalty.

- Maximize  $\underbrace{q(A, B)}_{= \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{i,j \in P: |A \cap \{i, j\}| = 1} p_{ij}}$
- Let  $\underline{Q} = \sum_{i \in P} (a_i + b_i)$ .  

$$q(A, B) = \underline{Q} - (\underbrace{\sum_{i \in B} a_i - \sum_{j \in A} b_j}_{=} - \sum_{i,j \in P: |A \cap \{i, j\}| = 1} p_{ij})$$
 minimize.
- Equivalent goal:  $= \sum_{i \in B} b_i + \sum_{i \in A} a_i$
- ✓ Minimize  $\sum_{i \in B} a_i + \sum_{j \in A} b_j + \sum_{i,j \in P: |A \cap \{i, j\}| = 1} p_{ij}$ .

# ALGORITHM DESIGN



## Reduction

- Each pixel becomes a node,
- Add edges between neighbours  $i$  and  $j$  with capacity  $p_{ij}$ .
- Add a source  $s$  and connect to all nodes  $i$  with capacity  $a_i$ .
- Add a sink  $t$  and connect all nodes  $i$  with capacity  $b_i$  to  $t$ .

# ALGORITHM DESIGN

*cut capacity =  $\sum$  outgoing edges from A.*

max flow  
value

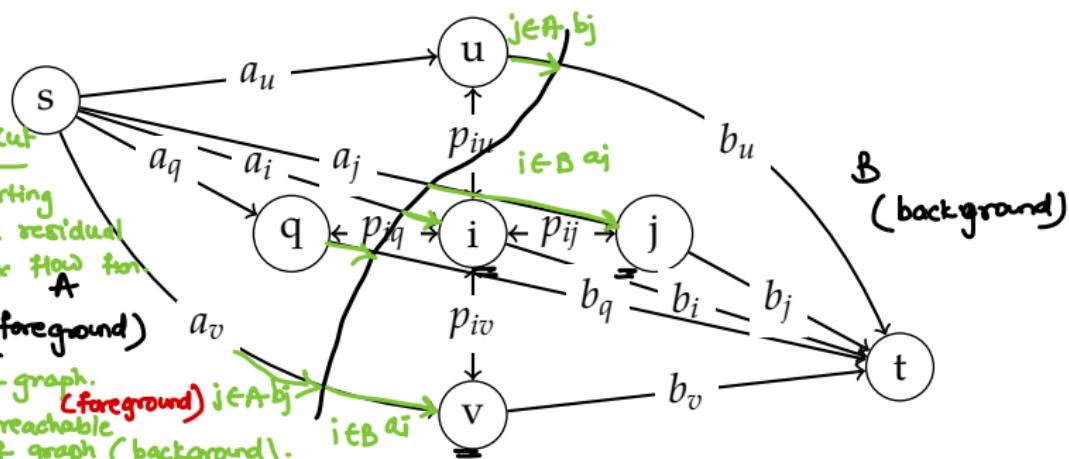
= value of min-cut

BFS / DFS : starting from s on the residual graph of max flow from A

All nodes reachable from s (foreground)

s = A side of graph.

(foreground)  $j \in A \setminus B$   
All nodes not reachable = B side of graph (background).



## Solution

- Min-cut will minimize  $\sum_{i \in B} a_i + \sum_{j \in A} b_j + \sum_{i,j \in P: |A \cap \{i,j\}|=1} p_{ij}$ .
- Consider  $i \in A$ : Foreground and contributes  $b_i$  to cut.
- Consider  $j \in B$ : Background and contributes  $a_i$  to cut.
- Consider  $i \in A, j \in B$  and  $i, j$  adjacent: contributes  $p_{ij}$  to cut.

*value of min cut  
= value we are  
trying to  
minimize.*

# Node Demand and Lower Bounds

# FLOW NETWORK EXTENSION

## ADDING NODE DEMAND

~~Any node can add/remove flow~~

### Flow Network with Demand

- Each node has a demand  $\underline{d}_v$ :
  - if  $\underline{d}_v < 0$ : a source that demands  $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$ .
  - if  $\underline{d}_v = 0$ : internal node ( $f^{\text{in}}(v) - f^{\text{out}}(v) = 0$ ).
  - if  $\underline{d}_v > 0$ : a sink that demands  $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$ .
- $S$  is the set of sources ( $d_v < 0$ ).
- $T$  is the set of sinks ( $d_v > 0$ ).

### Flow Conditions

*same as NF for capacity.*

- Capacity: For each  $e \in E$ ,  $0 \leq f(e) \leq c_e$ .

- Conservation: For each  $v \in V$ ,  $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$ .

*unchanged.*

*flow in - flow out equals demand.*

### Goal

Feasibility: Does there exist a flow that satisfies the conditions?

# FEASIBILITY

## Goal

Feasibility: Does there exist a flow that satisfies the conditions?

### Lemma 10

If there is a feasible flow, then  $\sum_{v \in V} d_v = 0$ .

$\rightarrow$  total demand = 0.

$d_s < 0$  } cancels out  
 $d_t > 0$

this if and only if  
feasible.

## Proof.

- Suppose that  $f$  is a feasible flow, then, by definition, for all  $v$ ,  $d_v = f^{\text{in}}(v) - f^{\text{out}}(v)$ . 
- For every edge  $e = (u, v)$ ,  $f_e^{\text{out}}(u) = f_e^{\text{in}}(v)$ . Hence,  

$$f_e^{\text{in}}(v) - f_e^{\text{out}}(u) = 0.$$

$$\begin{aligned} du &= f^{\text{in}}(u) - f^{\text{out}}(u) \\ &= f^{\text{in}}(u) - f_e^{\text{out}}(v) - f_e^{\text{out}}(u) \\ dv &= f^{\text{in}}(v) - f^{\text{out}}(v) \\ &= f_e^{\text{in}}(v) + f_e^{\text{in}}(v) - f_e^{\text{out}}(v) \end{aligned}$$
□
- $\sum_{v \in V} d_v = 0.$

# FEASIBILITY

## Goal

Feasibility: Does there exist a flow that satisfies the conditions?

## Lemma 10

If there is a feasible flow, then  $\sum_{v \in V} d_v = 0$ .

## Corollary 11

If there is a feasible flow, then

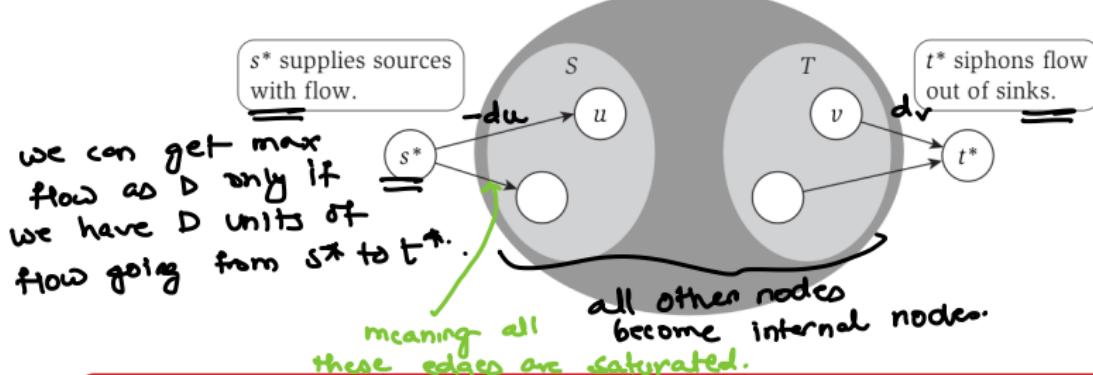
$$D = \sum_{\substack{v: d_v > 0 \in V \\ \text{sink nodes}}} d_v = \sum_{\substack{v: d_v < 0 \in V \\ \text{source nodes}}} -d_v$$

Not iff

Feasibility  $\implies \sum_{v \in V} d_v = 0$ , but  $\sum_{v \in V} d_v = 0 \not\implies$  feasibility.

$\Rightarrow$  demands  $\rightarrow$  max flow reduction : show equal computational power.

## REDUCTION TO MAX-FLOW



### Reduction from $G$ (demands) to $G'$ (no demands)

- Super source  $s^*$ : Edges from  $s^*$  to all  $v \in S$  with  $d_V < 0$  with capacity  $-d_v$ .
- Super sink  $t^*$ : Edges from all  $v \in T$  with  $d_V > 0$  to  $t^*$  with capacity  $d_v$ .
- Maximum flow of  $\underline{D} = \sum_{v:d_v>0 \in V} d_v = \sum_{v:d_v<0 \in V} -d_v$  in  $G'$  shows feasibility.

# ANOTHER FLOW NETWORK EXTENSION

ADDING FLOW LOWER BOUND on edges. in addition to demands on nodes.

## Adding Lower Bound

- For each edge  $e$ , define a lower bound  $\underline{l}_e$ , where  $0 \leq \underline{l}_e \leq \overline{c}_e$ .

## Flow Conditions

- Capacity: For each  $e \in E$ ,  $\underline{l}_e \leq f(e) \leq \overline{c}_e$ .
- Conservation: For each  $v \in V$ ,  $f^{\text{in}}(v) - f^{\text{out}}(v) = \underline{d}_v$ .

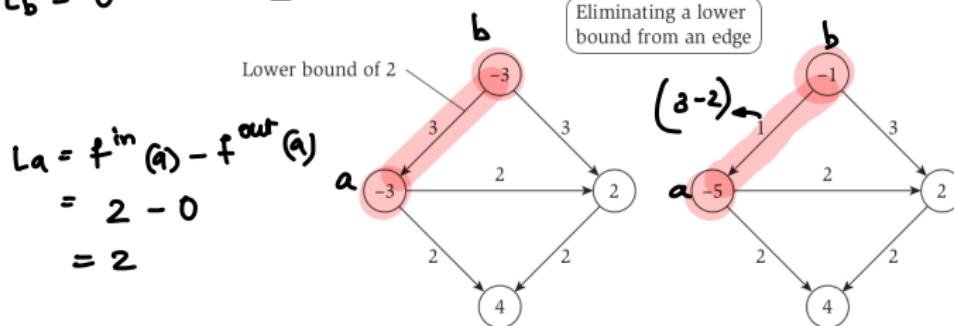
## Goal

Feasibility: Does there exist a flow that satisfies the conditions?

total two reductions demand + LB → demand → NF

## REDUCTION TO ONLY DEMAND

$$L_b = 0 - 2 = -2$$



$$\begin{aligned} d'a &= da - La \\ &= -3 - 2 \\ &= -5 \end{aligned}$$

$$\begin{aligned} d'b &= db - Lb \\ &= -3 - (-2) \\ &= -3 + 2 \\ &= 1 \end{aligned}$$

takes lower bound and pushes into demands.

(Step 1: Reduction from  $G$  (demand + LB) to  $G'$  (demand))

- Consider an  $f_0$  that sets all edge flows to  $\ell_e$ :  
equal to lower bound on all edges.

$$\underline{L_v} = f_0^{\text{in}}(v) = f_0^{\text{out}}(v).$$

- if  $L_v = d_v$ : Condition is satisfied.
- if  $L_v \neq d_v$ : Imbalance.

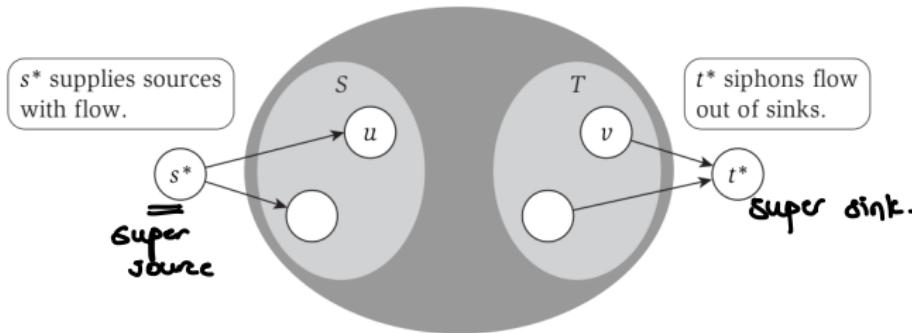
- For  $G'$ : graph with demand only
  - ① update edges
  - ② update demand.

- Each edge  $e$ ,  $c'_e = c_e - \ell_e$  and  $\ell_e = 0$ .

- Each node  $v$ ,  $d'_v = d_v - L_v$ .  $\rightarrow$  update demands on nodes

adjacent to  
edges having  
LB.

$\curvearrowleft$  2nd reduction.  
**REDUCTION TO ONLY DEMAND**



### Step 2: Reduction from $G'$ (demand) to $G''$ (no demand)

- Super source  $s^*$ : Edges from  $s^*$  to all  $v \in S$  with  $d_V < 0$  with capacity  $-d_v$ .
- Super sink  $t^*$ : Edges from all  $v \in T$  with  $d_V > 0$  with capacity  $d_v$  to  $t^*$ .
- Maximum flow of  $D = \sum_{v:d_v>0 \in V} d_v = \sum_{v:d_v<0 \in V} -d_v$  in  $G'$  shows feasibility.

# SURVEY DESIGN

---

# SURVEY DESIGN

## Problem

- Study of consumer preferences.
- A company, with  $k$  products, has a database of  $n$  customer purchase histories.
- Goal: Define a product specific survey.



Bipartite graph.

$X = \text{customers}$

$y = \text{products}$ .

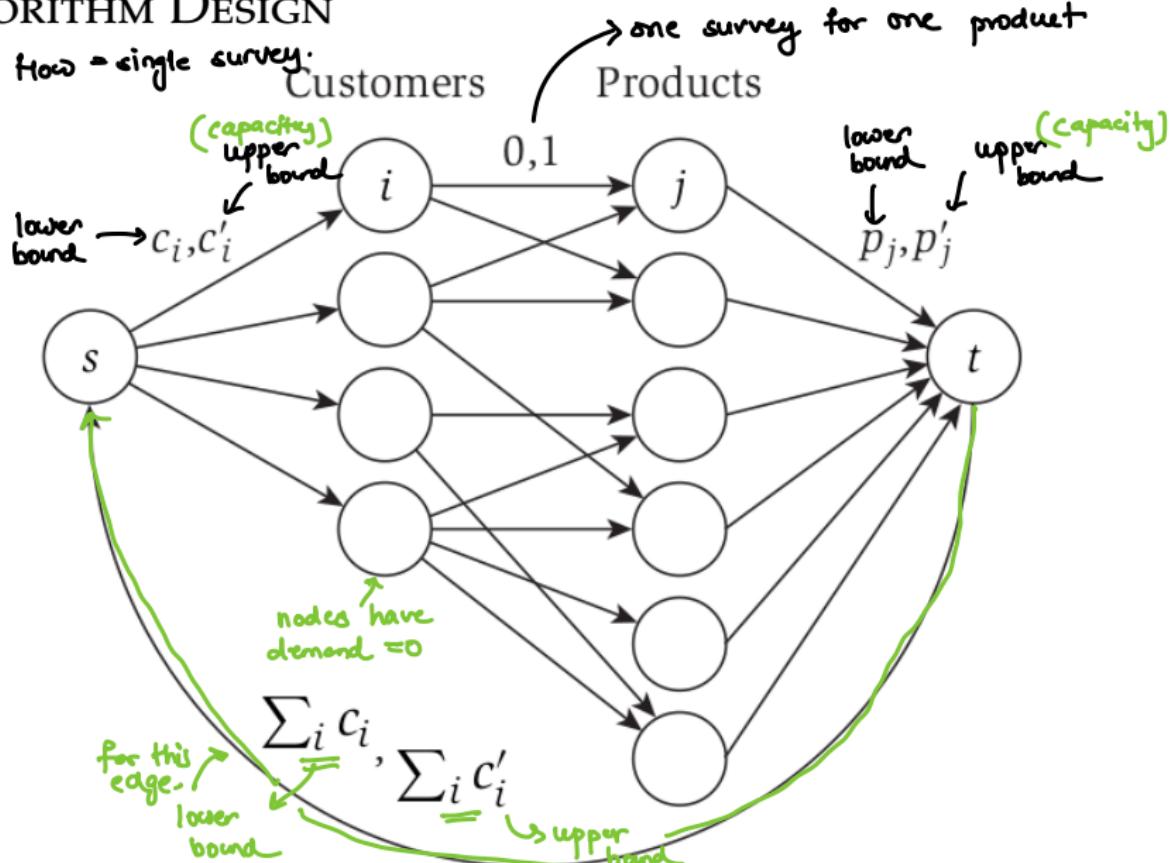
## Survey Rules

- Each customer receives a survey based on their purchases.
- Customer  $i$  will be asked about at least  $c_i$  and at most  $c'_i$  products.  
→  $c_i$  customer constraint
- To be useful, each product must appear in at least  $p_i$  and at most  $p'_i$  surveys.  
→  $p_i$  product constraint

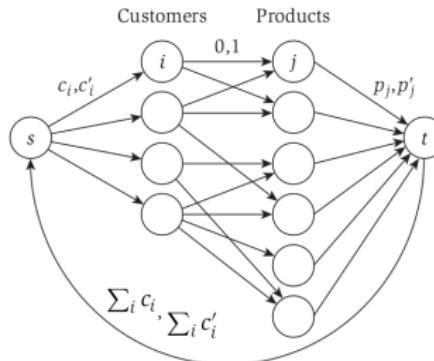
Lower bound + demand bipartite matching problem.

## ALGORITHM DESIGN

1 unit flow = single survey.



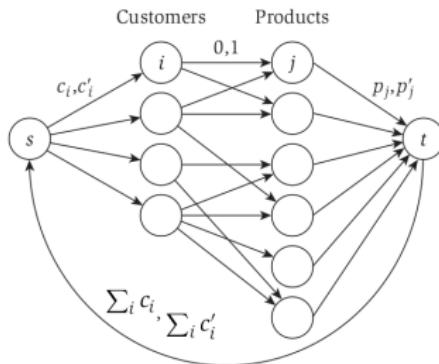
# ALGORITHM DESIGN



## Reduction

- Bipartite Graph: Customers to products with min of 0 and max of 1.
- Add  $s$  with edges to customer  $i$  with min of  $c_i$  and max of  $c'_i$ .
- Add  $t$  with edges from product  $j$  with min  $p_j$  and max of  $p'_j$ .
- Edge  $(t, s)$  with min  $\sum_i c_i$  and max  $\sum_i c'_i$ .
- All nodes have a demand of 0.

# ALGORITHM DESIGN



## Solution

- Feasibility means it is possible to meet the constraints.
  - Edge  $(i, j)$  carries flow if customer  $i$  asked about product  $j$ .
  - Flow  $(t, s)$  overall # of questions.
  - Flow  $(s, i)$  # of products evaluated by customer  $i$ .
  - Flow  $(j, t)$  # of customers asked about product  $j$ .
- flow between customers and products means we send customer i survey of product j.*
- no. of surveys sent out.*
- no. of surveys sent to customer i.*
- = no. of surveys asked about product-j.*

# AIRLINE SCHEDULING

---

---

# AIRLINE SCHEDULING

$k = 2$ .

Flights: (2 airplanes)

- Flight segments.*
- ① Boston (6 am) – Washington DC (7 am)
  - ② Philadelphia (7 am) – Pittsburgh (8 am)
  - ③ Washington DC (8 am) – Los Angeles (11 am)
  - ④ Philadelphia (11 am) – San Francisco (2 pm)
  - ⑤ San Francisco (2:15 pm) – Seattle (3:15 pm)
  - ⑥ Las Vegas (5 pm) – Seattle (6 pm)

## Simple Version

- Scheduling a fleet of  $k$  airplanes.
- $m$  <sup>edges</sup> flight segments, for segment  $i$ :
  - Origin and departure time.
  - Destination and arrival time.

→ Are we able to schedule all flight segments given we have k planes?

## AIRLINE SCHEDULING

Flights: (2 airplanes)

- 
- ① Boston (6 am) – Washington DC (7 am)
  - ② Philadelphia (7 am) – Pittsburgh (8 am)
  - ③ Washington DC (8 am) – Los Angeles (11 am)
  - ④ Philadelphia (11 am) – San Francisco (2 pm)
  - ⑤ San Francisco (2:15 pm) – Seattle (3:15 pm)
  - ⑥ Las Vegas (5 pm) – Seattle (6 pm)

we can service 2 different flight segments if the destination of 1 Rules is the same as origin of 2.

The same plane can be used for flight  $i$  and  $j$  if:

- $i$  destination is the same as  $j$  origin and there is enough time for maintenance between  $i$  arrival and  $j$  departure;
- Or, there is enough time for maintenance and to fly from  $i$  destination to  $j$  origin.

How might you represent this as a graph?

solid line  $\rightarrow$  flight segment.

## ALGORITHM DESIGN

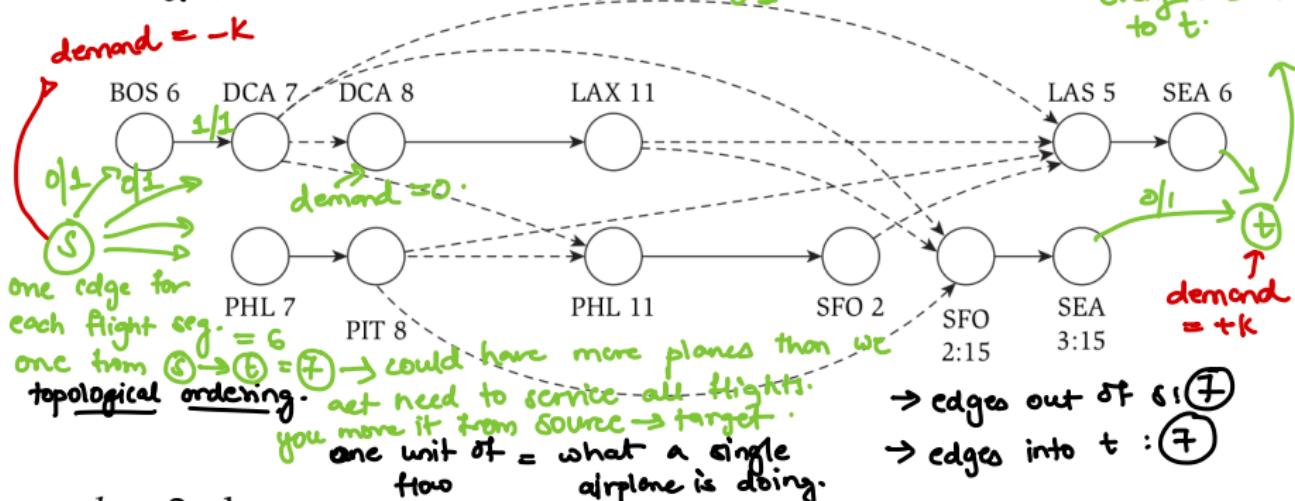
dashed line  $\rightarrow$  the other flights/destinations it can go to.

\* edge  $e$  from  $s \rightarrow t$   
lower bound 0, upper bound:  $k$ .

0..1

every destination to  $t$ .

demand =  $-k$



$$k = 2 \text{ planes}$$

$\rightarrow$  use lower bounds and demands to model the problem.

### Exercise: Reduce to a flow network

Hint: Use lower bounds and demand.

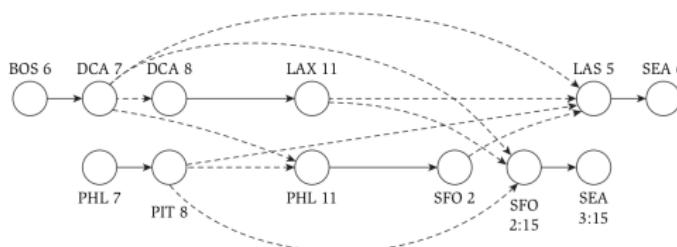
- TH17: Are  $s-t$  new nodes?  $\rightarrow$  add new nodes.

$$\text{capacity} = 1$$

- TH18: What is the max capacity of the edges from  $G$ ?  $\rightarrow$  graph G

# ALGORITHM DESIGN

$k = 2$  planes

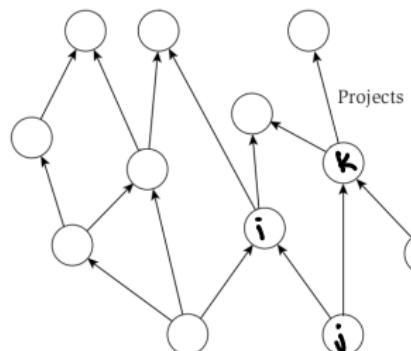


## Reduction

- Units of flow correspond to airplanes.
- Each edge of a flight has capacity  $(1, 1)$ .
- Each edge between flights has capacity of  $(0, 1)$ .
- Add node  $s$  with edges to all origins with capacity of  $(0, 1)$ .
- Add node  $t$  with edges from all destinations with cap  $(0, 1)$ .
- Edge  $(s, t)$  with a min of 0 and a max of  $k$ .
- Demand:  $d_s = -k, d_t = k, d_v = 0 \forall v \in V \setminus \{s, t\}$ .

# PROJECT SELECTION

# PROJECT SELECTION



Use Min-Cut to solve this problem.

↳ what projects you have to do before you can do the one below it.  
if j, you need to complete i and k.

## Problem

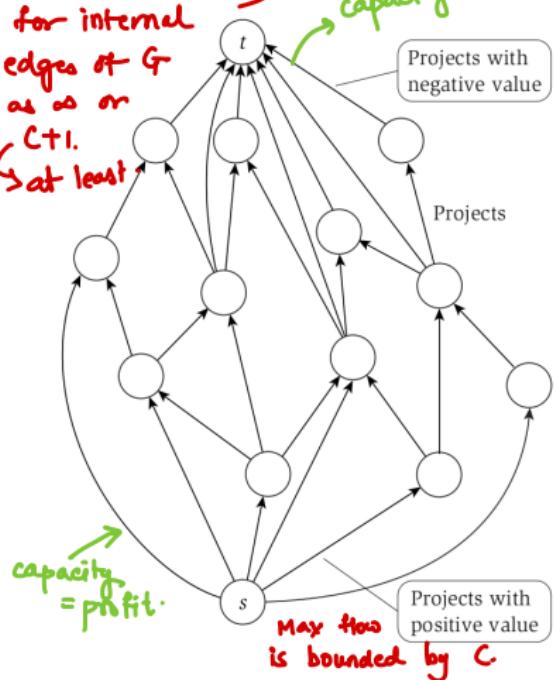
- Set of projects:  $P$ .
  - Each  $i \in P$ : profit  $p_i$  (which can be negative).
  - Directed graph  $G$  encoding precedence constraints.
  - Feasible set of projects  $A$ : PROFIT( $A$ ) =  $\sum_{i \in A} p_i$ .
  - Goal: Find  $A^*$  that maximizes profit.
- ⇒ set of projects with max profit.

# ALGORITHM DESIGN

① set the capacity

for internal  
edges of  $G$   
as  $\infty$  or  
 $C+1$ .

(at least)

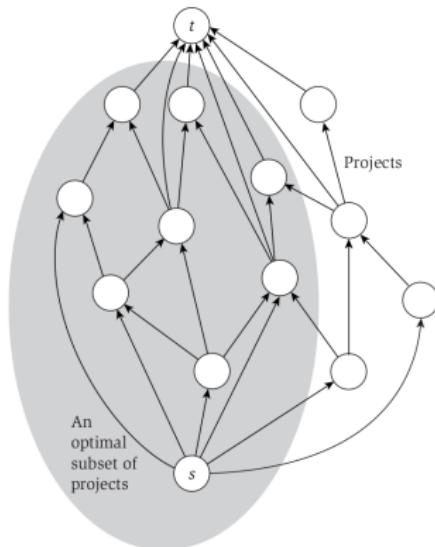


Reduction

→ image segmentation like  
 $s$ : projects with +ve values  
 $t$ : projects with -ve values.

- Use Min-Cut  $t$
- Add  $s$  with edge to every project  $i$  with  $p_i > 0$  and capacity  $\underline{p_i}$ .
- Add  $t$  with edge from every project  $i$  with  $p_i < 0$  and capacity  $-\underline{p_i}$ .  $C = \sum_{i \in P: p_i > 0} p_i$
- Max-flow is  $\leq C = \sum_{i \in P: p_i > 0} p_i$ .
- For edges of  $G$ , capacity is  $\infty$  (or  $C + 1$ ).

# ALGORITHM ANALYSIS



## Observation 4

If  $c(A', B') \leq C$ , then  $A = A' \setminus \{s\}$  satisfies precedence as edges of  $G$  have capacity  $> C$ .

## Lemma 12

Let  $(A', B')$  be a cut satisfies precedence; then

$$c(A', B') = C - \sum_{i \in A} p_i.$$

*Cut capacity*      *sum of profits in the set*

## Proof.

Consider the different edges:

- $(i, t)$ :  $-p_i$  for  $i \in A$ .
- $(s, i)$ :  $p_i$  for  $i \notin A$ .

$$c(A', B') = \sum_{i \in A: p_i < 0} -p_i + C - \sum_{i \in A: p_i > 0} p_i = C - \sum_{i \in A} p_i$$

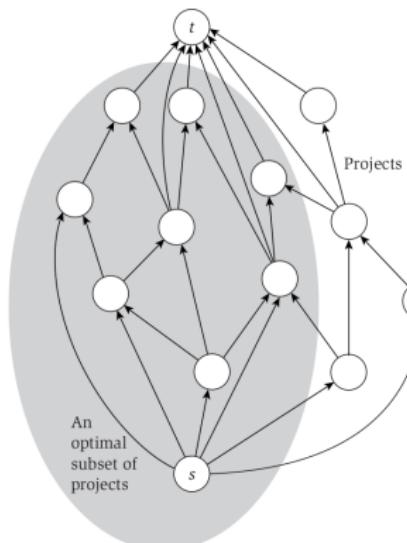
□

# ALGORITHM ANALYSIS

you find a min-cut on all the ~~(-ve)~~ profit value and exclude that set. The set you are left with is the answer. (most profit).

## Theorem 12

If  $(A', B')$  is a min-cut in  $G'$ , then  $A = A' \setminus \{s\}$  is an optimal solution.



finding  
min cut  
ends up maximizing  
profit

Proof.  $\rightarrow$  same thing as image segmentation

- Obs:  $c(A', B') = C - \sum_{i \in A} p_i$  means feasible.

$$\begin{aligned} \min_{\text{cut}} \rightarrow c(A', B') &= C - \text{PROFIT}(A) \\ \iff \text{PROFIT}(A) &= C - c(A', B') \end{aligned}$$

- Given that  $c(A', B')$  is a minimum, profit is maximized as  $C$  is a constant.



# CS 577 - Computational Intractability

---

---

Marc Renault

Department of Computer Sciences  
University of Wisconsin – Madison

Fall 2023

TopHat Section 001 Join Code: 477366  
TopHat Section 002 Join Code: 560750



# COMPUTATIONAL INTRACTABILITY

↳ theory of computational stuff.

## Easy Problems

- Problems that can be solved by efficient algorithms.
- Polynomial running time.
- Complexity class: P  
↳ polynomial time solution.

## Hard Problems

NP (non deterministic polynomial time).

- Problems for which we do not know how to solve efficiently.
  - NP-hard
  - NP-complete
- 

# DECISION PROBLEM

you find the solution  
 Optimization:

## Bipartite Matching

Given a bipartite graph  $G$ ,  
 find the largest matching.

solve using network flow

### Decision Problem

- binary output: yes / no answer.

all decision problems  
 will have an output yes/no.

you say yes/no.  
 Decision:

## Bipartite Matching

Given a bipartite graph  $G$ , is  
 there a matching of size  $\geq k$ ?

output: yes/no

# DECISION PROBLEM

Optimization:

## Bipartite Matching

Given a bipartite graph  $G$ ,  
find the largest matching.



Decision:

## Bipartite Matching

Given a bipartite graph  $G$ , is  
there a matching of size  $\geq k$ ?

*you increase  $k$  till the  
answer you get is no.*

*use binary search to find the  
optimal value.*

## Optimization to Decision

- Solve the optimization version.
- If the solution of size  $\geq k$ , return yes.

# DECISION PROBLEM

Optimization:

## Bipartite Matching

Given a bipartite graph  $G$ ,  
find the largest matching.



Decision:

## Bipartite Matching

Given a bipartite graph  $G$ , is  
there a matching of size  $\geq k$ ?

## Decision to Optimization

- Upper bound on maximum matching is  $N = \min(|A|, |B|)$ .
- For  $k = N$  to 0, return first  $k$  that returns yes.  
(Or, binary search between  $[0, \underline{N}]$ .)

↳ more efficient

# REDUCTIONS

=

↳ make claims if a problem  
is hard or not.

## POLYNOMIAL-TIME REDUCTION

Problem Reduction:  $Y \leq_p X$  reduction from  $y \rightarrow x$

other problems  
↑  
basic NP problem

- Consider any instance of problem  $Y$ . *solve  $X$  somehow in polynomial time.*
- Assume we have a black-box solver for problem  $X$ .
- Efficiently transform an instance of problem  $Y$  into a polynomial number of instances of  $X$  that we solve (black-box solver) for problem  $X$  and aggregate efficiently to solve  $Y$ . *yes instance polynomial time*, *yes instance*, *black-box solver  $X$*

*becomes input*  $\xleftarrow{\text{(instance)}} I \in Y \xrightarrow{ \leq_p } I' \in X \xrightarrow{\text{sol}(I')}$   $\xrightarrow{\text{sol}(I)}$  *polynomial time.*

Suppose  $Y \leq_p X$ . If  $X$  is solvable in polynomial time, then  $Y$  can be solved in polynomial time.

$X$  is at least as hard as  $Y$  *→ contrapositive.*

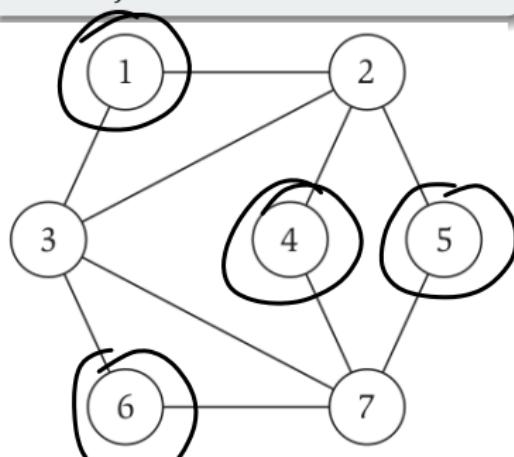
Suppose  $Y \leq_p X$ . If  $Y$  cannot be solved in polynomial time, then  $X$  cannot be solved in polynomial time.

# INDEPENDENT SET $\iff$ VERTEX COVER

Given a graph  $G$  and a number  $k$ .

## Independent Set (IS)

- Does  $G$  contain an IS of size  $\geq k$ ?
- $S \subseteq V$  is *independent* if no 2 nodes in  $S$  are adjacent.



## Vertex Cover (VC)

- Does  $G$  contain a vertex cover of size  $\leq k$ ?
- $S \subseteq V$  is *vertex cover* if every edge is incident to at least 1 node in  $S$ .

TopHat 1: What is size of the largest independent set?

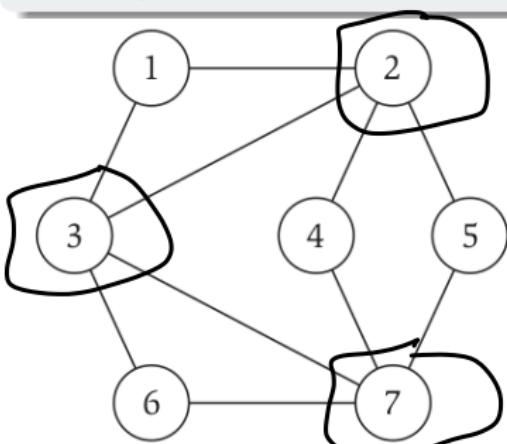
4

# INDEPENDENT SET $\iff$ VERTEX COVER

Given a graph  $G$  and a number  $k$ .

## Independent Set (IS)

- Does  $G$  contain an IS of size  $\geq k$ ?
- $S \subseteq V$  is *independent* if no 2 nodes in  $S$  are adjacent.

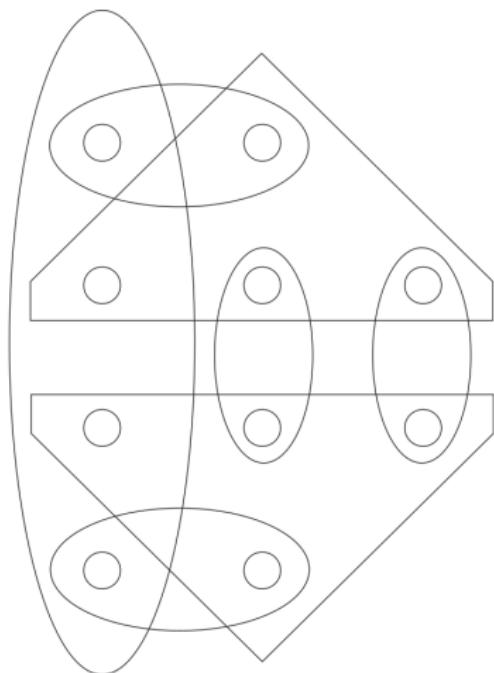


## Vertex Cover (VC)

- Does  $G$  contain a vertex cover of size  $\leq k$ ?
- $S \subseteq V$  is *vertex cover* if every edge is incident to at least 1 node in  $S$ .

TopHat 2: What is size of the smallest vertex cover?

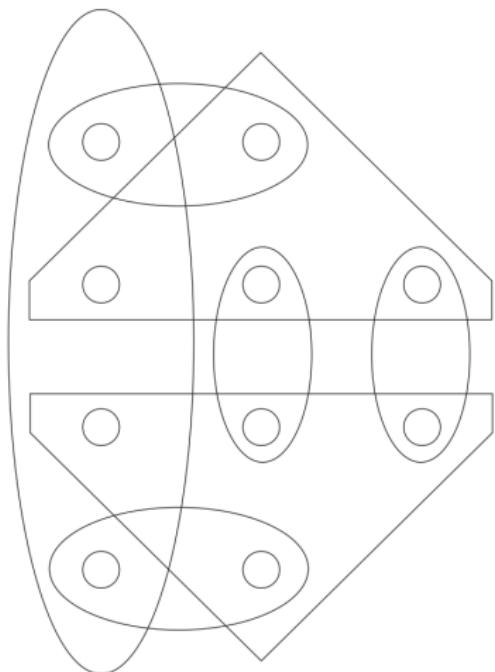
# SET COVER (SC)



## Problem Definition

- A universe  $U$  of  $n$  elements.
- A collection of subsets of  $U$ :  $S_1, S_2, \dots, S_m$ .
- A number  $k$ .
- Goal: Does there exist a collection of at most  $k$  of the subsets whose unions equal  $U$ .

# SET PACKING (SP)



## Problem Definition

- A universe  $U$  of  $n$  elements.
- A collection of subsets of  $U$ :  $S_1, S_2, \dots, S_m$ .
- A number  $k$ .
- Goal: Does there exist a collection of at least  $k$  of the subsets that don't intersect.

Exercise: Show that IS  $\leq_p$  SP

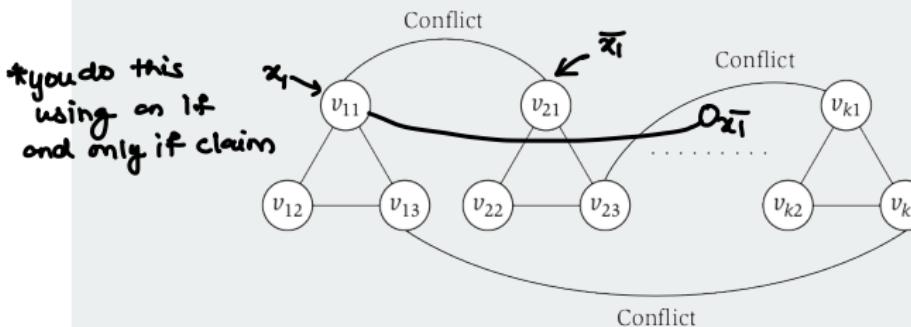
# 3SAT TO INDEPENDENT SET (IS)

## Theorem 4

$\text{3SAT} \leq_p \text{IS}$

## Proof.

- Assume we have a black-box solver for IS.
- Transfer any 3SAT to IS:
  - Clause gadget:  $k_3$  graph



$$\begin{aligned} c_1 &: (x_1 \vee x_2 \vee x_3) \\ c_2 &: (\bar{x}_1 \vee x_2 \vee x_5) \end{aligned}$$

$I \in \text{3SAT} \leq_p I' \in \text{IS}$   
 $\leq_p$  correct:  
 (reduction correct)

\*if  $\text{sol}(I)$  is yes  
 then  
 $\text{sol}(I')$  is yes.  
 if  $\text{sol}(I)$  is no.  
 then  
 $\text{sol}(I')$  is no.

- Add an edge between  $v_{ij} = x_q$  and all  $v_{i'j'} = \bar{x}_q$ .

## EFFICIENT CERTIFICATION

↳ checking if the solution is correct.

### Input Formalization

For a problem instance:

- Let  $s$  be a binary string that encodes the input.
- $|s|$  is the length of  $s$ , i.e., the # of bits in  $s$ .

↳ no. of bits in the input.

### Polynomial Run-Time

polynomial in terms of the no. of bits it takes to encode the input into a binary string.

Algorithm  $A$  has a polynomial run-time if run-time is  $O(\text{poly}(|s|))$  in the worst-case, where  $\text{poly}(\cdot)$  is a polynomial function.

↳ efficient algo.

### Complexity class P

↳ stands for polynomial time.

P is the set of all problems for which there exists an algorithm  $A$  that solves the problem with polynomial run-time.

$\Rightarrow B(s, t)$  returns yes or no based on  $t$ .

## EFFICIENT CERTIFICATION

Compare P to NP.

### Efficient Certification

Certifier  $B(s, t)$  for a problem  $P$ :

$s$  should be efficient.

- $s$  is an input instance of  $P$ .  $\hookrightarrow$  encode the input in binary.

- $t$  is a certificate; a proof that  $s$  is a yes-instance,

- Efficient:  $\hookrightarrow$  that subset of nodes (solution)

certificate has to be polynomial  
in terms  
of  $s$

- For every  $s$ , we have  $s \in P$  iff there exists a  $t$ ,  $|t| \leq \text{poly}(|s|)$ , for which  $B(s, t)$  returns yes.
- In other words, using  $t$ , we can check if  $s$  is a yes-instance in polynomial time.  $\hookrightarrow$  we get to define  $t$  -
- $B(s, t)$  returning no does not mean that  $s$  is a no-instance... only that  $t$  is not a valid proof.  $\hookrightarrow$  you need to check all certificates to find a yes-instance.
- $B(s, t)$  provides a brute-force algorithm: For a given  $s$ , check every possible  $t$ .

# NP PROBLEMS

class of all problems for which we can verify a solution in polynomial time.

## Complexity Class NP

- Non-deterministic, Polynomial time: can be solved in polynomial time by testing every  $t$  simultaneously (non-deterministic) → do multiple calculations simultaneously.
- Set of all problems for which there exists an efficient certifier.

Certifier → Brute force algo. for solving these problems.

### Theorem 5

P ⊆ NP show P is a subset of NP.

#### Proof.

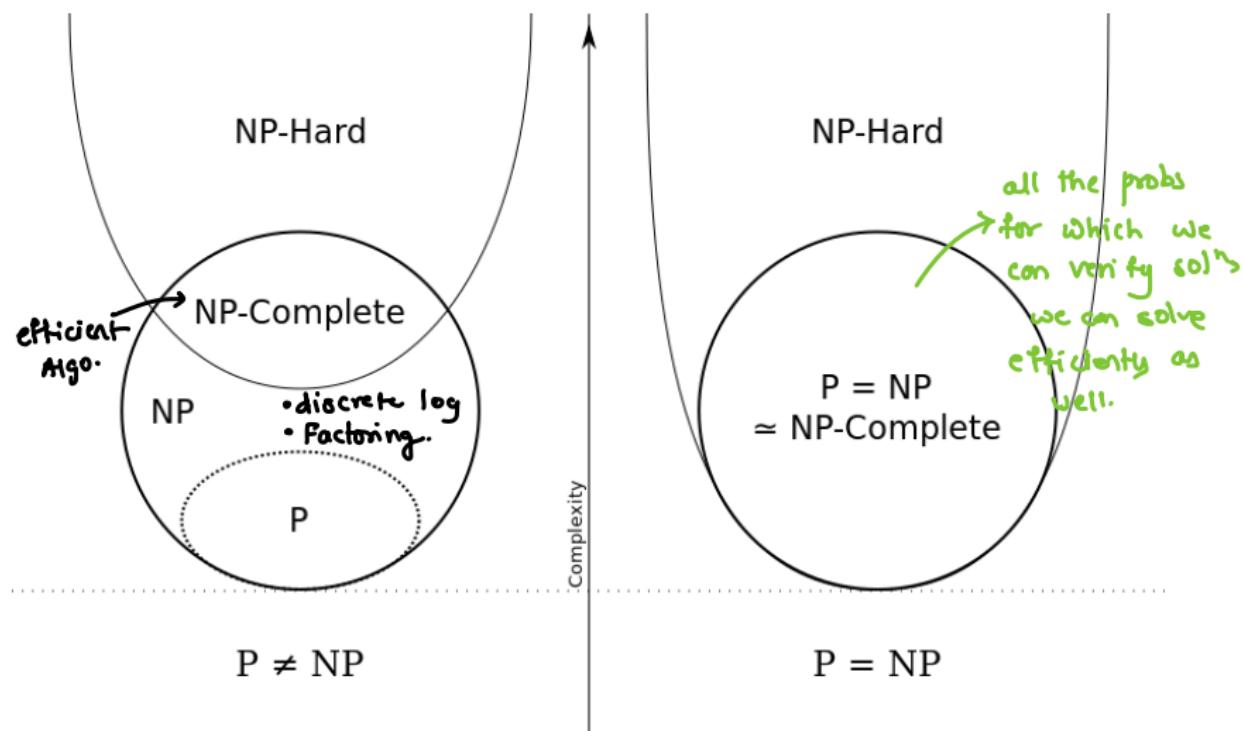
problem in P; there exists an algo A that runs in polynomial time.

- For every  $p \in P$ ,  $\exists$  an algorithm A that runs in polynomial time → efficiently.
- $B(s, t)$  for any  $t$  returns  $A(s)$ . (algo for s).  
certifier gets ignored, as you have an efficient algo.



# MILLION DOLLAR QUESTION: P vs NP

1 OF 7 CLAY MATHEMATICS INSTITUTE MILLENNIUM PRIZE PROBLEMS



# HARDEST NP PROBLEMS

you take any NP problem, then you can reduce it to prob X in polynomial time. X becomes NP-Hard. (strong claim).

## NP-Hard

Problem X is NP-Hard if:  $\rightarrow$  y can be polynomially reduced to X.

- For all  $Y \in \underline{NP}$ ,  $Y \leq_p \underline{X}$ .
- NP-Hard problem may or may not be in NP.

Showing in NP : coming up with an efficient certifier. for that NP-Complete

then coming up with a polynomial time reduction from a known NP-Complete prob to the prob we are trying to show is NP-complete.

- Problem X is NP-Complete if:
- For all  $Y \in \underline{NP}$ ,  $Y \leq_p \underline{X}$ . } same as being NP-Hard.
  - $\underline{X}$  is in NP. } that problem  $X$  also has to be in class NP.

# HARDEST NP PROBLEMS

## NP-Complete

Problem X is NP-Complete if:

- For all  $Y \in \text{NP}$ ,  $Y \leq_p X$ .
- X is in NP.

$\rightarrow$  if  $P = NP$ , then satisfiability is solved in polynomial time

Theorem 6 cook's theorem.

Suppose  $X \in \text{NP-Complete}$ . Then, X is solvable in polynomial time iff

P = NP, if  $P = NP$ , then there has to be a polynomial time "sol" for the NP-complete prob X.

Proof. ①

$\Leftarrow$ : Suppose  $P = NP$ , then by definition of  $P$ , X can be solved in polynomial time,

# HARDEST NP PROBLEMS

## NP-Complete

Problem X is NP-Complete if:

- For all  $Y \in \text{NP}$ ,  $Y \leq_p X$ .
- $X$  is in NP.

## Theorem 6

*Suppose  $X \in \text{NP-Complete}$ . Then,  $X$  is solvable in polynomial time iff  $P = \text{NP}$ .*

Proof. <sup>(2)</sup> NP-complete problems are NP-Hard.

$\Rightarrow$ : Suppose  $X$  can be solved in polynomial time. Then, by definition of NP-Complete, all problems  $\in \text{NP} \leq_p X$ . Hence, solvable in polynomial time and  $\in P$ .  $(\text{NP} = P)$  □

# FIRST NP-COMPLETE PROBLEM

## Theorem 6

*Cook (1971) – Levin (1973) Theorem [Paraphrase]: Circuit Satisfiability Problem (CSAT) is NP-Complete.*

↳ proving that there exists an NP-complete problem.



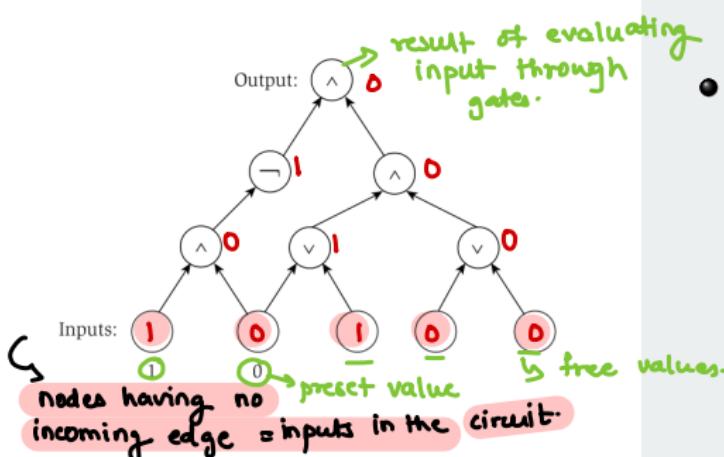
Stephen Cook  
(1968)



Leonid Levin  
(2010)

# CIRCUIT SATISFIABILITY PROBLEM (CSAT)

↳ circuit with logical gates.  
 Circuit is a DAG  
 (directed acyclic graph)



## Problem Definition

- 3 types of gates:  $\wedge$  (AND),  $\vee$  (OR), and  $\neg$  (NOT).
- Circuit  $k$ :
  - A DAG (nodes have 0, 1, or 2 incoming edges).
  - Source: Nodes with no incoming edges; may have a preset binary value.
  - Every other node is labelled with a gate.
  - Output: Result of the node with no outgoing edges.

# FIRST NP-COMPLETE PROBLEM

## Theorem 6

Cook (1971) – Levin (1973) Theorem [Paraphrase]: Circuit Satisfiability Problem (CSAT) is NP-Complete.

Partial Proof.

- NP-complete**
- ↑  
NP-Hard      ↓  
                belongs to NP  
                (complicated)      (easy)
- ① Show that CSAT  $\in$  NP:
- Input size is  $\Omega(|V|)$ .  $\xrightarrow{\text{input = no. of nodes}}$
  - A single gate can be evaluated in constant time.  $\Theta(1)$
  - Evaluate a certificate of the inputs can be verified in  $O(|V|)$  time.

*certificate  $\rightarrow$  assignment of boolean values to those free inputs. Start at bottom, then work your way up.*

# FIRST NP-COMPLETE PROBLEM

## Theorem 6

Cook (1971) – Levin (1973) Theorem [Paraphrase]: Circuit Satisfiability Problem (CSAT) is NP-Complete.

Let's show  $y \in \text{NP-complete}$ .

Partial Proof. ①  $y \in \text{NP}$ .  
 exist a certificate  $\rightarrow$  evaluate it in polynomial time.

① Show that CSAT  $\in \text{NP}$ : ②  $y \in \text{NP-Hard}$ .

$\text{CSAT} \leq_p y$ .

② Reduce every problem  $\in \text{NP}$  to CSAT:

- Consider an arbitrary problem  $X \in \text{NP}$ .
- We need to show  $X \leq_p \text{CSAT}$ .
- By definition for  $X$ :

- $X$  has an input of  $|s|$  bits. + bits for certificate
- Produces 1 bit of output (yes/no).
- $\exists$  an efficient certifier  $B_X(\cdot, \cdot)$ .

$s$ : fixed input / fixed bits.  
 $t$ : map to the free bits.

$X$ : you can verify soln  
in polynomial time

$\downarrow$   
we have an efficient  
certifier for  $X$ .

$B(s, t)$   
 $\downarrow$   
input instance  
 $\downarrow$  certificate.

certifier will be a circuit

# FIRST NP-COMPLETE PROBLEM

## Theorem 6

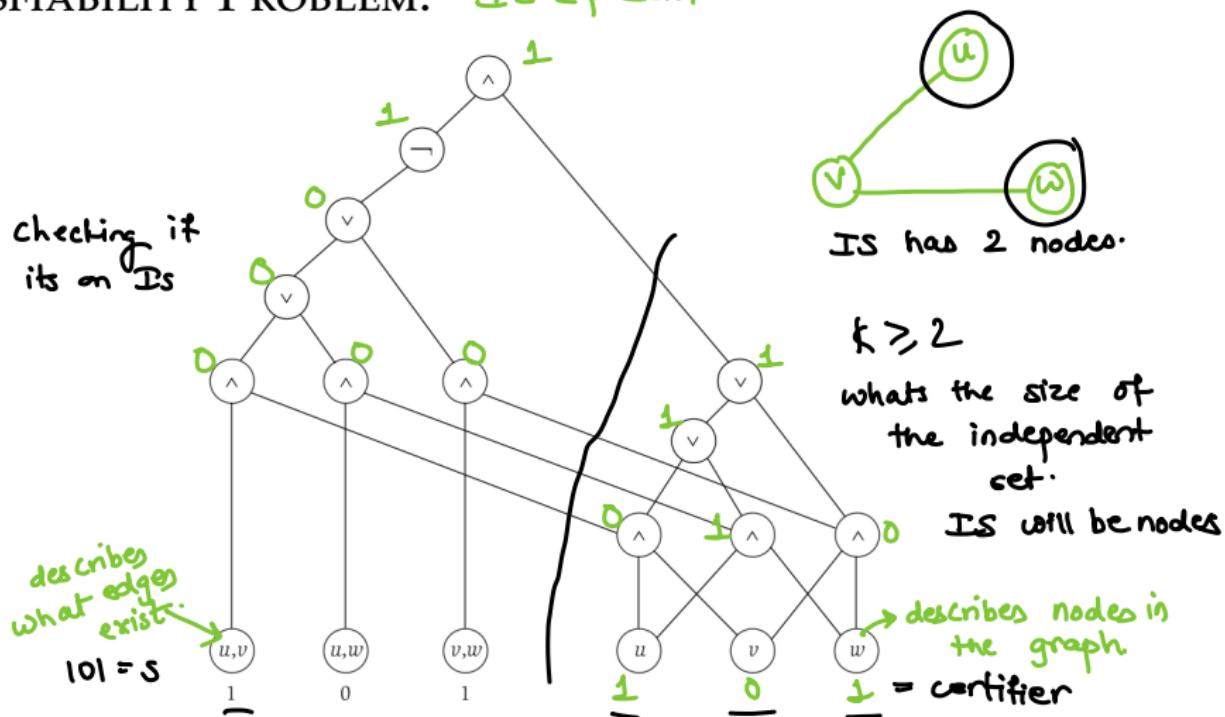
*Cook (1971) – Levin (1973) Theorem [Paraphrase]: Circuit Satisfiability Problem (CSAT) is NP-Complete.*

### Partial Proof.

- ① Show that  $\text{CSAT} \in \text{NP}$ :
- ② Reduce every problem  $\in \text{NP}$  to CSAT:
  - Consider an arbitrary problem  $X \in \text{NP}$ .
  - Reduction to CSAT:  $X$  output 1 : yes instance, no otherwise.
    - Output is 1 when  $X$  is yes; otherwise 0.
    - Sources:  $|s| + |t| = n + \text{poly}(n)$  bits.
    - The first  $n$  bits are hard-coded to the  $X$  instance input.
    - The  $\text{poly}(n)$  bits are free and used to find a  $t$  such that  $B_X(s, t)$  is yes. *certifier returns yes, circuit returns yes as well.*
    - The gates of the circuit are a translation of algorithm  $B_X$ .



# EXAMPLE: INDEPENDENT SET ( $k \geq 2$ ) AS CIRCUIT SATISFIABILITY PROBLEM. $IS \leq_p CSAT$



TH8: Draw the underlying Independent Set graph.

# STRATEGIES FOR PROVING NP-COMPLETENESS

## Showing that Problem X is NP-Complete

Cook Reduction:

- ① Prove that  $X \in \text{NP}$ .
- ② Choose a problem  $Y \in \text{NP-Complete}$ .
- ③ Prove  $Y \leq_p X$ .

$$\text{NP-complete} \rightarrow \text{NP} \quad \left. \begin{array}{c} \\ \end{array} \right\} \text{transitivity}$$

Typical Step 3

$$\text{then } \underline{\text{yes}} \rightarrow \underline{y} \leq_p \underline{x} \rightarrow \text{if yes}$$

- ③ Karp Reduction: For an arbitrary instance  $s_Y$  of  $\underline{Y}$ , show how to construct, in polynomial time, an instance  $s_X$  of  $\underline{X}$  such that  $s_Y$  is a yes iff  $s_X$  is a yes.

Steps:

- ① Provide efficient reduction.
- ② Prove  $\Rightarrow$ : if  $s_Y$  is a yes,  $s_X$  is a yes.
- ③ Prove  $\Leftarrow$ : if  $s_X$  is a yes, then  $s_Y$  had to have been a yes.

# 3SAT is NP-COMPLETE

## Theorem 7

3SAT is NP-Complete.

### Show that Problem 3SAT is NP-Complete

Cook Reduction: *come up with an efficient certificate*

*certificate: clauses : s*

- ① Prove that  $3\text{SAT} \in \text{NP}$ . *t: formula, conjunction of clauses.*
- ② Choose a problem  $Y \in \text{NP-Complete}$ .
- ③ Prove  $Y \leq_p 3\text{SAT}$ .  *$\hookrightarrow \text{CSAT} \leq_p 3\text{SAT}$*

### Proof.

- ① Use a truth assignment of the literals as a certificate. This can be verified in polynomial time.
- ② The only NP-Complete problem we know is CSAT.

# 3SAT IS NP-COMPLETE

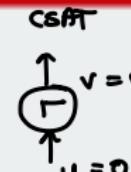
$\text{CSAT} \leq_p \text{3SAT}$  } transitivity

## Proof.

NOT:



$u, w$  has to be a 1.



$$\text{3SAT: } x_v = \overline{x_u}$$

$$x_u$$

$$x_w$$

- ⑥ For an arbitrary circuit k:
  - Each node  $v$  is assigned a variable  $x_v$ .
  - For each gate:
    - NOT: Let  $u$  be the input. We need  $x_u = \overline{x_v}$ .
  $\rightarrow$  2 clauses:  $(x_v \vee x_u) \wedge (\overline{x_v} \vee \overline{x_u})$ .
    - OR: Let  $u, w$  be the inputs. We need  $x_v = x_u \vee x_w$ .
  $\rightarrow$  3 clauses:  $(x_v \vee \overline{x_u}) \wedge (x_v \vee \overline{x_w}) \wedge (\overline{x_v} \vee x_u \vee x_w)$ .
    - AND: Let  $u, w$  be the inputs. We need  $x_v = x_u \wedge x_w$ .
  $\rightarrow$  3 clauses:  $(\overline{x_v} \vee x_u) \wedge (\overline{x_v} \vee x_w) \wedge (x_v \vee \overline{x_u} \vee \overline{x_w})$ .
  - For each constant source  $s$ :  $\rightarrow$  if input = 0, set it to  $\overline{x_s}$ 
 $\rightarrow$  1 clause:  $(x_s)$  if 1, and  $(\overline{x_s})$  if 0.
  - For the output  $o$ : 1 clause  $(x_o)$ . = 1 answer: satisfiability achieved.
  - Convert clauses to length 3:  $\rightarrow$  add  $z_1, z_2$  to clauses too short.
    - We need 2 variables  $z_1$  and  $z_2$  that are always 0 in a satisfying assignment.
  $(x_o \vee z_1 \vee z_2) = \text{output}$ .
    - To ensure this, we need 4 variables:  $z_1, z_2, z_3, z_4$ .

# 3SAT is NP-COMPLETE

~~Imp~~ → Reduction proof

Proof.

- ③  $s_{\text{CSAT}}$  is a yes iff  $s_{\text{3SAT}}$  is a yes:

- $\Rightarrow$ : If  $s_{\text{CSAT}}$  is a yes, then the satisfying assignment to the circuit inputs can be used to calculate the value of each gate. By the reduction, these values will satisfy all the clauses of  $s_{\text{3SAT}}$ .
- $\Leftarrow$ : If  $s_{\text{3SAT}}$  is a yes, then the assignment of the variables give the satisfying assignment of the circuit inputs, and the reduction guarantees that the assigned values for the nodes match the gate calculations.



# 3SAT is NP-COMPLETE

From our previous reductions

and

$$\left. \begin{array}{c} 3\text{SAT} \leq_p \text{IS} \leq_p \text{VC} \leq_p \text{SC} \\ \text{NP-complete} \end{array} \right\} \text{transitivity.}$$
$$3\text{SAT} \leq_p \text{IS} \leq_p \text{SP}$$

and the fact that 3SAT is NP-Complete:

## Corollary 7

*The following problems are NP-Complete:*

3SAT, IS, VC, SC, SP.

# TAXONOMY OF NP-COMPLETENESS

---

---

# SEQUENCING PROBLEMS

## Travelling Salesperson Problem (TSP)

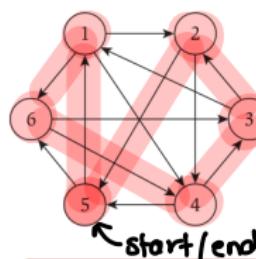
- A salesperson must visit n cities  $v_1, v_2, \dots, v_n$ .
- Starting at some  $v_1$ , visit all cities and return to  $v_1$ .
- Distance function:  $d(\cdot, \cdot)$  for all pairs of cities (not necessarily symmetric nor metric).
- Optimization: What is the shortest tour?

you visit all  
the nodes only  
once using distinct  
paths.

# SEQUENCING PROBLEMS

## Travelling Salesperson Problem (TSP)

- A salesperson must visit  $n$  cities  $v_1, v_2, \dots, v_n$ .
- Starting at some  $v_1$ , visit all cities and return to  $v_1$ .
- Distance function:  $d(\cdot, \cdot)$  for all pairs of cities (not necessarily symmetric nor metric).



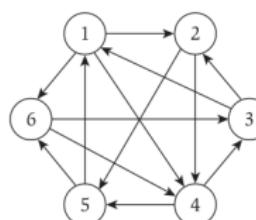
## Hamiltonian Cycle

- Graph analogue of TSP.
- *Hamiltonian cycle*: a tour of the nodes of  $G$  that visits each node once.
- Given a digraph G, does it contain a Hamiltonian cycle? **yes**.

# SEQUENCING PROBLEMS

## Travelling Salesperson Problem (TSP)

- A salesperson must visit  $n$  cities  $v_1, v_2, \dots, v_n$ .
- Starting at some  $v_1$ , visit all cities and return to  $v_1$ .
- Distance function:  $d(\cdot, \cdot)$  for all pairs of cities (not necessarily symmetric nor metric).



TH9: Does this graph contain a Hamiltonian cycle? yes

## Hamiltonian Cycle

- Graph analogue of TSP.
- *Hamiltonian cycle*: a tour of the nodes of  $G$  that visits each node once.
- Given a digraph  $G$ , does it contain a Hamiltonian cycle?

# $3\text{SAT} \leq_p \text{Hamiltonian}$

Theorem 8

you need to prove this theorem.

NP

NP-Hard

Hamiltonian Cycle is NP-complete.

Proof.

$t$ : certificate  
 $B(s, t)$

- ① In NP: A certificate would be a sequence of vertices which can be verified in polynomial time.
- ② Choose an NP-complete problem:  $3\text{SAT}$ .

you need to reduce  $3\text{SAT}$  to Hamiltonian

$3\text{SAT} \leq_p \text{Hamiltonian}$

this will prove Hamiltonian as NP-Hard.

# $3\text{SAT} \leq_p \text{Hamiltonian}$

## Theorem 8

Hamiltonian Cycle is NP-complete.

$3\text{SAT}$ : clauses = 3 boolean variables  
in each

Proof. (could be negated/not)

to satisfy formula, ③  $3\text{SAT} \leq_p \text{Hamiltonian}$ :  
all clauses should be satisfied.  
assignment of 0's and 1's to literals

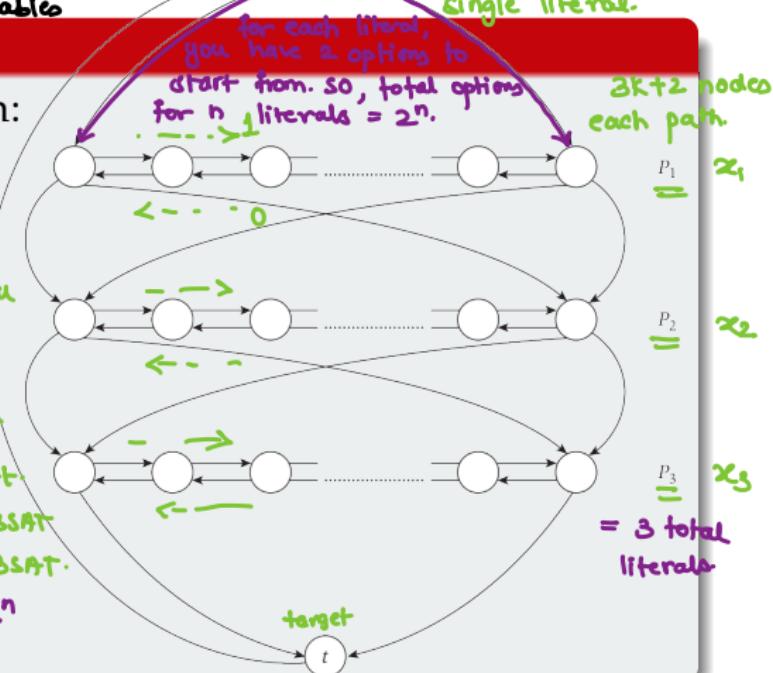
- $P_i$  (containing  $3k + 2$  nodes)  
for each  $X_i$ : left traversal for 1 and right traversal for 0.
- you get a Hamiltonian cycle if you completely traverse all paths from left to right or right to left.

$\text{left} \rightarrow \text{right} = \text{value of 1 for 3SAT}$   
 $\text{right} \rightarrow \text{left} = \text{value of 0 for 3SAT}$

from the base graph, we get  $2^n$  possible Hamiltonian cycles.

Base graph captures all possible solutions for

edge going from  $t \rightarrow s$  source  
create a path for every single literal.



# 3SAT $\leq_p$ Hamiltonian

$$c_1 : \alpha_1 \vee \overline{\alpha_2} \vee \alpha_3$$

## Theorem 8

Hamiltonian Cycle is NP-complete.

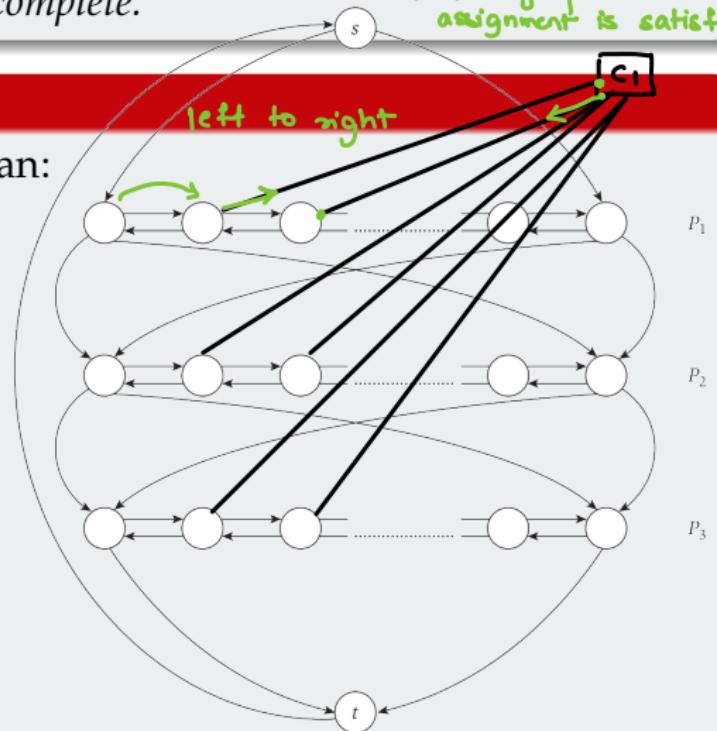
## Proof.

- ③ 3SAT  $\leq_p$  Hamiltonian:

$c_i$  for each clause is connect based on  $\alpha_i$  or  $\overline{\alpha_i}$

$\alpha_i$  = left to right traversal  
 $\overline{\alpha_i}$  = right to left traversal.

you visit the clause node after every node in the graph if the assignment is satisfiable.



# 3SAT $\leq_p$ Hamiltonian

## Theorem 8

Hamiltonian Cycle is NP-complete.

$3k+2$ : polynomial no. of nodes per literal.

Proof.

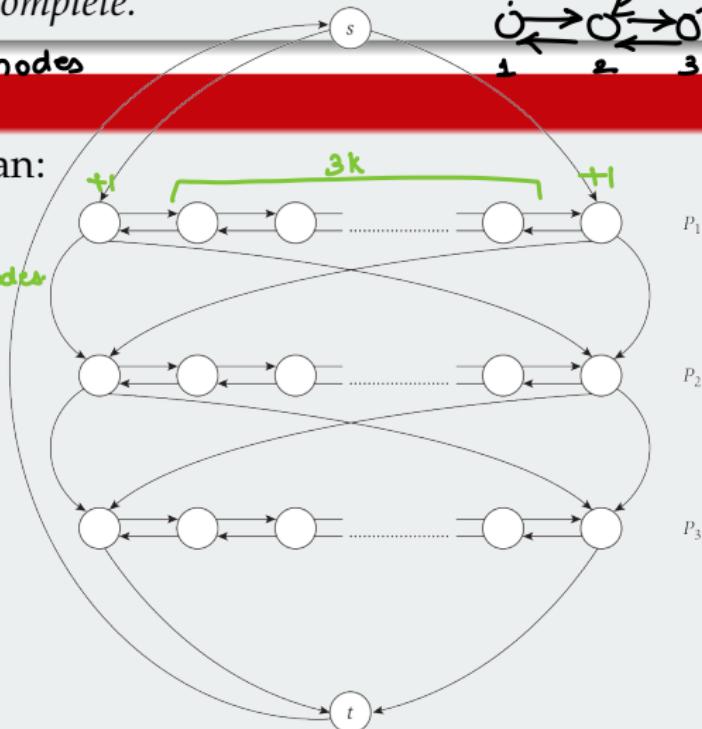
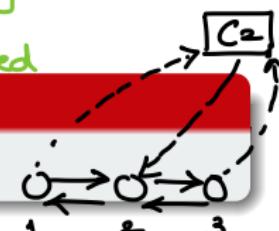
### ③ 3SAT $\leq_p$ Hamiltonian:

+2 nodes: beginning and ending nodes

- $P_i$  (containing  $(3k + 2)$  nodes) for each  $X_i$ : left traversal for 1 and right traversal for 0.

for every clause, we have 3 nodes that we can use for each literal.

$$C_2: \begin{array}{c} x_1 \vee \bar{x}_1 \vee x_2 \\ = = \\ \text{always satisfied} \end{array}$$



To prove NP-Hard, you need to prove the if and only if part: (write this in exam).

$S_{3SAT}$  is yes-instance if  $S_{Hamiltonian}$  is a yes-instance:

→ If  $3SAT$  is a yes, then each clause node can be visited from one of the paths corresponding to one of the variables when the path is traversed in the direction of the satisfying assignment.

( $3SAT$  is a yes if and only if the Hamiltonian instance that is generated by reduction is also a yes).

Start: yes from  $3SAT$ .

then work through reduction → has to produce a yes instance.

My reduction gave me a yes-instance, this only comes from a yes-instance in the  $3SAT$ .

← If the hamiltonian instance is a yes, then every clause node is visited, and the direction of each path traversal gives a value assignment for the corresponding variable in  $3SAT$ . The reduction guarantees that value assignment for a variable the path used to traverse the clause node will be the assignment of a variable that satisfies the corresponding clause.

with a hamiltonian cycle in  $G$ , the shortest tour will be of length  $n$ .

## TRAVELLING SALESPERSON

### Theorem 9

NP

NP-Hard.

Travelling Salesperson (TSP) is NP-complete.

→ you need to show this : Hamiltonian  $\leq_p$  TSP.

### Proof.

- ① In NP: Certificate that is a tour of the cities. (check in polynomial time).

- ② Use Hamiltonian Cycle.

- ③ Hamiltonian Cycle  $\leq_p$  TSP:

Given a graph  $G = (V, E)$ :

- For each  $v$ , make a city.
- For each edge  $(u, v) \in E$ , define  $d(u, v) = \underline{1}$ .
- For each pair  $(u, v) \notin E$ , define  $d(u, v) = \underline{2}$ .
- Set the tour bound to be  $n$ .

tour distance = n.  
(no. of nodes)

sequence of nodes

- take every pair in the tour.
- add up the distance by calling the distance function.
- the total distance should be what you predicted.

$$\underbrace{1+1+\dots+1}_{n \text{ times}} = n$$

if its more than n then its not hamiltonian.

# TRAVELLING SALESPERSON

~~NP-hard: If shortest tour is length  $n$ , then no  $d(u,v) = 2$  is used, so only edges from the graph are used implying a Hamiltonian cycle in  $G$ .~~

## Theorem 9

*Travelling Salesperson (TSP) is NP-complete.*

### Proof.

- ① In NP: Certificate that is a tour of the cities.
- ② Use Hamiltonian Cycle.
- ③ Hamiltonian Cycle  $\leq_p$  TSP:

Given a graph  $G = (V, E)$ :

- For each  $v$ , make a city.
- For each edge  $(u, v) \in E$ , define  $d(u, v) = 1$ .
- For each pair  $(u, v) \notin E$ , define  $d(u, v) = 2$ .
- Set the tour bound to be  $n$ .

# EXERCISE: SHOW THAT HAMILTONIAN PATH IS

NP-COMPLETE

- simple path in a digraph  $G$  that contains all nodes.
- another sequencing problem (is there a path that contains all nodes?).

## Theorem 10

*Hamiltonian Path is NP-complete*

### Proof.

- ① In NP: Certificate is a path in  $G$  which can be verified in polynomial time.
- ② NP-complete problem: Hamiltonian Cycle.

Hamiltonian cycle  $\leq_p$  Hamiltonian path.

sequence of nodes  $\rightarrow$  you trace this sequence  $\rightarrow$  is there always an edge that follows this sequence?  
and you only visit each one once (verify this in polynomial time).

# EXERCISE: SHOW THAT HAMILTONIAN PATH IS NP-COMPLETE

Theorem 10

*Hamiltonian Path is NP-complete*



① If in  $G$ , we have a Hamiltonian cycle, after splitting the nodes, we know we will have a hamiltonian path.

② If we have a hamiltonian path in  $G'$ , that means we can find a simple path from  $v'$  to  $v''$ ,  $v'$  and  $v''$  are the same node in  $G$ , this means we have a cycle.

Proof.

③ Hamiltonian Cycle  $\leq_p$  Hamiltonian Path:

For  $G = (V, E)$  create  $G'$ :

- Choose an arbitrary  $v \in V$ :  $V' = V \setminus \{v\} \cup \{v', v''\}$ . split on arbitrary node.
- Initialize  $E' = E$ :
  - For each edge  $(v, w) \in E$ :  $E' \setminus \{(v, w)\} \cup \{(v', w)\}$ .
  - For each edge  $(u, v) \in E$ :  $E' \setminus \{(u, v)\} \cup \{(u, v'')\}$ .
- A path  $v' \rightarrow v''$  means Hamiltonian Cycle.



# PARTITIONING PROBLEMS

↳ taxonomy problems (matching problems)

take input, partition them into sets with diff constraints.

→ 3D version of Bipartite matching:

## 3-D Matching

- Given 3 disjoint sets:  $X, Y, Z$  (each of size  $n$ ).
- A set of  $m \geq n$  trebles  $T \subseteq X \times Y \times Z$ . ~~treble T consist of something from X, Y, Z.~~
- Does there exist a set of  $n$  trebles from  $T$  so that each item is in exactly one of these trebles?

is there a subset in the  $m$  trebles of size  $n$  such that if you take the union of these trebles, you end up with the entire universe.  
 (you collect every element of  $X, Y, Z$ ).

① NP: certificate is a set of triples which can be verified in polynomial time.  
 count no. of triples  $\rightarrow$  go through and see if anything is duplicated.

## 3-D MATCHING IS NP-COMPLETE

Theorem 11  $3SAT \leq_p 3D\text{-matching}$

3-D Matching is NP-Complete.

each triangle = one triple.  
 $k = \text{no. of clauses.}$

$\rightarrow$  you do a gadget per variable, for every  $x_i \rightarrow$  we create a gadget.  
 Proof. basically we define elements and triples

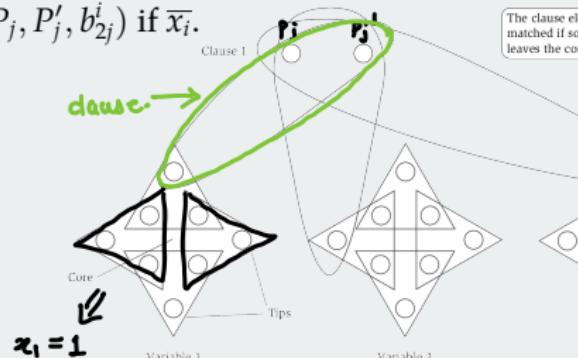
②  $3SAT \leq_p 3\text{-D Matching:}$

Consider an arbitrary 3SAT:

- Clause  $C_j$  gadget:

$C_1: x_1 \vee \bar{x}_2 \vee x_3$  • Add  $P_j = \{p_j, p'_j\}$  with trebles:  $(p_j, p'_j, b_{2j-1}^i)$  if  $x_i$  and  $(P_j, P'_j, b_{2j}^i)$  if  $\bar{x}_i$ .

The clause elements can only be matched if some variable gadget leaves the corresponding tip free.



$\rightarrow$  variable  $x_i$  gadget.  
 $\rightarrow$  core:  
 $A_i = \{a_1^i, \dots, a_{2k}^i\}$   
 $\rightarrow$  tips:  
 $B_i = \{b_1^i, \dots, b_{2k}^i\}$ .  
 $\rightarrow t_j^i = (a_j^i, a_{j+1}^i, b_j^i)$   
 $\text{for } j = 1, 2, \dots, 2k$   
 $(\text{add mod } 2k)$ .

# 3-D MATCHING IS NP-COMPLETE

Theorem 11

$p_j$ : clause gadget     $q_i$ : clean-up gadget.

3-D Matching is NP-Complete.

$$x = \underbrace{\{a_j^i \text{ even}\}}_{\text{even cores}} \cup \{p_j\} \cup \{q_i\}, \quad y = \underbrace{\{a_j^i \text{ odd}\}}_{\text{odd cores}} \cup \{p_j'\} \cup \{q_i'\}$$

Proof.

③ 3SAT  $\leq_p$  3-D Matching:  $z = \{b_j\}$  all the tips.

Consider an arbitrary 3SAT: each tip corresponds to two cores.

- Counting cores: covered by even/odd choice. if you select 2 tips, you cover all cores.
- Counting tips: n variables, each have 2k tips.

- Even/odd tips cover nk.
- Clauses cover k.  $\rightarrow$  all clauses will cover k tips.
- $(n - 1)k$  uncovered.
- $(n - 1)k$  clean-up gadgets:

- $Q_i = \{q_i, q'_i\}$  with treble  $(q_i, q'_i, b)$  for every tip  $b$ .

$\rightarrow$  we need to cover all the uncovered tips.

every  $a_j^i$  gets connected to all  $(n - 1)k$  uncovered tips (they form a treble).

# 3-D MATCHING IS NP-COMPLETE

## Theorem 11

*3-D Matching is NP-Complete.*

### Proof.

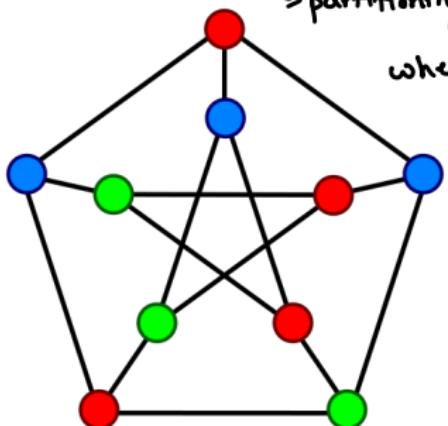
*→ this step happens after reduction. (if and only if proofs).*

- ③  $\text{3SAT} \leq_p \text{3-D Matching}$ :

Consider an arbitrary 3SAT:

- $\Rightarrow$  For a yes 3SAT, there is a matching that takes the even/odd tip trebles, leaving at least one tip as part of each clause gadget treble. The remaining unmatched tips are matched to a clean-up gadget.
- $\Leftarrow$  A yes for 3-D Matching from the reduction means that each clause gadget is part of a selected treble, each variable gadget has selected the odd or even tips, and the remaining tips are matched to a clean-up gadget. Each clause will be satisfied by the tip matched by the clause gadget. The even/odd selection for each variable guarantees all variables are assigned 1 or 0.

# GRAPH COLOURING



## k-Colour

- Colouring of the nodes of a graph such that no adjacent nodes have the same colour, using at most k colours.
- Labelling (partitioning) function  $f : \underline{V} \rightarrow \{1, \dots, k\}$  such that, for every  $(\underline{u}, \underline{v}) \in E, f(\underline{u}) \neq f(\underline{v})$ .

partitioning function - assignment from every node to one of these k partitions.

↳ partitioning problem.

when  $k = 2$ , you get a bipartite graph.

## Problem

Given a graph  $G$  and a bound k, does  $G$  have a k-colouring?

↳ assignment of nodes to some color.  
you can use at most k colors.

no adjacent nodes can have the same color.

# 3-COLOURING IS NP-COMPLETE

→ prove this:

## Theorem 12

3-Colouring is NP-Complete.

NP

NP-Hard.

Proof. (Certificate) (reduction + if and only if)

- ① In NP: Certificate is a colouring of the nodes which can be verified in polynomial time. each color gets one set and no two nodes in the same set should share an edge.
- ② NP-complete problem: 3SAT.

$3SAT \leq_p 3\text{-coloring}$  partitioning problem.

→ more optimal!

3D-matching  $\leq_p$  3-coloring

# 3-COLOURING IS NP-COMPLETE

\* you need to make sure  $v_i$  and  $\bar{v}_i$  have different colors, so you make them adjacent.

\* B - base color to color our variables two edges from  $v_i \rightarrow B$ ,  $\bar{v}_i \rightarrow B$ .

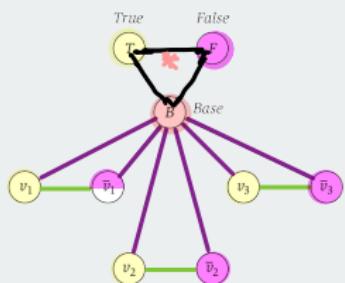
$v_i, \bar{v}_i$  cannot be colored by the base color.

## Theorem 12

3-Colouring is NP-Complete.

## Proof.

### ③ 3SAT $\leq_p$ 3 Colouring:



No. of valid 3-colorings:  $2^n$

$$= 2^3 = 8$$

$$x_i : v_i \text{ and } \bar{v}_i$$

- For each literal: Nodes  $v_i$  and  $\bar{v}_i$ .
- Nodes T (true), F (false), and B (base).
- Edges:  $(v_i, \bar{v}_i), (v_i, B), (\bar{v}_i, B)$ .

- Edges:  $(T, F), (F, B), (T, B)$ . \* form a  $\Delta$  so that each of them get diff colors

$2^n$  binary strings from 3SAT  $\rightarrow 2^n$  possible colorings that could work.

# 3-COLOURING IS NP-COMPLETE

## Theorem 12

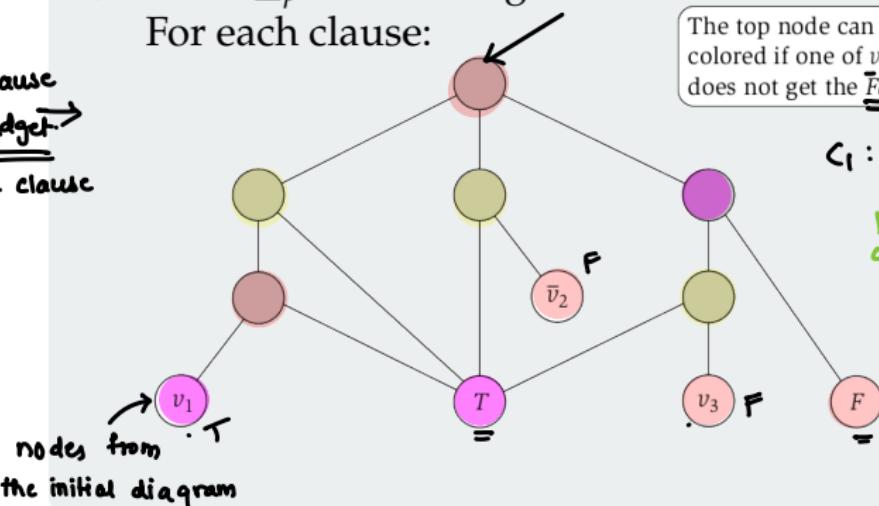
3-Colouring is NP-Complete.

→ Structure has been created such that the only way we can come up with a valid coloring is if one of the  $v_i$  nodes has been coloured True  $\Rightarrow$  at least one literal has to be true for our clause to get satisfied.

⑥ 3SAT  $\leq_p$  3 Colouring: new nodes added in the graph.

For each clause:

clause  
gadget  
 $\xrightarrow{=}$   
1 clause



$$C_1: \underline{x_1} \vee \underline{\bar{x}_2} \vee x_3$$

purple color = true  
clause satisfied, so

3-coloring should have a valid coloring.

# NUMERICAL PROBLEMS

↳ NP-complete problem  $\Rightarrow$  input is numbers and you want to do some sort of aggregation to the numbers.

## Subset Sum Problem

Given a set of  $\underline{n}$  natural numbers  $\{w_1, \dots, w_n\}$  and a target  $\underline{W}$ , is there a subset of the numbers that  $\underline{\text{add up to}} \underline{W}$ ?

↳ subset problem  $\rightarrow$  special case of Knapsack prob. (Dynamic programming).

## Dynamic Programming Approach

- We saw an  $O(\underline{n} \underline{W})$  algorithm.
- Pseudo-polynomial:  $W$  is unbounded, e.g.,  $\underline{2^n}$ .

# SUBSET SUM IS NP-COMPLETE

→ Since it is a special case of Knapsack problem, Knapsack is also NP-complete.

## Theorem 13

*Subset Sum is NP-Complete.*

Proof. Certificate would be a subset of the input numbers.  
→ take items in the subset, add them up and compare to  $W$ .

- ① In NP: Certificate is a subset of the numbers which can be verified in polynomial time.
- ② NP-complete problem: 3-D Matching.

# SUBSET SUM IS NP-COMPLETE

## Theorem 13

*Subset Sum is NP-Complete.*

## Proof.

- ➊ 3-D Matching  $\leq_p$  Subset Sum: Exercise: Try it, but tough.

# SUBSET SUM IS NP-COMPLETE

3D matching: Are there  $m$  subsets that contain the entire universe?

## Theorem 13

Subset Sum is NP-Complete.

each subset:  $\langle 001|001|101 \rangle$

Proof.

1 in the set

0 not in the set

total  $3n$ .



### ⑥ 3-D Matching $\leq_p$ Subset Sum:

we need to encode a treble.

$i \in X$

$j \in Y$

$k \in Z$

convert

- 3-D Matching: Subsets can be viewed as length  $3n$  bit vectors with a 1 indicating that item is in the set.

- For each treble  $(i, j, k)$  from  $X \times Y \times Z$  construct a  $w_t$ :

- A digits with 1 at  $i, n + j$ , and  $2n + k$ .
- For base  $d$ ,  $w_t = d^{i-1} + d^{n+j-1} + d^{2n+k-1}$ .
- Set base  $d = m + 1$  to avoid addition carry overs.

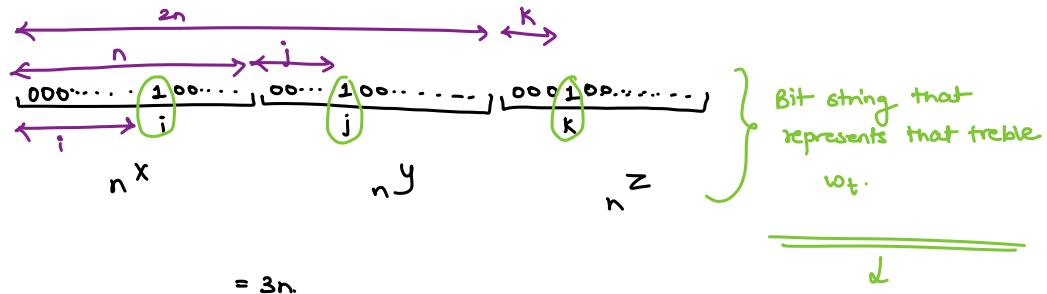
- Set  $W = \sum_0^{3n-1} (m + 1)^i$  which corresponds to have each item exactly once.

$$W = \sum_0^{3n-1} d^i$$

find  $W$  value where every single digit is 1.

□

A treble will look something like this:



A digit at index  $\underline{i}$  then  $\underline{(n+j)}$  then  $\underline{(2n+k)}$

then you convert this into your own number.

Eg:  $\frac{625}{321} : 6 \times 10^2 + 2 \times 10^1 + 5 \times 10^0$

similarly:  $d^{i-1} + d^{n+j-1} + d^{2n+k-1}$

base  $d = m+1$   $m = \text{total no. of trebles}$ .

if we have  $m$  trebles, for a given item  $i$ , it can appear  $\leq$  total times.

so, if you take your base as  $(m+1)$ , you don't get any carryovers.

Base 3:  $\begin{array}{r} 1 \\ + 1 \\ \hline 2 \end{array}$

tells us we have collected  $i$  twice. } not a proper 60<sup>th</sup> matching but

then you take this treble and interpret it as a digit (a number).

when we look at  $(w.t)$  after any sum, the value that we see at that digit = no. of elements we collected so far.

if and only if proof:

→ all of our trebles become exact numbers.

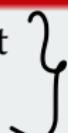
for Base 2:  $\begin{array}{r} 10 \\ \hline 2 \end{array}$  } not 2.

Because we went up to the value of the base, we need another digit to represent it. So that creates a carryover.

# TAXONOMY OF HARD PROBLEMS

Packing Problems → collect as many elements as you can from the input, subject to some sort of constraint.

- Independent Set
- Set Packing



→ generalized version of IS.

Covering Problems

- Vertex Cover
- Set Cover

→ collect elements in the input, but you want to collect the least amount of elements.

→ generalized version of VC.

Sequencing Problems

- TSP (Travelling Salesperson problem)
- Hamiltonian Cycle
- Hamiltonian Path

→ elements from input, come up with the right order on that input.  
(order input according to some sort of constraints).

# TAXONOMY OF HARD PROBLEMS

partition elements from input into diff sets, again according to some constraints.

## Partitioning Problems

- 3-D Matching
- Graph Colouring

problems dealing with numbers, aggregate numbers from the input somehow.

## Numerical Problems

- Subset Sum
- Knapsack

→ 3SAT, CSAT.

## Constraint Satisfaction Problems

- 3SAT

# Complexity theory

coNP

compliment of NP.

## ASYMMETRY OF NP

if we want to show yes instance, we only have to find one single certificate for which  $B(s,t)$  returns yes.

### Efficient Certifier Asymmetry

To show no instance, we need to show that for all certificates that exist, our

Given an instance  $s$  of problem  $X$ :  $B(s,t)$  returns no.

- thus there is on asymmetry*
- For any  $t$ ,  $B(s,t) = \text{yes}$  implies yes-instance. E
  - For all  $t$ ,  $B(s,t) = \text{no}$  implies no-instance. A

$s \notin X$ , then problem will be a no for its compliment.

## Complimentary Problem

For every problem  $X$ , there is a complementary problem  $\bar{X}$ :

- For all input  $s$ ,  $s \in X$  iff  $s \notin \bar{X}$ .
- Note that, if  $X \in P$ , then  $\bar{X} \in P$ .

there is a polynomial algo that solves  $X$ .

we solve  $X$ , and return the answer's compliment.

# coNP

## Complexity Class coNP

A problem  $X \in \text{coNP}$  iff  $\overline{X} \in \text{NP}$ .

## Open Question

Does  $\underline{\text{NP}} = \underline{\text{coNP}}$ ?

Theorem 14 *→ to prove this:*

If  $\text{NP} \neq \text{coNP}$ , then  $\underline{\text{P}} \neq \underline{\text{NP}}$ .

Proof.  $\neg q \rightarrow \neg p$ .

Contra-positive: Assume  $\underline{\text{P}} = \underline{\text{NP}}$ :

based on the definition  
of coNP class.

- $X \in \underline{\text{NP}} \xrightarrow{\text{assumption.}} X \in \underline{\text{P}} \rightarrow \overline{X} \in \underline{\text{P}} \xrightarrow{\quad} \overline{X} \in \underline{\text{NP}} \rightarrow X \in \underline{\text{coNP}}$ .
- $X \in \underline{\text{coNP}} \rightarrow \overline{X} \in \underline{\text{NP}} \rightarrow \overline{X} \in \underline{\text{P}} \rightarrow X \in \underline{\text{P}} \rightarrow X \in \underline{\text{NP}}$ .



# PSPACE

$\underset{\curvearrowleft}{\equiv}$  polynomial space (does not restrict time at all).  
(class of all problems that can be solved using polynomial space)

# BEYOND TIME

## Complexity Class PSPACE

Set of all problems that can be solved using polynomial space.

Theorem 15  $\rightarrow P$  is a subset of PSPACE.

$P \subseteq \text{PSPACE}$  (polynomial time restricts you from using polynomial space.)

$\hookrightarrow_p$  can be solved using polynomial time, you can't use more than polynomial space.

Theorem 16 certificate = bit string polynomial in size.

$NP \subseteq \text{PSPACE}$  we use polynomial space to enumerate through all possible certificates.

Proof. (prove by example: 3SAT in this case.)

we have an NP certifier.

- For 3SAT, a bit vector can encode an assignment.
- We can try all bit vectors with one  $n$ -length vector in memory:
  - Start with 0 until  $2^n - 1$ , adding 1 at each iteration.
- Since  $3SAT \in \text{PSPACE}$  and is NP-complete, for any  $Y \in \text{NP}$ ,  $Y \leq_p 3SAT$  and solve in PSPACE.



# PROTOTYPICAL PSPACE PROBLEM

Let  $\Phi(x_1, \dots, x_n)$  be a conjunction of  $k$  disjunctions of  $n$  variables (like SAT).  
*representing some SAT formula.*

## Quantified SAT

- $\exists x_1 \forall x_2 \exists x_3 \dots Q_n x_n \Phi(x_1, \dots, x_n)$  (Prenex normal form).
- Contingency planning.

① QSAT needs to be in PSPACE.

Theorem 17 ② QSAT needs to be PSPACE hard.

QSAT is PSPACE-complete.

# CS 577 - Randomized Algorithms

↳ speed up algos or get correct answers.

Marc Renault

Department of Computer Sciences  
University of Wisconsin – Madison

Fall 2023

TopHat Section 001 Join Code: 477366

TopHat Section 002 Join Code: 560750



**WISCONSIN**  
UNIVERSITY OF WISCONSIN-MADISON

# RECALL: LINEAR TIME SELECTION

Problem

↳ MOM select: linear time.

Find the  $k$ th value in an unsorted array  $A$  of  $n$  numbers if  $A$  were sorted.  $O(n \log n)$  time: sort  $A$  and go to the  $k$ th index.

Algorithm: QUICKSELECT

Input : A array  $A[1..n]$  and an int  $k$ .

Output: The  $k$ th element of  $A$  if  $A$  were sorted.

if  $n = 1$  then return  $A[1]$

Choose a pivot  $A[p]$  *↳ Basically sorting the pivot.*

$r := \text{PARTITION}(A[1..n], p)$

if  $k < r$  then

    | return  $\text{QUICKSELECT}(A[1..r - 1], k)$

else if  $k > r$  then

    | return  $\text{QUICKSELECT}(A[r + 1..n], k - r)$

else

    | return  $A[r]$

end

# QUICKSORT

---

## Algorithm: QUICKSORT

---

**Input :** An array  $A[1..n]$ .

**Output:**  $A$  sorted from 1 to  $n$ .

Choose a pivot  $\underline{\underline{A[p]}}$

$r := \text{PARTITION}(A[1..n], p)$

→ pivot ends up in the sorted position  
in the array.

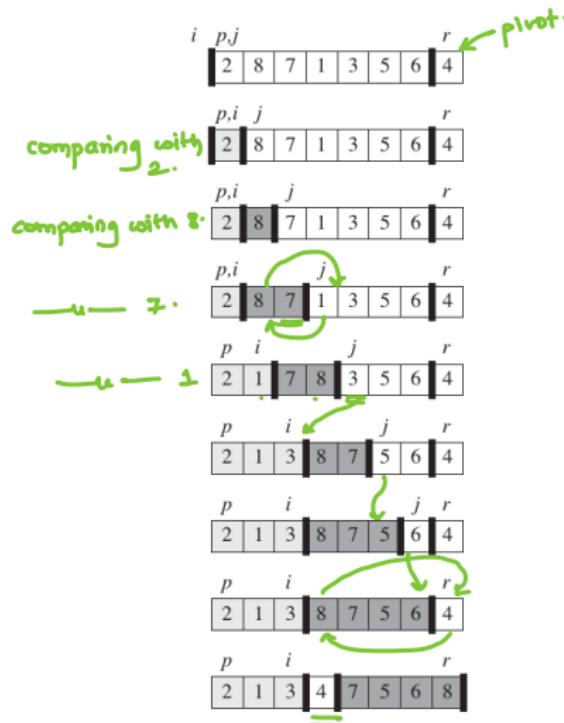
$\xrightarrow{\text{smaller}} \text{QUICKSORT}(A[1..r - 1])$  } recursively call both  
 $\xrightarrow{\text{larger}} \text{QUICKSORT}(A[r + 1..n])$  sides of the array.

**return  $A$**

---

# QUICKSORT

QUICKSORT partition step: linear time.  $O(n)$ .



# QUICKSORT

---

## Algorithm: QUICKSORT

---

**Input :** An array  $A[1..n]$ .

**Output:**  $A$  sorted from 1 to  $n$ .

Choose a pivot  $A[p]$

$r := \text{PARTITION}(A[1..n], p)$

$\text{QUICKSORT}(A[1..r - 1])$

$\text{QUICKSORT}(A[r + 1..n])$

**return**  $A$

---

→ we aren't dividing the array like in mergesort.

Why no combine step?

Because QUICKSORT sorts in-place.

↳ your array gets modified.

# QUICKSORT ANALYSIS

*WORST CASE* ↗ pivot = largest / smallest of all the elements of the array.

## Algorithm: Quicksort

**Input :** An array  $A[1..n]$ .

**Output:**  $A$  sorted from 1 to  $n$ .

Choose a pivot  $A[p]$   $\text{TC}$

$r := \text{PARTITION}(A[1..n], p) - o(n)$

$\text{QUICKSORT}(A[1..r - 1])$  ↗ one of these two  
 $\text{QUICKSORT}(A[r + 1..n])$  will have  $(n-1)$  elements, other will have zero.

**return**  $A$

## Worst-case recurrence

$$\begin{aligned}
 T(n) &\leq T(\underline{n-1}) + T(\underline{0}) + O(n) \\
 &\leq T(n-2) + 2T(0) + 2O(n) \\
 &\leq n(T(0) + O(n)) \\
 &= O(\underline{n^2})
 \end{aligned}$$

} unwind method.

# QUICKSORT ANALYSIS

## BEST CASE

---

### Algorithm: QUICKSORT

---

**Input :** An array  $A[1..n]$ .

**Output:**  $A$  sorted from 1 to  $n$ .

Choose a pivot  $A[p]$

$r := \text{PARTITION}(A[1..n], p)$

$\text{QUICKSORT}(A[1..r - 1])$

$\text{QUICKSORT}(A[r + 1..n])$

**return**  $A$

---

divided exactly in half.

Best-case recurrence

$$\begin{aligned} T(n) &\leq 2T(n/2) + O(n) \\ &= O(\underline{n \log n}) \end{aligned}$$

# QUICKSORT ANALYSIS

## AVERAGE CASE

### Observation 1

For  $0 < \varepsilon < 1$ ,  $\varepsilon$ : splits it into two.  
 ↗ multiplicative factor.

problem is  
 $\varepsilon = 0, 1$ .

$$\begin{aligned} T(n) &= T(\varepsilon n) + T((1 - \varepsilon)n) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

### Probabilistic Argument

#### Expected Runtime:

$$\begin{aligned} T(n) &\leq \overbrace{\Pr[\Theta(n) \text{ split}] \cdot \Theta(n \log n)}^{\text{good split}} + \overbrace{\Pr[o(n) \text{ split}] \cdot \Theta(n^2)}^{\text{bad split}} \\ &= (1 - \Pr[o(n) \text{ split}]) \cdot \Theta(n \log n) + \Pr[o(n) \text{ split}] \cdot \Theta(n^2) \\ &= \Theta(n \log n), \text{ if } \Pr[o(n) \text{ split}] = O\left(\frac{\log n}{n}\right) \end{aligned}$$

↗ chances of us getting a bad split.

# QUICKSORT ANALYSIS

## AVERAGE CASE

- values in our sum repeat twice, so you choose one value and multiply by 2.

### Average Case Recurrence (uniform dist on orderings)

you get total runtime when you multiply by probability, then you sum over all i's.

$$T(n) \leq \frac{1}{n} \sum_{i=1}^n (T(i-1) + T(n-i)) + O(n)$$

you can choose the pivot from any one of them.

probability of choosing a pivot =  $\frac{1}{n}$

Runtime given that I have grabbed the  $i^{th}$  position.

$$= \frac{2}{n} \sum_{i=1}^n (T(i-1)) + O(n) = O(n \log n)$$

$$i=1 \quad T(0) + T(n-1)$$

$$i=n \quad T(n-1) + T(0)$$

### Uniform Assumption Realistic?

- Probably not...

$\rightarrow$  MOM pivot.

- Improve Quicksort by more complicated pivot choice.

VAR - uniformly at random.

$\rightarrow$  make your quicksort random, choose pivot uniformly at random

$$T(n) \leq \frac{1}{n} \sum_{i=1}^n (T(i-1) + T(n-i)) + O(n)$$

$$T(n) = \frac{2}{n} \sum_{i=1}^n (T(i-1)) + O(n)$$

$$n T(n) = 2 \sum_{i=1}^n T(i-1) + cn^2.$$

$$-(n-1) T(n-1) = 2 \sum_{i=1}^{n-1} T(i-1) + c(n-1)^2$$

} subtract  
these two

$$n T(n) - (n-1) T(n-1) = \left( 2 \sum_{i=1}^n T(i-1) - 2 \sum_{i=1}^{n-1} T(i-1) \right) + (cn^2 - c(n-1)^2)$$

$$n T(n) - (n-1) T(n-1) = 2 \left( (T(0) + T(1) + T(2) + \dots + T(n-1)) - (T(0) + T(1) + T(2) + \dots + T(n-2)) \right) + (c(n^2 - (n-1)^2))$$

$$n T(n) - (n-1) T(n-1) = 2 (T(n-1)) + c(n^2 - (n^2 - 2n + 1))$$

$$n T(n) - (n-1) T(n-1) = 2 (T(n-1)) + c(2n-1)$$

$$n T(n) = 2 T(n-1) + (n-1) T(n-1) + c \times (2n-1)$$

$$n T(n) = T(n-1) \times (n+1) + c(2n-1)$$

$$T(n) = T(n-1) \times \frac{(n+1)}{n} + c \frac{(2n-1)}{n} \quad ???$$

# RANDOMIZED ALGORITHMS

not random:  $x \rightarrow \boxed{\text{Algo}} \rightarrow y$

random:  
non-deterministic  
(can't determine the output.)

$x \rightarrow \boxed{\text{Algo+}}$

$\nearrow i$	$\rightarrow y$	0.5 chance.
$\nearrow j$	$\rightarrow 7$	0.49 chance
$\nearrow l$	$\rightarrow 6$	0.01 chance
$\nearrow r$	$\rightarrow 2$	0.25 chance

# RANDOMIZATION AND ALGORITHMS

## Random Input

→ can do with a deterministic algo.

what would the behaviour be if my input is random.

- Average Case analysis:
  - Input is drawn from some distribution  $\pi$ .
  - Under distribution  $\pi$ , average run-time, memory, etc...
- We saw an example when we analyzed QuickSort for a uniform distribution.

## Randomized Algorithms

- defined with randomization in terms of how it processes
- Algorithm flips a coin to make some decisions.
  - Non-Deterministic: simultaneously considers multiple algorithms weighted by the probability distribution.

# RANDOMIZED ALGORITHMS

Types of Randomized Algorithms:

→ will return correct answer with some probability.

Monte Carlo efficient and will give an answer, but answer may

- With probability  $p$  returns the correct answer: not be 100% correct.
  - Run multiple times to boost the probability of correct answer.
  - Provide an approximation guarantee in expectation.

→ not guaranteed to return a value, but if they return something, that is 100% correct.

Las Vegas

- Always returns the correct solution, or informs about failure.
- Has a run-time that is polynomial in expectation.

→ Vegas + Monte Carlo

Atlantic City

- Probabilistic run-time and correctness.

# RANDOMIZATION AND APPROXIMATION

→ how close are we to the correct answer.

## Guarantee in Expectation

Returns a solution that has a  $r$  approximation ratio in expectation:

$$\forall I, \mathbb{E}[\text{ALG}(I)] \leq r \cdot \text{OPT}(I) + \eta$$

=      ↓      ↘      ↗

for all inputs.    expected value    optimal value.

algo should return

inputs might be large.  
so, for any input  
your max off by  
a factor of  $r$ .

# PROBABILITY REVIEW / PRIMER

## Probability Space

- Sample space  $\underline{\Omega}$  of all possible outcomes.
  - Can be infinite, but we will focus on finite.
  - Ex: 4-sided die (D4):  $\Omega = \{1, 2, 3, 4\}$ .
- Probability mass: each  $i \in \Omega$  has a nonnegative probability mass:  $\underline{1} \geq \underline{p(i)} \geq \underline{0}$ . } non-negative
- Total probability mass is 1:  $\sum_{i \in \Omega} p(i) = 1$ .  
total probability mass = 1.

## Probability Event

- An event  $\underline{\varepsilon}$  is a set of outcomes of  $\Omega$ .
- $\Pr[\varepsilon] = \sum_{i \in \varepsilon} p(i)$ . } sum of their probability mass.
- Note:  $\Pr[\bar{\varepsilon}] = 1 - \Pr[\varepsilon]$

probability of event not happening.

TH:  $\Pr[\text{Roll 1 on a fair D4}] = 1/4$ ;

TH:  $\Pr[\text{Roll 2, 3, or 4 on a fair D4}] = 3/4$

# CONDITIONAL PROBABILITY AND INDEPENDENCE

Conditional Probability → probability of some event given the other event has already occurred.

Probability of  $\varepsilon$  given  $\mathcal{F}$ .

$$\Pr[\varepsilon | \mathcal{F}] = \frac{\Pr[\varepsilon \cap \mathcal{F}]}{\Pr[\mathcal{F}]}$$

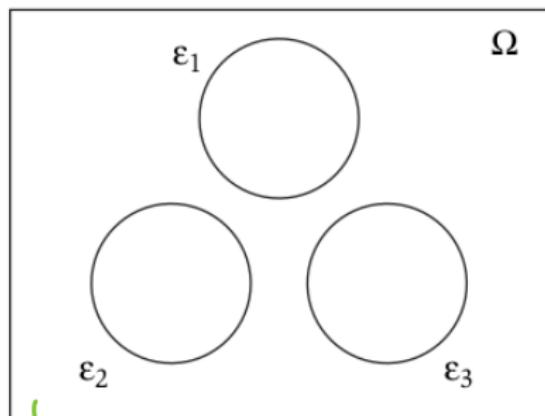
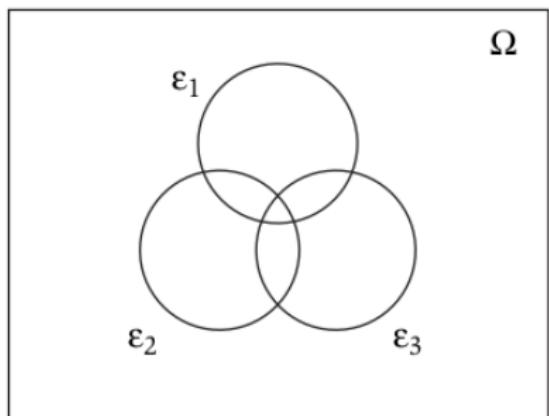
this event has happened.

## Independent Events

- Events  $\varepsilon$  and  $\mathcal{F}$  are independent if  $\Pr[\varepsilon | \mathcal{F}] = \Pr[\varepsilon]$  and  $\Pr[\mathcal{F} | \varepsilon] = \Pr[\mathcal{F}]$ . → probability of  $\varepsilon$  and  $\mathcal{F}$  happening.
- This implies  $\Pr[\varepsilon \cap \mathcal{F}] = \Pr[\varepsilon] \cdot \Pr[\mathcal{F}]$ . → only when independent.
- Generalization: Say  $\varepsilon_1, \varepsilon_2, \dots, \varepsilon_n$  are independent.

$$\Pr\left[\bigcap_{i=1}^n \varepsilon_i\right] = \prod_{i=1}^n \Pr[\varepsilon_i] \quad \left.\right\} \text{multiply their probabilities.}$$

# UNION BOUND



OR: Add probabilities:

equality when mutually exclusive.

## Union Bound

OR

$$\Pr \left[ \bigcup_{i=1}^n \varepsilon_i \right] \leq \sum_{i=1}^n \Pr[\varepsilon_i],$$

↑ union of all sets.

where equality only if events are mutually exclusive.

# RANDOM VARIABLES AND EXPECTATION

## Random Variables

- Technical: Given a probability space, a random variable  $X$  is a function from the sample space to the natural (finite – real if infinite) numbers, such that, for number  $j$ ,  $\underline{X^{-1}(j)}$  is the set of all sample points taking the value  $j$  is an event.

Ex:  $\Pr[X = 1] = 1/4$ , where  $X$  is a toss of a 4-sided die.

# RANDOM VARIABLES AND EXPECTATION

## Random Variables

- Informally: A random variable  $X$  takes on a value that depends on a random process.

Ex:  $\Pr[X = 1] = 1/4$ , where  $X$  is a toss of a 4-sided die.

↳ probability distribution across the values that  $X$  could be assigned.

## Expected Value

↳ weighted based on the values of that variable that can be assigned.

- “Weighted average value” be assigned.
- $\mathbb{E}[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j]$

TH: What is  $\mathbb{E}[X]$ , where  $X$  is a toss of a 4-sided die? 2.5

$$\begin{aligned}\mathbb{E}[X] &= 1 \times \frac{1}{4} + 2 \times \frac{1}{4} + 3 \times \frac{1}{4} + 4 \times \frac{1}{4} \\ &= 2.5\end{aligned}$$

# RANDOM VARIABLES AND EXPECTATION

## Expected Value

- “Weighted average value”
- $\mathbb{E}[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j]$

TH: What is  $\mathbb{E}[X]$ , where  $X$  is a toss of a 4-sided die? 2.5

## Expectation Properties

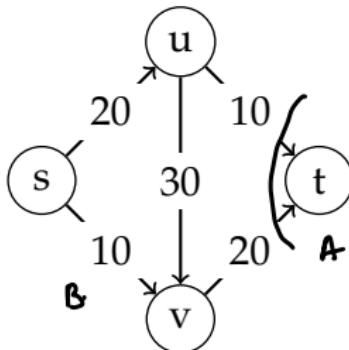
Let  $\underline{X}$  and  $\underline{Y}$  be random variables, and  $a$  be a constant.

- Linearity of expectation:
  - $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$
  - $\mathbb{E}[aX] = a \mathbb{E}[X]$
- If  $X$  and  $Y$  are independent,  $\mathbb{E}[XY] = \mathbb{E}[X] \mathbb{E}[Y]$ .

# MIN-CUT

*global min-cut*

# RANDOM MIN-CUT



Global min cut : 0

## Min-Cut

- A Cut: Partition of  $V$  into sets  $(A, B)$  with  $s \in A$  and  $t \in B$ .
- Cut capacity:  $c(A, B) = \sum_{e \text{ out of } A} c_e$  } minimum capacity cut.
- Minimum-cut of  $G$ : The cut  $(A^*, B^*)$  that minimizes  $c(A^*, B^*)$  for  $G$ .

## Why?

- We saw a polynomial time algorithm (flows).
- Because:
  - Nice example of a Monte Carlo algorithm.
  - Has a good run-time for dense graphs.

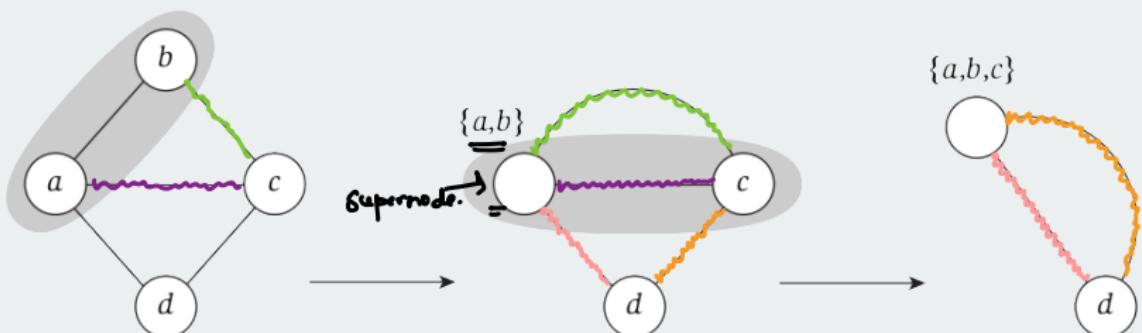
graph with a lot of edges

# GLOBAL MIN-CUT

## Some Notations

- Global meaning for any  $(s, t)$  pair. *→ consider all pairs of nodes as your (source, target).*
- An undirected multigraph  $G = \underline{(V, E)}$ :
- Every edge has capacity 1.
  - $E$  is a multiset:  $(u, v)$  might be in  $E$  more than once.
- $(u, v)$  edge contraction:
  - create a supernode  $\{u, v\}$

*supernode = combination of nodes.  
(you basically fuse nodes).*



# KARGER'S ALGORITHM

↳ Random Algo to find global min-cut.

---

## Algorithm: CONTRACTION ALGORITHM

---

**Input :** Multigraph  $G = (V, E)$

**Output:** Edge set representing a cut. } set of edges belonging to the cut.

if  $G$  has exactly 2 nodes  $u$  and  $v$  then

    | return the set of edges between  $u$  and  $v$  } Base case

else → edge contraction.

    | Choose an edge  $(u, v)$  uniformly at random. → will become the edge we contract.

    |  $\underline{G'} := G$  after contracting  $(u, v)$ .

    | return CONTRACTION ALGORITHM( $G'$ )

end

↳ recursively call on the new graph.

---

TH: Will this algorithm always return the correct answer? No

TH: What kind of randomized algorithm is this? Monte Carlo

# ANALYSIS OF KARGER'S ALGORITHM

*→ probability of this algo returning the global min-cut.*

## Theorem 1

*The CONTRACTION ALGORITHM returns a global min-cut of  $G$  with probability of at least  $1/\binom{n}{2}$ .*

*↳ lower bound.*

## Proof.

- Suppose that the global min-cut  $(A, B)$  has a size of  $k$ , and let  $F$  be the edge set. *if you have a node of degree less than  $k$ , then there will be a cut with  $u$  on one side and everything else on the other.*
- Every node has degree  $\geq k \Rightarrow |E| \geq \frac{1}{2}kn$ .
- $\Pr[\text{Edge in } F \text{ is contracted at step 1}] \leq \frac{k}{\frac{1}{2}kn} = \frac{2}{n}$ .

$$\Pr[\text{Edge in } F \text{ is contracted at step 1}] \leq \frac{k}{\frac{1}{2}kn} = \frac{2}{n}$$

↓  
 probability of making  
 a mistake in our  
 algo.  
 ↓  
 size  
 edge

-  $n$  nodes in the graph.  
 - every node has degree at least  $k$ .  
 - edge involves 2 nodes, so you divide by 2.

# ANALYSIS OF KARGER'S ALGORITHM

## Theorem 1

*The CONTRACTION ALGORITHM returns a global min-cut of  $G$  with probability of at least  $1/\binom{n}{2}$ .*

## Proof.

- Let  $\varepsilon_i$  be the event that an edge  $\in F$  is not contracted at step  $i$ :

↗ conditional prob.

$$\begin{aligned}
 \Pr[\text{success}] &= \Pr[\varepsilon_1] \cdot \Pr[\varepsilon_2 | \varepsilon_1] \cdots \Pr[\varepsilon_{n-2} | \varepsilon_1 \cap \varepsilon_2 \cap \cdots \cap \varepsilon_{n-3}] \\
 &\geq \left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n-1}\right) \cdots \left(1 - \frac{2}{n-i}\right) \cdots \left(1 - \frac{2}{3}\right) \\
 &= \left(\frac{n-2}{n}\right) \left(\frac{n-3}{n-1}\right) \left(\frac{n-4}{n-2}\right) \left(\frac{n-5}{n-3}\right) \cdots \left(\frac{2}{4}\right) \left(\frac{1}{3}\right) \\
 &= \frac{2}{n(n-1)} = \binom{n}{2}^{-1} \cdot \frac{1}{\binom{n}{2}}
 \end{aligned}$$

bound it by the probability of failure.

probability that we never choose an edge from that set.

# MULTIPLE RUNS OF CONTRACTION ALGORITHM

$$\lim_{n \rightarrow \infty} \left(1 - \frac{1}{e}\right)^n = \frac{1}{e}$$

## Multiple Runs

- With  $\binom{n}{2}$  runs, we get:

$$\Pr[\text{failure}] \leq \left(1 - \left(\frac{1}{\binom{n}{2}}\right)\right)^{\binom{n}{2}}$$

prob of success.  
}  $n^2$  runs.

- With  $\underbrace{\binom{n}{2} \ln n}$  runs, we get:

$$\Pr[\text{failure}] \leq \left(\frac{1}{e}\right)^{\ln n} = \frac{1}{n}$$

# HASHING

# HASHING

## Definition

A function that converts some input value into a hash value.

- Input: A large universe of values  $U$ . Typically, assume  $|U| \gg n$ .
- Output: A hash value for  $u \in U$  to  $\{0, 1, 2, \dots, n - 1\}$ .

*massive universe of items*

*turning element into a number*

## Why?

Typically used to generate keys for a dictionary data structure.

# DICTIONARY DATA STRUCTURE

## Dictionary

- Storage of a subset of values from  $U$ .
- A map, where the key is generated/hashed (efficiently) from the value.  
    ↳ Unique  
    ↳ Stored in the hashset / hashmap / hashtable.

## Dictionary Operations

- MAKEDICTIONARY: Initializes a fresh dictionary that can maintain a subset  $S$  of  $U$  that is initially empty.  
    ↳ declaring hashtable or hashmap
- INSERT( $u$ ): Adds  $u \in U$  to the dictionary ( $S$ ).
- DELETE( $u$ ): Remove  $u$  from  $S$ .
- LOOKUP( $u$ ): Determine if  $u$  is in  $S$ ; if so retrieve  $u$ .

# HASHING

## Motivation

- The values in  $U$  may be huge. Ex: Blog posts,
- Take a value  $u \in U$  and build a smaller key.

↳ efficiency: building a smaller key

## Hashing

- Hash Table*: a  $n$ -length array  $H$  to store the values.
- Hash Function*: Map  $u \in U$  to an index in  $H$ ;  

$$h : U \rightarrow [0..n - 1]$$

↳ map index to digit/number/key.

## Dictionary Hashing

- TH: Let  $u, v \in U$ . Say  $|U| \gg n$ , can  $\overbrace{h(u) = h(v)}$ ? Yes.
- Collision:  $h(u) = h(v)$  – At  $H[i]$  is a linked-list (bucket) to store any values where  $\underline{\underline{h(u) = i}}$ .

collisions are possible.

multiple things mapping the same value.

# HASHING

## Motivation

- The values in  $U$  may be huge. Ex: Blog posts.
- Take a value  $u \in U$  and build a smaller key.

## Hashing

- *Hash Table*: a  $n$ -length array  $H$  to store the values.
- *Hash Function*: Map  $u \in U$  to an index in  $H$ ;  
 $h : U \rightarrow [0..n - 1]$

## Dictionary Hashing

- Collision:  $h(u) = h(v)$  – At  $H[i]$  is a linked-list (bucket) to store any values where  $h(u) = i$ .
- TH: Say  $|S| = n$ , what is the worst-case number of comparisons to  $\text{LOOKUP}(u)$ ?  $O(n)$  } collide with each other.

# HASH FUNCTION DESIGN

→ Kolmogorov complexity (how easy it is to describe something).  
Good Hash Function → easy to calculate.

- Compact and efficient.
- Minimize the collisions.

## Some ideas for hash functions

- Hash as a prefix: Collisions can result from similar prefixes. E.g. many phrases in English start with "The".
- $u \bmod n$ : Risk of collision can be large especially if say  $n$  is a power of 2.
- $u \bmod p$ , where  $p$  is a prime: Less risk than  $n$  especially if  $p$  is not tiny, but  $p \approx n$ .

# RANDOM HASH FUNCTION

$h(x)$  : Return a value from  $(0)$  to  $(n - 1)$  UAR.

*uniformly at random*

## Lemma 2

Given  $h(x)$ , the probability that  $\underbrace{h(u) = h(v)}$  for any  $u, v \in U$  is  $\underline{\underline{1/n}}$ .  
*collision*

## Proof.

- There are  $n^2$  possible pairs of values  $(h(u), h(v))$ . Exactly  $n$  of them have  $\underbrace{h(u) = h(v)}$ . Hence,  $\Pr[h(u) = h(v)] = \frac{n}{n^2} = \frac{1}{n}$ .
- Alternate proof: Since  $h(u)$  and  $h(v)$  are independent:
  - Fix  $h(u)$ . What is the probability that  $h(u) = h(v)$ ?
  - $\Pr[h(v) = x | h(u) = x] = \frac{1}{n}$ .



What is the problem with this random hash function?

For a dictionary,  $\text{DELETE}(u)$  and  $\text{LOOKUP}(u)$  won't work since  $h(u)$  returns a random value!

# UNIVERSAL CLASS OF HASH FUNCTIONS

RANDOMLY CHOOSING A HASH FUNCTION

- ① want probability of collisions to be  $\gamma_n$ .
- ② want to lookup and delete

## Definition

Let  $\mathcal{H}$  be a class of functions such that:

- Universal property: For any pair of values  $u, v \in U$ , the probability that a randomly chosen  $h \in \mathcal{H}$  has  $h(u) = h(v)$  is  $\leq \frac{1}{n}$ .
- Each  $h \in \mathcal{H}$  is represented compactly and can be computed efficiently.

**MAKEDICTIONARY:** Given  $\mathcal{H}$ , choose  $h$  from  $\mathcal{H}$  UAR for the dictionary.

# UNIVERSAL CLASS OF HASH FUNCTIONS

RANDOMLY CHOOSING A HASH FUNCTION

→ If there exists a universe of class of hash functions, show

Theorem 3 that expected number of elements ( $n$  or less) that have/had collisions will be  $\leq 1$ .

Let  $\mathcal{H}$  be a universal class of hash functions mapping  $U$  to  $[0..n - 1]$ .

Let  $S \subseteq U$  be of size  $\leq n$ . The expected number of elements  $s \in S$  where  $h(s) = h(u)$  for any  $u \in U$  when  $h$  is chosen UAR from  $\mathcal{H}$  is  $\leq 1$ .

↳ pigeonhole principle

Proof.

$$X_s = 1 \text{ if } h(s) = h(u)$$

- Fix  $u \in U$ . Let  $X_s$  be a random variable that is 1 if  $h(s) = h(u)$ ; 0 otherwise.
- Let  $X = \sum_{s \in S} X_s$ . } how many collisions?
- By linearity of expectation:

$$\mathbb{E}[X] = \mathbb{E} \left[ \sum_{s \in S} X_s \right] = \sum_{s \in S} \mathbb{E}[X_s] \leq |S| \cdot \frac{1}{n} \leq 1.$$

summing across all items in  
S. odds of collision.

# MAX SAT

# MAX 3-SAT

## 3SAT Problem

Given a set of literals:  $X : x_1, \dots, x_n$ , and a collection of clauses  $\mathcal{C} : C_1 \wedge C_2 \wedge \dots \wedge C_k$ , each of length 3, does there exist a satisfying assignment?

## MAX 3SAT Problem

Given a 3SAT problem satisfying as many clauses as possible.

↳ given a 3SAT problem, how many clauses can we satisfy

## Random Assignment

For each  $x_i$ , independently assign a value of 0 or 1 with probability  $\frac{1}{2}$  each.

$$\underline{Y_2} \times \underline{Y_2} \times \underline{Y_2} = Y_8$$

## ANALYZE RANDOM ASSIGNMENT

Clause  $C_i$  *→ how many clauses do we expect to be satisfied given this random assignment to the boolean literals.*

- Let  $Z_i$  be a random variable:  $\underline{1}$  if clause is satisfied, 0 otherwise.
- TH: What is  $\Pr[Z_i = 0]?$   $\left(\frac{1}{2}\right)^3 = \frac{1}{8}$  *each clause has 3 literals.*
- Each clause has 3 variables  $x_i$  each with  $\Pr[x_i = 0] = \frac{1}{2}$ :

$$\Pr[Z_i = 1] = 1 - \Pr[Z_i = 0] = 1 - \left(\frac{1}{2}\right)^3 = \frac{7}{8}$$

- So,  $\mathbb{E}[Z_i] = 1 \cdot \frac{7}{8} + 0 \cdot \frac{1}{8} = \frac{7}{8}$ .

# ANALYZE RANDOM ASSIGNMENT

## Clause $C_i$

- Let  $Z_i$  be a random variable: 1 if clause is satisfied, 0 otherwise.
- So,  $\mathbb{E}[Z_i] = 1 \cdot \frac{7}{8} + 0 \cdot \frac{1}{8} = \frac{7}{8}$ . } for 1 clause

Overall

*If we have k clauses, so how many clauses do we expect to be satisfied. → you sum up the  $Z_i$  across all clauses.*

$$\text{Let } Z = \sum_{i=1}^k Z_i:$$

$$\mathbb{E}[Z] = \mathbb{E} \left[ \sum_{i=1}^k Z_i \right]$$

$= \mathbb{E}[Z_1] + \mathbb{E}[Z_2] + \dots + \mathbb{E}[Z_k]$ , by Linearity of Expectation,

$$= \frac{7}{8}k \rightarrow \text{expect } 7/8 \text{ clauses to be satisfied.}$$

# INTERESTING COROLLARIES

## Theorem 5

*Random Assign satisfies 7/8 of the clauses in expectation.*

→ expected value is a weighted avg, it will fall between

Corollary 6 the two extremes of what is possible

*For every 3-SAT, there is an assignment that satisfies 7/8 of the clauses.*

## Proof.

Since the expectation is a weighted average, its value is between the maximum and minimum possible values. □

# INTERESTING COROLLARIES

## Theorem 5

*Random Assign satisfies  $7/8$  of the clauses in expectation.*

## Corollary 6

*For every 3-SAT, there is an assignment that satisfies  $7/8$  of the clauses.*

## Corollary 7

*less than or equal to 7 clauses.*

*Every 3-SAT with  $\leq 7$  clauses is satisfiable.*

## Proof.

For  $k \leq 7$ ,  $\frac{7}{8}k > k - 1$ .

