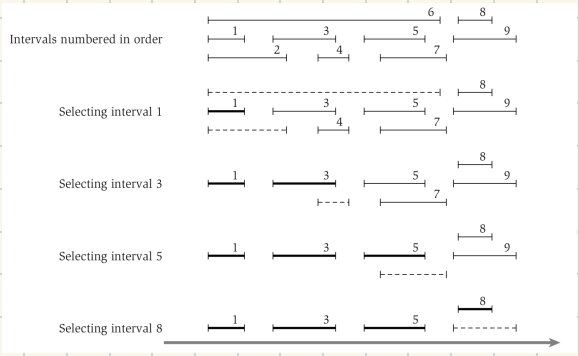
GREEDY

INTERVAL SCHEDULING

WE HAVE A SINGLE RESOURCE AND MANY DIFFERENT REQUESTS.

```
Initially let R be the set of all requests, and let A be empty
While R is not yet empty
  Choose a request i ∈ R that has the smallest finishing time
  Add request i to A
  Delete all requests from R that are not compatible with request i
EndWhile
Return the set A as the set of accepted requests
```

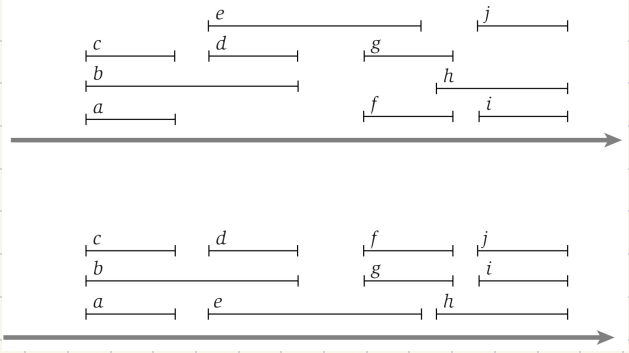


$O(n \log n)$

INTERVAL PARTITIONING

WE HAVE TO PARTITION ALL REQUESTS ACROSS MULTIPLE IDENTICAL RESOURCES.

```
Sort the intervals by their start times, breaking ties arbitrarily
Let I_1, I_2, ..., I_n denote the intervals in this order
For j = 1, 2, 3, ..., n
  For each interval I_i that precedes I_j in sorted order and overlaps it
     Exclude the label of I_i from consideration for I_j
  Endfor
  If there is any label from {1, 2, ..., d} that has not been excluded then
    Assign a nonexcluded label to I_j
  Else
    Leave I_j unlabeled
  Endif
Endfor
```

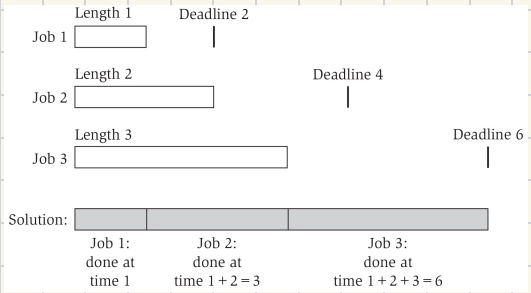


THIS REQUIRE AT LEAST D RESOURCES.
D IS THE MAXIMUM NUMBER OF INTERVALS WHICH OVERLAPS IN THE SAME TIME.

SCHEDULING TO MINIMIZE LATENESS

WE HAVE A SINGLE RESOURCE AND MANY DIFFERENT REQUESTS, BUT THIS TIME EACH REQUEST HAS A TIME LENGTH t_i AND A DEADLINE d_i.

```
Order the jobs in order of their deadlines
Assume for simplicity of notation that d_1 ≤ ... ≤ d_n
Initially, f = s
Consider the jobs i = 1, ..., n in this order
   Assign job i to the time interval from s(i) = f to f(i) = f + t_i
   Let f = f + t_i
End
Return the set of scheduled intervals [s(i), f(i)] for i = 1, ..., n
```



THERE ARE NO IDLE TIME (GAPS) AND NO INVERSIONS.

EXCHANGE ARGUMENT

TO PROVE THAT A IS OPTIMAL, WE TAKE AN OPTIMAL SOLUTION O WHICH WE TRASFORM STEP BY STEP INTO A, THROUGH A SEQUENCE OF LOCAL EXCHANGES, WITHOUT MAKING IT WORSE. THUS A IS AT LEAST AS GOOD AS O, SO OPTIMAL.

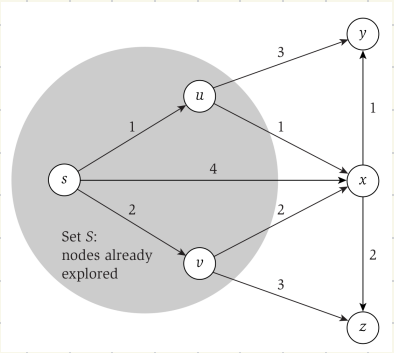
OPTIMAL CACHING

WHAT DATA I HAVE TO KEEP IN THE CACHE FOR A FAST ACCESS ?
WE WANT AS FEW CASH MISS AS POSSIBLE (BRING A REQUIRE DATA FROM THE MAIN MEMORY INTO THE CACHE).

> When d_i needs to be brought into the cache,
> evict the item that is needed the farthest into the future

SHORTEST PATHS IN GRAPHS

> Dijkstra's Algorithm (G, ℓ)
> Let S be the set of explored nodes
> For each $u \in S$, we store a distance $d(u)$
> Initially $S = \{s\}$ and $d(s) = 0$
> While $S \neq V$
> Select a node $v \notin S$ with at least one edge from S for which
> $d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible
> Add v to S and define $d(v) = d'(v)$
> EndWhile



WITH A PRIORITY QUEUE, THE ALG RUNS IN $O(m \log n)$

* IT 1: $S = \{s\}$ $d(s) = 0$

IT 2: $d'_s(u) = 1$ $S = \{s, u\}$ $d(u) = 1$
$d'_s(v) = 2$
$d'_s(x) = 4$

IT 3: $d'_s(v) = 2$ $S = \{s, u, v\}$ $d(v) = 2$
$d'_s(x) = 4$
$d'_u(x) = 1 + 1 = 2$
$d'_u(y) = 1 + 3 = 4$

IT 4: $d'_s(x) = 4$ $S = \{s, u, v, x\}$ $d(x) = 2$
$d'_u(x) = 1 + 1 = 2$
$d'_v(x) = 2 + 2 = 4$
$d'_u(y) = 1 + 3 = 4$
$d'_v(z) = 2 + 3 = 5$

IT 4: $d'_u(y) = 1 + 3 = 4$ $S = \{s, u, v, x, y\}$ $d(y) = 3$
$d'_x(y) = 1 + 1 + 1 = 3$
$d'_v(z) = 2 + 3 = 5$
$d'_x(z) = 1 + 1 + 2 = 4$

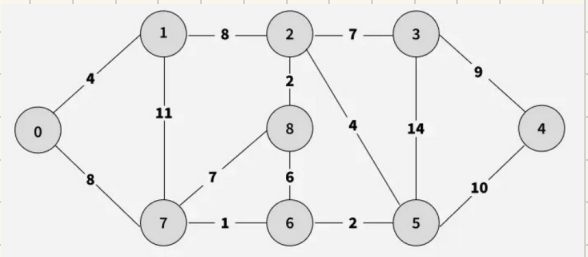
IT 4: $d'_v(z) = 2 + 3 = 5$ $S = \{s, u, v, x, y, z\}$ $d(z) = 4$
$d'_x(z) = 1 + 1 + 2 = 4$

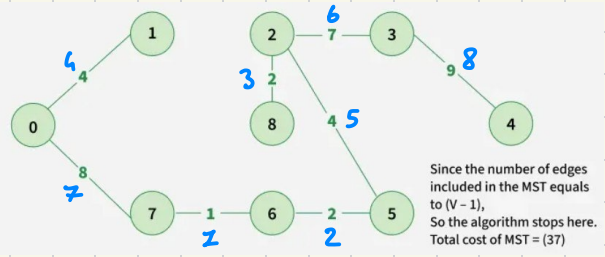
THE MINIMUM SPANNING TREE

THE PROBLEM IS TO FIND A SUBSET OF THE EDGES $T \subseteq E$ SUCH THAT THE GRAPH (V, T) IS CONNECTED, AND THE TOTAL COST IS AS SMALL AS POSSIBLE.

KRUSKAL $O(m \log n)$

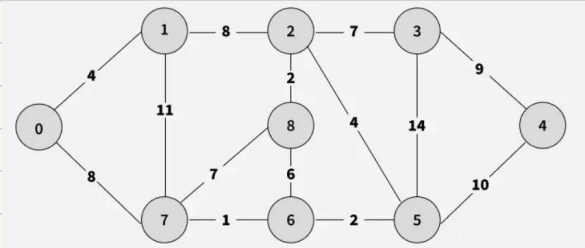
WE START WITHOUT ANY EDGES.
WE INSERT EDGES IN ORDER OF INCREASING COST AS LONG AS THEY DON'T CREATE A CYCLE.
IF INSERTING e WOULD RESULT IN A CYCLE, WE DISCARD IT.

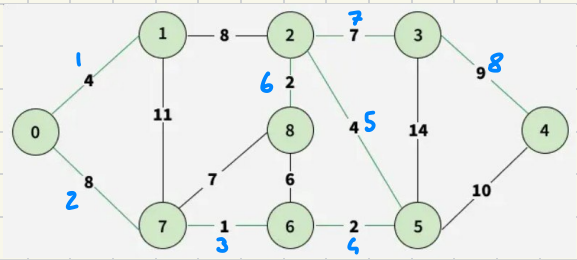




PRIM $O(m \log n)$

WE START WITH A ROOT NODE s AND TRY TO GROW A TREE FROM IT.
WE MAINTAIN A SET S OF ALREADY CONNECTED NODES.
AT EACH STEP WE ADD THE NODE (NOT IN S) THAT CAN BE ATTACHED AS CHEAPLY AS POSSIBLE (THE CHEAPEST EDGE).





REVERSE DELETE

WE START WITH THE FULL GRAPH (V, E).
WE DELETE EDGES IN ORDER OF DECREASING COST.

UNION FIND DATA STRUCTURE

WHEN AN EDGE IS ADDED TO THE GRAPH IN KRUSKAL'S ALG, WE DON'T WANT TO HAVE TO RECOMPUTE THE CONNECTED COMPONENTS FROM SCRATCH.
WITH A UNION FIND STRUCTURE WE STORE A REPRESENTATION OF THE COMPONENTS IN A WAY THAT SUPPORTS RAPID SEARCHING AND UPDATING.

FIND (u) = FIND (v) → THERE IS ALREADY AN EDGE
FIND (u) =! FIND (v) → UNION (FIND (u), FIND (v))

$O(m \log n)$

EXERCISE

1. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

Let G be an arbitrary connected, undirected graph with a distinct cost $c(e)$ on every edge e. Suppose e^* is the cheapest edge in G; that is, $c(e^*) < c(e)$ for every edge $e \neq e^*$. Then there is a minimum spanning tree T of G that contains the edge e^*.

TRUE, BECAUSE e^* IS THE FIRST EDGE THAT WOULD BE CONSIDERED BY KRUSKAL'S ALG, SO IT WILL BE INCLUDED IN THE MST.

2. For each of the following two statements, decide whether it is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

(a) Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph G, with edge costs that are all positive and distinct. Let T be a minimum spanning tree for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2, thereby creating a new instance of the problem with the same graph but different costs.

True or false? T must still be a minimum spanning tree for this new instance.

(b) Suppose we are given an instance of the Shortest s-t Path Problem on a directed graph G. We assume that all edge costs are positive and distinct. Let P be a minimum-cost s-t path for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2, thereby creating a new instance of the problem with the same graph but different costs.

True or false? P must still be a minimum-cost s-t path for this new instance.

a) TRUE, IF WE FEED THE COSTS c_e^2 INTO KRU'S ALG, IT WILL SORT THEM IN THE SAME ORDER, AND HENCE PUT THE SAME SUBSET OF EDGES IN THE MST.

b) FALSE, LET G HAVE EDGES (s,v), (v,t), (s,t), WHERE THE FIRST TWO OF THESE HAVE COST 3, AND THE LAST 5. INITIALLY THE SHORTEST PATH IS (s,t), BUT AFTER SQUARING THE COSTS IS $(s,v),(v,t)$.

3. You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package i has a weight w_i. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking. Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.

Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Your proof should follow the type of analysis we used for the Interval Scheduling Problem: it should establish the optimality of this greedy packing algorithm by identifying a measure under which it "stays ahead" of all other solutions.

WE CONSIDER TWO SOLUTIONS. IF THE GREEDY SOLUTION FITS $b_1 \dots b_j$ BOXES INTO THE FIRST k TRUCKS, AND THE OTHER SOLUTION FITS $b_1 \dots b_i$ BOXES INTO THE FIRST k TRUCKS, THEN $i \leq j$.

WE PROVE THIS BY INDUCTION.

$k=1$ → THE GREEDY ALG FITS AS MANY BOXES AS POSSIBLE INTO THE FIRST TRUCK.

k^{TH} → THE OTHER SOLUTION PACKS IN $b_{i'+1} \dots b_i$. THUS SINCE $j' \geq i'$, THE GREEDY ALG IS ABLE AT LEAST TO FIT ALL BOXES $b_{j'+1} \dots b_i$ INTO THE k^{TH} TRUCK.

4. Some of your friends have gotten into the burgeoning field of *time-series data mining*, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges—what's being bought—are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain "patterns" in them—for example, they may want to know if the four events

`buy Yahoo, buy eBay, buy Yahoo, buy Oracle`

occur in this sequence S, in order but not necessarily consecutively.

They begin with a collection of possible *events* (e.g., the possible transactions) and a sequence S of n of these events. A given event may occur multiple times in S (e.g., `Yahoo` stock may be bought many times in a single sequence S). We will say that a sequence S' is a *subsequence* of S if there is a way to delete certain of the events from S so that the remaining events, in order, are equal to the sequence S'. So, for example, the sequence of four events above is a subsequence of the sequence

`buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle`

Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of S. So this is the problem they pose to you: Give an algorithm that takes two sequences of events—S' of length m and S of length n, each possibly containing an event more than once—and decides in time $O(m+n)$ whether S' is a subsequence of S.

$S = S_1 \dots S_n$ $\quad K_i =$ FOUND MATCHES
$S' = S'_1 \dots S_m$

WE GIVE AN ALG THAT FINDS THE FIRST EVENT IN S THAT IS THE SAME AS S'_1, MATCHES THESE TWO EVENTS, THEN FINDS THE FIRST EVENT AFTER THIS THAT IS THE SAME AS S'_2...

```
i = j = 1
WHILE i ≤ n ∧ j ≤ m
  IF S_i = S'_j THEN
      K_j = i
      i = i + 1
      j = j + 1
  ELSE
      i = i + 1
END
```

5. Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible.

WE START AT THE WESTERN END AND BEGIN MOVING EAST UNTIL WE FIND A HOUSE h EXACTLY 4 MILES TO THE WEST, AND WE PLACE A BASE STATION. WE THEN DELETE ALL THE HOUSES COVERED AND ITERATE THE PROCESS ON THE REMAINING HOUSES.

$S = \{s_1, \dots, s_k\}$ GREEDY'S SOLUTION
$T = \{t_1, \dots, t_m\}$ O's SOLUTION

WE SHOW THAT $k = m$

$\forall i.\ s_i \geq t_i \rightarrow$ $i = 1$, TRUE SINCE WE GO AS FAR AS POSSIBLE TO EAST BEFORE PLACING THE FIRST STATION

$i \geq 1$, s_i COVER $h+8$ SPACES, THAT IS THE MAX. t_i CAN COVER ALSO $h+8$. THEREFORE GREEDY IS AS OPTIMAL AS THE OTHER SOLUTION.

8. Suppose you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

SUPPOSE FOR CONTRADDICTION THAT G HAS TWO MST, $T_1 \neq T_2$. SINCE THEY ARE DIFFERENT, THERE EXIST AN EDGE e (THE CHEAPEST) IN T_1 BUT NO IN T_2 ADDING e TO T_2 CREATES A CYCLE, WHICH MUST CONTAIN AN EDGE f THAT BELONGS TO T_2 BUT NO TO T_1. SINCE ALL COSTS ARE DIFFERENT AND e WAS CHOSEN AS THE CHEAPEST IN T_1, WE HAVE $c(e) < c(f)$.
REPLACING f WITH e WE HAVE A NEW SPANNING TREE WITH SMALLER TOTAL COST, CONTRADICTING THE ASSUMPTION THAT T_2 IS A MST.
THE MST OF G MUST BE UNIQUE.

9. One of the basic motivations behind the Minimum Spanning Tree Problem is the goal of designing a spanning network for a set of nodes with minimum *total* cost. Here we explore another type of objective: designing a spanning network for which the *most expensive* edge is as cheap as possible.

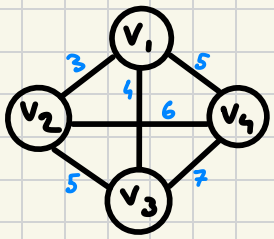
Specifically, let $G = (V, E)$ be a connected graph with n vertices, m edges, and positive edge costs that you may assume are all distinct. Let $T = (V, E')$ be a spanning tree of G; we define the *bottleneck edge* of T to be the edge of T with the greatest cost.

A spanning tree T of G is a *minimum-bottleneck spanning tree* if there is no spanning tree T' of G with a cheaper bottleneck edge.

(a) Is every minimum-bottleneck tree of G a minimum spanning tree of G? Prove or give a counterexample.

(b) Is every minimum spanning tree of G a minimum-bottleneck tree of G? Prove or give a counterexample.

a) FALSE. CONSIDER $\{v_1, v_2, v_3, v_4\}$ A GRAPH WITH EDGE COST $v_i, v_j = i + j$.



A POSSIBLE MBT IS $v_3 \to v_2 \to v_1 \to v_4$ THAT IS NOT A MST, SINCE THE MST IS FROM v_1 TO EVERY EDGES.

b) TRUE. A MST HAS ONLY THE CHEAPEST EDGES, AND THERE IS ALWAYS AT LEAST AN EDGE THAT IS BIGGER (BOTTLENECK). SINCE THIS PATH IS A MST, THERE IS NOT A CHEAPER BOTTLENECK SPANNING TREE, OTHERWISE THE PATH WOULDN'T HAVE BEEN A MST.

10. Let $G = (V, E)$ be an (undirected) graph with costs $c_e \geq 0$ on the edges $e \in E$. Assume you are given a minimum-cost spanning tree T in G. Now assume that a new edge is added to G, connecting two nodes $v, w \in V$ with cost c.

(a) Give an efficient algorithm to test if T remains the minimum-cost spanning tree with the new edge added to G (but not to the tree T). Make your algorithm run in time $O(|E|)$. Can you do it in $O(|V|)$ time? Please note any assumptions you make about what data structure is used to represent the tree T and the graph G.

(b) Suppose T is no longer the minimum-cost spanning tree. Give a linear-time algorithm (time $O(|E|)$) to update the tree T to the new minimum-cost spanning tree.

a) WE REPRESENT T USING A LIST, AND WE FIND THE v-w PATH IN T IN LINEAR TIME. IF EVERY EDGE HAS COST LESS THAN c, T IS A MST, OTHERWISE WE HAVE TO EXCHANGE THE EXPENSIVE ONE WITH THE NEW.

b) IF T IS NOT A MST MEANS THAT THERE IS AN EDGE f THAT IS EXPENSIVE AND IF $c(e) < c(f)$ WE SWAP THEM.

DYNAMIC PROGRAMMING

WEIGHTED INTERVAL SCHEDULING

THIS IS INTERVAL SCHEDULING IN WHICH EACH INTERVAL HAS A WEIGHT. THE GOAL IS TO SELECT A SUBSET $S \subseteq \{1, \dots, n\}$ OF MUTUALLY COMPATIBLE INTERVALS, SO AS TO MAXIMIZE THE SUM OF THE VALUES.

WE DEFINE $p(j)$ TO BE THE LARGEST INDEX $i < j$ SUCH THAT INTERVALS i AND j ARE DISJOINT.

$$OPT(j) = \max\left(v_j + OPT(p(j)), OPT(j-1)\right)$$

REQUEST j BELONGS TO AN OPTIMAL SOLUTION IFF $v_j + OPT(p(j)) \geq OPT(j-1)$

```
Compute-Opt(j)
  If j = 0 then
    Return 0
  Else
    Return max(v_j+Compute-Opt(p(j)), Compute-Opt(j − 1))
  Endif
```

COMPUTE - OPT (j) IS GOOD BUT WE CAN'T IMPLEMENT IT BECAUSE IT WOULD TAKE EXP TIME TO RUN.

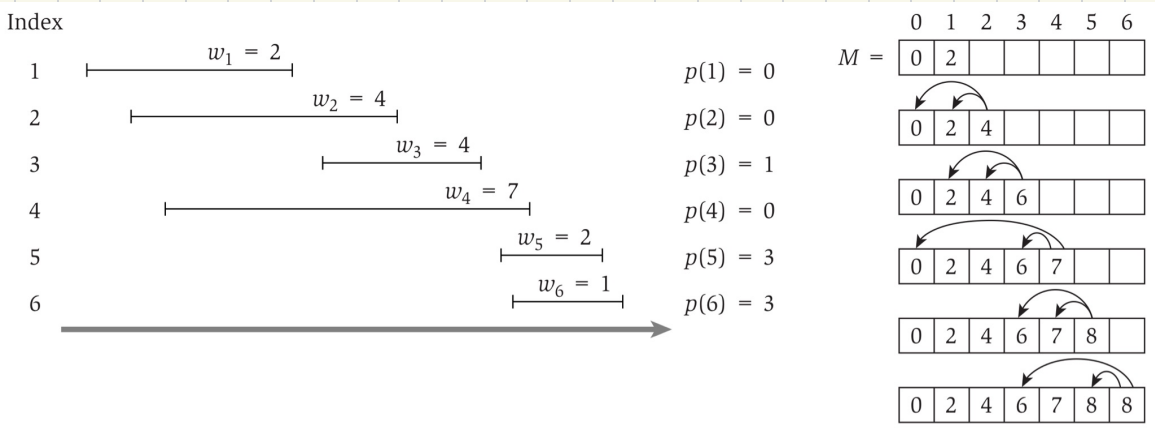
```
M-Compute-Opt(j)
  If j = 0 then
    Return 0
  Else if M[j] is not empty then
    Return M[j]
  Else
   Define M[j] = max(v_j+M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
    Return M[j]
  Endif
```

M - COMPUTE - OPT (j) (O(n)) USES MEMORIZATION STORING THE VALUE OF COMPUTE - OPT (j) IN A GLOBALLY PLACE THE FIRST TIME WE COMPUTE IT AND THEN USE THIS VALUE FOR ALL FUTURE REWRSIVE CALLS.

```
Find-Solution(j)
  If j = 0 then
    Output nothing
  Else
    If v_j + M[p(j)] ≥ M[j − 1] then
      Output j together with the result of Find-Solution(p(j))
    Else
      Output the result of Find-Solution(j − 1)
    Endif
  Endif
```

M - COMPUTE - OPT (j) COMPUTE THE VALUE OF AN OPTIMAL SOLUTION, BUT WE ALSO WANT A FULL OPTIMAL SET OF INTERVALS AS WELL.

FIND - SOLUTION (j) RETURNS AN OPTIMAL SOLUTION IN O(n) TIME



$$M[j] = \max\left(v_i + M[p(j)], M[j-1]\right)$$

SEGMENTED LEAST SQUARES - MULTI WAY CHOICES

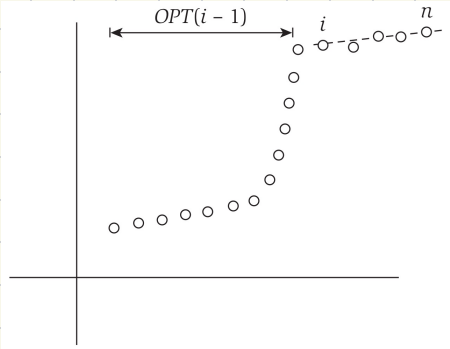
AT EACH STEP WE HAVE A POLYNOMIAL NUMBER OF POSSIBILITIES TO CONSIDER FOR THE OPTIMAL SOLUTION.

WE HAVE STATISTICAL DATA PLOTTED ON A TWO-DIMENSIONAL AXES. THE GOAL IS TO FIND A SET OF LINES THROUGH THE POINTS THAT MINIMIZE THE ERROR USING AS FEW LINES AS POSSIBLE

WE FIRST PARTITION THE SET OF POINTS P INTO SEGMENTS S, AND FOR EACH WE COMPUTE THE LINE. THERE ARE TWO PENALTY:

a. # OF TOTAL SEGMENTS · C (GIVEN)
b. ∀ SEGMENT, THE ERROR VALUE OF THE OPTIMAL LINE THROUGH THAT SEG

AS WE INCREASE THE NUMBER OF SEGMENT, WE REDUCE THE PENALTY b BUT WE INCREASE a.



IF THE LAST SEGMENT OF THE OPTIMAL PARTITION IS $p_i \dots p_n$, THEN THE VALUE OF THE OPTIMAL SOLUTION IS $OPT(n) = e_{i,n} + c + OPT(i-1)$

FOR THE SUBPROBLEM ON $p_1 \dots p_j$:

$$OPT(j) = \min_{1 \le i \le j} (e_{i,j} + c + OPT(i-1))$$

```
Segmented-Least-Squares(n)
  Array M[0...n]
  Set M[0] = 0
  For all pairs i ≤ j
    Compute the least squares error e_{i,j} for the segment p_i, ..., p_j
  Endfor
  For j = 1, 2, ..., n
    Use the recurrence (6.7) to compute M[j]
  Endfor
  Return M[n]
```

```
Find-Segments(j)
  If j = 0 then
    Output nothing
  Else
    Find an i that minimizes e_{i,j} + C + M[i − 1]
    Output the segment {p_i, ..., p_j} and the result of
          Find-Segments(i − 1)
  Endif
```

SUBSET SUMS - KNAPSACK

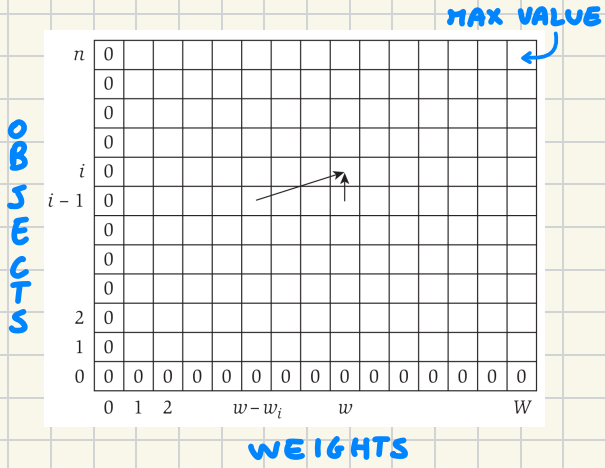
WE HAVE A MACHINE THAT PROCESS JOBS AND n REQUEST, EACH REQUIRE w_i TIME. THE MACHINE CAN BE USED BETWEEN TIME 0 AND W. THE GOAL IS TO PROCESS JOBS SO AS TO KEEP THE MACHINE AS BUSY AS POSSIBLE UP TO THE CUT OFF W, BUT WHICH JOBS?

IF $w < w_i$ THEN $OPT(i,w) = OPT(i-1,w)$. OTHERWISE

$$OPT(i,w) = \max\left(OPT(i-1,w),\ w_i + OPT(i-1,w-w_i)\right)$$

```
Subset-Sum(n, W)
  Array M[0...n, 0...W]
  Initialize M[0, w] = 0 for each w = 0, 1, ..., W
  For i = 1, 2, ..., n
     For w = 0, ..., W
        Use the recurrence (6.8) to compute M[i, w]
     Endfor
  Endfor
  Return M[n, W]
```

$O(nW)$



EX: $W = 5$ $\quad w_1 = 1,\ w_2 = 3,\ w_3 = 4$

IF $w < w_i$

$$M[i][w] = M[i-1][w]$$

ELSE

$$M[i][w] = \max(M[i-1][w], w_i + M[i-1][w-w_i])$$

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

$M[n][w] = 5$

FOR KNAPSACK WE HAVE ALSO A VALUE v_i FOR EACH REQUEST. SO THE GOAL IS TO MAXIMIZE THE SUM OF THE VALUE.

IF $w < w_i$ THEN $OPT(i,w) = OPT(i-1,w)$. OTHERWISE

$$OPT(i,w) = \max\left(OPT(i-1,w),\ v_i + OPT(i-1,w-w_i)\right)$$

SEQUENCE ALIGNMENT

o-currance
occurrence

HOW SHOULD WE DEFINE SIMILARITY BETWEEN TWO STRINGS?

A **MATCHING** IS A SET OF ORDERED PAIRS IN WHICH EACH ITEM OCCURS IN AT MOST ONE PAIR.
A MATCHING M OF TWO SETS IS AN **ALIGNMENT** IF THERE ARE NO CROSSING PAIRS. IF $(i,j),(i',j') \in M$ AND $i < i'$, THEN $j < j'$.

THE GOAL IS TO FIND THE OPTIMAL ALIGNMENT BETWEEN X AND Y, MINIMIZING THE TOTAL COST.
SUPPOSE M IS A GIVEN ALIGNMENT:

- THERE IS A PARAMETER $\delta > 0$ THAT DEFINES **GAP PENALTY**. FOR EACH POSITION IN X OR Y THAT IS NO MATCHED IN M (GAP) WE INCUR A COST OF δ.
- FOR EACH $(i,j) \in M$, WE PAY A **MISMATCH COST** α_{x_i, y_j} FOR LINING UP x_i WITH y_j ($\alpha_{pp} = 0$).
- THE COST OF M IS THE SUM OF GAP + MISMATCH COSTS.

IN AN OPTIMAL ALIGNMENT M, AT LEAST ONE OF THESE IS TRUE:

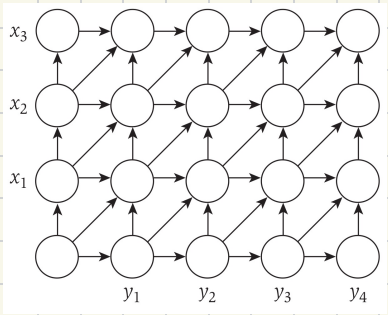
a. $(m,n) \in M$;
b. THE m^{TH} POSITION OF X IS NOT MATCHED;
c. THE n^{TH} POSITION OF Y IS NOT MATCHED.

↳ IF a HOLDS: $OPT(m,n) = \alpha_{x_m y_n} + OPT(m-1, n-1)$
IF b HOLDS: $OPT(m,n) = \delta + OPT(m-1, n)$
IF c HOLDS: $OPT(m,n) = \delta + OPT(m, n-1)$

THE MINIMUM ALIGNMENT COST SATISFY: FOR $i, j \geq 1$

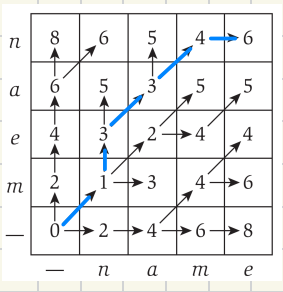
$$OPT(i,j) = \min\left[\alpha_{x_i y_j} + OPT(i-1, j-1),\ \delta + OPT(i-1, j),\ \delta + OPT(i, j-1)\right]$$

```
Alignment(X, Y)
  Array A[0...m, 0...n]
  Initialize A[i, 0] = iδ for each i
  Initialize A[0, j] = jδ for each j
  For j = 1, ..., n
     For i = 1, ..., m
        Use the recurrence (6.16) to compute A[i, j]
     Endfor
  Endfor
  Return A[m, n]
```



OPT(i,j) IS THE LENGHT OF THE SHORTEST PATH.

$$OPT[i][j] = \min \begin{cases} \alpha_{x_i y_j} + OPT[i-1][j-1] \\ \delta + OPT[i-1][j] \\ \delta + OPT[i][j-1] \end{cases}$$



$\delta = 2$

MISMATCH:
VOWEL-VOWEL : 1
CONS - CONS : 1
VOWEL-CONS : 3

SHORTEST PATHS IN GRAPHS

NOW THE EDGES CAN HAVE NEGATIVE COST.

1. GIVEN A GRAPH WE FIRST DEFINE IF G HAS A NEGATIVE CYCLE $\sum_{i,j \in C} c_{ij} < 0$.

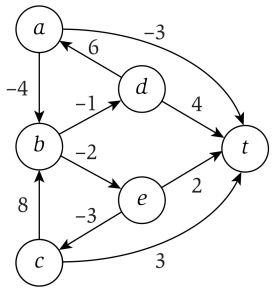
2. IF G HAS NO NEGATIVE CYCLES, WE FIND A PATH P FROM s TO t WITH THE MINIMUM COST.

DIJKSTRA DOESN'T WORK BECAUSE A PATH THAT STARTS ON AN EXPENSIVE EDGE, BUT THEN COMPENSATES WITH SUBSEQUENT EDGES OF NEGATIVE COSTS, CAN BE CHEAPER THAN A PATH THAT STARTS ON A CHEAP EDGE.

WE CAN USE SUBPROBLEM WITH BELLMAN FORD EQUATION.

IF $i > 0$ $\quad OPT(i, v) = \min\left(OPT(i-1, v), \min_{w \in V}\left(OPT(i-1, w) + c_{vw}\right)\right)$

```
Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[0 ... n − 1, V]
  Define M[0, t] = 0 and M[0, v] = ∞ for all other v ∈ V
  For i = 1, ..., n − 1
    For v ∈ V in any order
      Compute M[i, v] using the recurrence (6.23)
    Endfor
  Endfor
  Return M[n − 1, s]
```



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	−3	−3	−4	−6	−6
b	∞	∞	0	−2	−2	−2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

DISTANCE VECTOR PROTOCOL

THE BELLMAN'S ALG IS PULL BASED, IN EACH ITERATION i, EACH NODE v HAS TO CONTACT ITS NEIGHBOR w AND PULL THE NEW VALUE $M[w]$. IF A NODE w HASN'T CHANGED ITS VALUE, THERE IS NO NEED FOR v TO TAKE THE VALUE AGAIN. SO WE NEED A PUSH ALG, IN WHICH EACH NODE w WHOSE DISTANCE VALUE $M[w]$ CHANGES IN AN ITERATION INFORMS, PUSHING, ALL ITS NEIGHBORS OF THE CHANGE IN THE NEW ITERATION, BUT ONLY IF $M[w]$ CHANGED.

```
Push-Based-Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[V]
  Initialize M[t] = 0 and M[v] = ∞ for all other v ∈ V
  For i = 1, ..., n − 1
    For w ∈ V in any order
      If M[w] has been updated in the previous iteration then
        For all edges (v, w) in any order
          M[v] = min(M[v], c_vw + M[w])
          If this changes the value of M[v], then first[v] = w
        Endfor
    Endfor
    If no value changed in this iteration, then end the algorithm
  Endfor
  Return M[s]
```

```
Asynchronous-Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[V]
  Initialize M[t] = 0 and M[v] = ∞ for all other v ∈ V
  Declare t to be active and all other nodes inactive
  While there exists an active node
    Choose an active node w
      For all edges (v, w) in any order
        M[v] = min(M[v], c_vw + M[w])
        If this changes the value of M[v], then
          first[v] = w
          v becomes active
      Endfor
      w becomes inactive
  EndWhile
```

IF NODES ARE ROUTERS IN A NETWORK WE NEED AN ASYNCHRONOUS VERSION, IN WHICH EACH TIME A NODE w HAS AN UPDATE TO ITS $M[w]$, IT BECOMES ACTIVE AND NOTIFY ITS NEIGHBORS.

EXERCISE

1. Let $G=(V,E)$ be an undirected graph with n nodes. Recall that a subset of the nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we'll see that it can be done efficiently if the graph is "simple" enough.

Call a graph $G=(V,E)$ a *path* if its nodes can be written as $v_1, v_2, \ldots, v_n$, with an edge between v_i and v_j if and only if the numbers i and j differ by exactly 1. With each node v_i, we associate a positive integer *weight* w_i.

Consider, for example, the five-node path drawn in Figure 6.28. The *weights* are the numbers drawn inside the nodes.

The goal in this question is to solve the following problem:

Find an independent set in a path G whose total weight is as large as possible.

(a) Give an example to show that the following algorithm *does not* always find an independent set of maximum total weight.

```
The "heaviest-first" greedy algorithm
  Start with S equal to the empty set
  While some node remains in G
    Pick a node v_i of maximum weight
    Add v_i to S
    Delete v_i and its neighbors from G
  Endwhile
  Return S
```

(b) Give an example to show that the following algorithm also *does not* always find an independent set of maximum total weight.

```
Let S_1 be the set of all v_i where i is an odd number
Let S_2 be the set of all v_i where i is an even number
(Note that S_1 and S_2 are both independent sets)
Determine which of S_1 or S_2 has greater total weight,
  and return this one
```

(c) Give an algorithm that takes an n-node path G with weights and returns an independent set of maximum total weight. The running time should be polynomial in n, independent of the values of the weights.

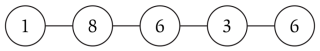


Figure 6.28 A paths with weights on the nodes. The maximum weight of an independent set is 14.

a) CONSIDER THE SEQUENCE OF WEIGHTS 2, 3, 2. THE ALG WILL TAKE 3, BUT THE MAXIMUM WEIGHT IS 4.

b) CONSIDER THE SEQUENCE OF WEIGHTS 3, 1, 2, 3. THE ALG WILL TAKE 3 AND 2 BUT THE MAXIMUM WEIGHT IS 3 AND 3.

c) LET S_i DENOTE AN INDEPENDENT SET ON $\{v_1 \ldots v_i\}$ AND LET X_i DENOTE ITS WEIGHT.

$X_0 = 0 \quad X_1 = w_1 \quad i > 1:$ a. $X_i = w_i + X_{i-2}$ b. $X_i = X_{i-1}$

$X_i = \max(X_{i-1}, w_i + X_{i-2})$

X_n IS THE RESULT AND WE CAN CALCULATE S_n BY TRACING BACK.

2. Suppose you're managing a consulting team of expert computer hackers, and each week you have to choose a job for them to undertake. Now, as you can well imagine, the set of possible jobs is divided into those that are *low-stress* (e.g., setting up a Web site for a class at the local elementary school) and those that are *high-stress* (e.g., protecting the nation's most valuable secrets, or helping a desperate group of Cornell students finish a project that has something to do with compilers). The basic question, each week, is whether to take on a low-stress job or a high-stress job.

If you select a low-stress job for your team in week i, then you get a revenue of $\ell_i > 0$ dollars; if you select a high-stress job, you get a revenue of $h_i > 0$ dollars. The catch, however, is that in order for the team to take on a high-stress job in week i, it's required that they do no job (of either type) in week $i-1$; they need a full week of prep time to get ready for the crushing stress level. On the other hand, it's okay for them to take a low-stress job in week i even if they have done a job (of either type) in week $i-1$.

So, given a sequence of n weeks, a *plan* is specified by a choice of "low-stress," "high-stress," or "none" for each of the n weeks, with the property that if "high-stress" is chosen for week $i > 1$, then "none" has to be chosen for week $i-1$. (It's okay to choose a high-stress job in week 1.) The *value* of the plan is determined in the natural way: for each i, you add ℓ_i to the value if you choose "low-stress" in week i, and you add h_i to the value if you choose "high-stress" in week i. (You add 0 if you choose "none" in week i.)

The problem. Given sets of values $\ell_1, \ell_2, \ldots, \ell_n$ and $h_1, h_2, \ldots, h_n$, find a plan of maximum value. (Such a plan will be called *optimal*.)

Example. Suppose $n=4$, and the values of ℓ_i and h_i are given by the following table. Then the plan of maximum value would be to choose "none" in week 1, a high-stress job in week 2, and low-stress jobs in weeks 3 and 4. The value of this plan would be $0+50+10+10=70$.

	Week 1	Week 2	Week 3	Week 4
ℓ	10	1	10	10
h	5	50	5	1

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
For iterations i = 1 to n
  If h_{i+1} > ℓ_i + ℓ_{i+1} then
    Output "Choose no job in week i"
    Output "Choose a high-stress job in week i+1"
    Continue with iteration i+2
  Else
    Output "Choose a low-stress job in week i"
    Continue with iteration i+1
  Endif
End
```

To avoid problems with overflowing array bounds, we define $h_i = \ell_i = 0$ when $i > n$.

In your example, say what the correct answer is and also what the above algorithm finds.

(b) Give an efficient algorithm that takes values for $\ell_1, \ell_2, \ldots, \ell_n$ and $h_1, h_2, \ldots, h_n$ and returns the *value* of an optimal plan.

a) THE ALG IS TOO SHORT-SIGHTED:

HERE IT WOULD TAKE h JOB IN W2, WHILE THE OPT IS $\ell, 0, h$

	W1	W2	W3
ℓ	3	4	2
h	1	10	50

b) $OPT(i) = \max(\ell_i + OPT(i-1), h_i + OPT(i-2))$

$OPT(1) = \max(\ell_1, h_1)$

THE SEQUENCE OF JOBS CAN BE RECONSTRUCTED BY TRACING BACK THROUGH THE SET OF OPT VALUES

3. Let $G = (V, E)$ be a directed graph with nodes $v_1, \ldots, v_n$. We say that G is an *ordered graph* if it has the following properties.

(i) Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form (v_i, v_j) with $i < j$.

(ii) Each node except v_n has at least one edge leaving it. That is, for every node v_i, $i = 1, 2, \ldots, n-1$, there is at least one edge of the form (v_i, v_j).

The length of a path is the number of edges in it. The goal in this question is to solve the following problem (see Figure 6.29 for an example).

Given an ordered graph G, find the length of the longest path that begins at v_1 and ends at v_n.

(a) Show that the following algorithm does not correctly solve this problem, by giving an example of an ordered graph on which it does not return the correct answer.

```
Set w = v_1
Set L = 0

While there is an edge out of the node w
  Choose the edge (w, v_j)
     for which j is as small as possible
  Set w = v_j
  Increase L by 1
end while
Return L as the length of the longest path
```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes an ordered graph G and returns the *length* of the longest path that begins at v_1 and ends at v_n. (Again, the *length* of a path is the number of edges in the path.)

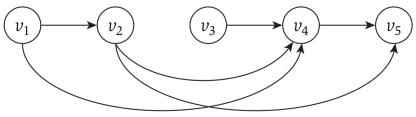
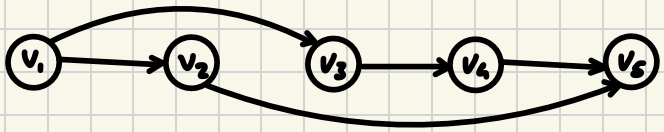


Figure 6.29 The correct answer for this ordered graph is 3: The longest path from v_1 to v_n uses the three edges (v_1, v_2),(v_2, v_4), and (v_4, v_5).

a)



THE ALG WUOLD RETURN 2 $(v_1,v_2)(v_2,v_5)$.
BUT THE OPT IS 3 $(v_1,v_3)(v_3,v_4),(v_4,v_5)$.

b)

```
ARRAY M[1...n]
M[1]=0
FOR i = 2...n
   M = -∞
   FORALL EDGES (j,i)
      IF M[j] ≠ -∞
         IF M < M[j+1]
            M = M[j+1]
   M[i]=M
RETURN M[n]
```

4. Suppose you're running a lightweight consulting business—just you, two associates, and some rented equipment. Your clients are distributed between the East Coast and the West Coast, and this leads to the following question.

Each month, you can either run your business from an office in New York (NY) or from an office in San Francisco (SF). In month i, you'll incur an *operating cost* of N_i if you run the business out of NY; you'll incur an operating cost of S_i if you run the business out of SF. (It depends on the distribution of client demands for that month.)

However, if you run the business out of one city in month i, and then out of the other city in month $i+1$, then you incur a fixed *moving cost* of M to switch base offices.

Given a sequence of n months, a *plan* is a sequence of n locations—each one equal to either NY or SF—such that the i^{th} location indicates the city in which you will be based in the i^{th} month. The *cost* of a plan is the sum of the operating costs for each of the n months, plus a moving cost of M for each time you switch cities. The plan can begin in either city.

The problem. Given a value for the moving cost M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, find a plan of minimum cost. (Such a plan will be called *optimal*.)

Example. Suppose $n = 4$, $M = 10$, and the operating costs are given by the following table.

	Month 1	Month 2	Month 3	Month 4
NY	1	3	20	30
SF	50	20	2	4

Then the plan of minimum cost would be the sequence of locations

$$[NY, NY, SF, SF],$$

with a total cost of $1 + 3 + 2 + 4 + 10 = 20$, where the final term of 10 arises because you change locations once.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
For i = 1 to n
  If N_i < S_i then
    Output "NY in Month i"
  Else
    Output "SF in Month i"
End
```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an example of an instance in which every optimal plan must move (i.e., change locations) at least three times.

Provide a brief explanation, saying why your example has this property.

(c) Give an efficient algorithm that takes values for n, M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, and returns the *cost* of an optimal plan.

a)

	M1	M2	M3
NY	2	5	4
SF	4	3	6

M = 20

2 + 3 + 4 + 60 INSTEAD OF 2+5+4

b)

	M1	M2	M3	M4
NY	1	100	1	100
SF	100	1	100	1

M = 10

OPT IS 1 + 10 + 1 + 10 + 1 + 10 + 1 = 34
IT CHANGES 3 TIMES.

c)

$OPT_N(0) = OPT_S(0) = 0$
FOR $i = 1 \ldots n$
$OPT_N(i) = N_i + \min(OPT_N(i-1), M + OPT_S(i-1))$
$OPT_S(i) = S_i + \min(OPT_S(i-1), M + OPT_N(i-1))$
RETURN THE SMALLER BETWEEN OPT_N / OPT_S

5. As some of you know well, and others of you may be interested to learn, a number of languages (including Chinese and Japanese) are written without spaces between the words. Consequently, software that works with text written in these languages must address the *word segmentation problem*—inferring likely boundaries between consecutive words in the text. If English were written without spaces, the analogous problem would consist of taking a string like "meetateight" and deciding that the best segmentation is "meet at eight" (and not "me et at eight," or "meet ate ight," or any of a huge number of even less plausible alternatives). How could we automate this process?

A simple approach that is at least reasonably effective is to find a segmentation that simply maximizes the cumulative "quality" of its individual constituent words. Thus, suppose you are given a black box that, for any string of letters $x = x_1x_2 \cdots x_k$, will return a number *quality*(x). This number can be either positive or negative; larger numbers correspond to more plausible English words. (So *quality*("*me*") would be positive, while *quality*("*ght*") would be negative.)

Given a long string of letters $y = y_1y_2 \cdots y_n$, a segmentation of y is a partition of its letters into contiguous blocks of letters; each block corresponds to a word in the segmentation. The *total quality* of a segmentation is determined by adding up the qualities of each of its blocks. (So we'd get the right answer above provided that *quality*("*meet*") + *quality*("*at*") + *quality*("*eight*") was greater than the total quality of any other segmentation of the string.)

Give an efficient algorithm that takes a string y and computes a segmentation of maximum total quality. (You can treat a single call to the black box computing *quality*(x) as a single computational step.)

OPT(i) = min$_{j \le i}$ (OPT(j-1), QUALITY(j...n))

7. As a solved exercise in Chapter 5, we gave an algorithm with $O(n \log n)$ running time for the following problem. We're looking at the price of a given stock over n consecutive days, numbered $i = 1, 2, \ldots, n$. For each day i, we have a price $p(i)$ per share for the stock on that day. (We'll assume for simplicity that the price was fixed during each day.) We'd like to know: How should we choose a day i on which to buy the stock and a later day $j > i$ on which to sell it, if we want to maximize the profit per share, $p(j) - p(i)$? (If there is no way to make money during the n days, we should conclude this instead.)

In the solved exercise, we showed how to find the optimal pair of days i and j in time $O(n \log n)$. But, in fact, it's possible to do better than this. Show how to find the optimal numbers i and j in time $O(n)$.

X_j = RETURN IF SELL IN DAY j

X_j = max (0, X_{j-1} + (P(j) - P(j-1)))

DON'T HAVE THE STOCK

9. You're helping to run a high-performance computing system capable of processing several terabytes of data per day. For each of n days, you're presented with a quantity of data; on day i, you're presented with x_i terabytes. For each terabyte you process, you receive a fixed revenue, but any unprocessed data becomes unavailable at the end of the day (i.e., you can't work on it in any future day).

You can't always process everything each day because you're constrained by the capabilities of your computing system, which can only process a fixed number of terabytes in a given day. In fact, it's running some one-of-a-kind software that, while very sophisticated, is not totally reliable, and so the amount of data you can process goes down with each day that passes since the most recent reboot of the system. On the first day after a reboot, you can process s_1 terabytes, on the second day after a reboot, you can process s_2 terabytes, and so on, up to s_n; we assume $s_1 > s_2 > s_3 > \cdots > s_n > 0$. (Of course, on day i you can only process up to x_i terabytes, regardless of how fast your system is.) To get the system back to peak performance, you can choose to reboot it; but on any day you choose to reboot the system, you can't process any data at all.

The problem. Given the amounts of available data $x_1, x_2, \ldots, x_n$ for the next n days, and given the profile of your system as expressed by $s_1, s_2, \ldots, s_n$ (and starting from a freshly rebooted system on day 1), choose the days on which you're going to reboot so as to maximize the total amount of data you process.

Example. Suppose $n = 4$, and the values of x_i and s_i are given by the following table.

	Day 1	Day 2	Day 3	Day 4
x	10	1	7	7
s	8	4	2	1

The best solution would be to reboot on day 2 only; this way, you process 8 terabytes on day 1, then 0 on day 2, then 7 on day 3, then 4 on day 4, for a total of 19. (Note that if you didn't reboot at all, you'd process $8 + 1 + 2 + 1 = 12$; and other rebooting strategies give you less than 19 as well.)

(a) Give an example of an instance with the following properties.
- There is a "surplus" of data in the sense that $x_i > s_1$ for every i.
- The optimal solution reboots the system at least twice.

In addition to the example, you should say what the optimal solution is. You do not need to provide a proof that it is optimal.

(b) Give an efficient algorithm that takes values for $x_1, x_2, \ldots, x_n$ and $s_1, s_2, \ldots, s_n$ and returns the total *number* of terabytes processed by an optimal solution.

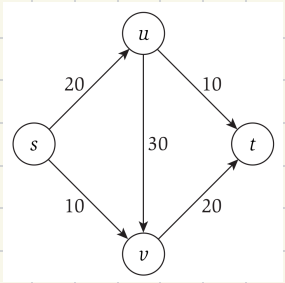
a)

	1	2	3	4	..
x_i	11	11	11	11	
s_i	10	1	1	1	

THE OPT SHOULD REBOOT IN EVERY OTHER DAY, PROCESSING 10 TB EVERY TWO DAYS.

NETWORK FLOW

MAXIMUM FLOW



FLOW NETWORK IS A DIRECT GRAPH $G = (V, E)$ WITH:

- EACH EDGE e HAS A CAPACITY c_e
- THERE IS A SINGLE SOURCE NODE $s \in V$
- THERE IS A SINGLE SINK NODE $t \in V$

AN **s-t FLOW** IS A FUNCTION $f: E \to R^+$ AND $f(e)$ REPRESENTS THE AMOUNT OF FLOW CARRIED BY e. A FLOW f MUST SATISFY:

- $\forall e \in E,\ 0 \leq f(e) \leq c_e$
- $\forall v \neq s \text{ v } t,\ \sum_{e \text{ INTO } v} f(e) = \sum_{\substack{e \text{ OUT} \\ \text{OF } v}} f(e)$

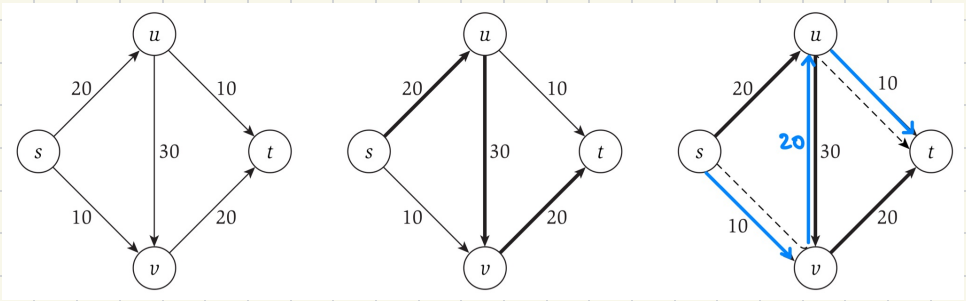
THE VALUE OF A FLOW $v(f) = \sum_{\substack{e \text{ OUT} \\ \text{OF } s}} f(e)$

GIVEN A FLOW NETWORK, THE GOAL IS TO FIND A FLOW OF MAXIMUM POSSIBLE VALUE.

WE CAN PUSH FORWARD ON EDGES WITH LEFTOVER CAPACITY, AND PUSH BACKWARD ON EDGES THAT ARE ALREADY CARRYING FLOW, TO DIVERT IT IN A DIFFERENT DIRECTION.

GIVEN A FLOW NETWORK G, AND A FLOW f ON G, WE DEFINE THE RESIDUAL GRAPH G_f OF G WITH RESPECT TO f THUS:

- G AND G_f HAVE THE SAME NODE SET.
- $\forall e = (u, v)$ OF G ON WHICH $f(e) < c_e$, THERE ARE $c_e - f(e)$ LEFTOVER UNITS OF CAPACITY ON WHICH WE COULD TRY TO PUSH FORWARD. SO WE INCLUDE e IN G_f, WITH CAPACITY $c_e - f(e)$ (FORWARD EDGE).
- $\forall e = (u, v)$ OF G ON WHICH $f(e) > 0$, THERE ARE $f(e)$ UNITS OF FLOW THAT WE CAN UNDO, BY PUSHING FLOW BACKWARD. SO WE INCLUDE $e' = (v, u)$ IN G_f WITH CAPACITY $f(e)$ (BACKWARD EDGE).



LET P BE AN s-t PATH IN G_f. WE DEFINE BOTTLENECK(P, f) TO BE THE MINIMUM RESIDUAL CAPACITY OF ANY EDGE ON P. WE DEFINE A NEW FLOW f' IN G, OBTAINED BY INCREASING / DECREASING THE FLOW VALUES ON EDGES ON P.

```
augment(f, P)
  Let b = bottleneck(P, f)
  For each edge (u, v) ∈ P
    If e = (u, v) is a forward edge then
      increase f(e) in G by b
    Else ((u, v) is a backward edge, and let e = (v, u))
      decrease f(e) in G by b
    Endif
  Endfor
  Return(f)
```

```
Max-Flow
  Initially f(e) = 0 for all e in G
  While there is an s-t path in the residual graph G_f
    Let P be a simple s-t path in G_f
    f' = augment(f, P)
    Update f to be f'
    Update the residual graph G_f to be G_f'
  Endwhile
  Return f
```

CUT

CONSIDER DIVIDING THE NODES OF THE GRAPH INTO TWO SETS A AND B SO THAT $s \in A$ AND $t \in B$. AN **s-t CUT** IS A PARTITION (A, B) OF THE VERTEX SET V, SO THAT $s \in A$ AND $t \in B$ THE CAPACITY OF A CUT $c(A,B)$ IS THE SUM OF THE CAPACITIES OF ALL EDGES OUT OF A: $c(A,B) = \sum_{e \text{ out of } A} c_e$ WITH A CUT $v(f) = f^{out}(A) - f^{in}(A)$ AND $v(f) = f^{in}(B) - f^{out}(B)$. $v(f) \leq c(A,B)$.

GOOD PATH

AUGMENTATION INCREASES THE VALUE OF THE MAXIMUM FLOW BY THE BOTTLENECK CAPACITY OF THE SELECTED PATH, SO IF WE CHOOSE PATHS WITH LARGE BOTTLENECK CAPACITY, WE WILL BE MAKING A LOT OF PROGRESS.

HAVING TO FIND SUCH PATHS CAN SLOW DOWN EACH ITERATION. SO WE WILL NOT SELECT THE PATH WITH THE LARGEST BOTTLENECK, BUT WE WILL MAINTAIN A SCALING PARAMETER Δ, AND WE WILL LOOK FOR PATHS THAT HAVE BOTTLENECK CAPACITY AT LEAST Δ.

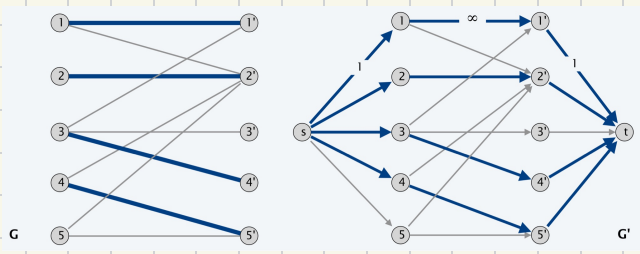
```
Scaling Max-Flow
  Initially f(e) = 0 for all e in G
  Initially set Δ to be the largest power of 2 that is no larger
       than the maximum capacity out of s: Δ ≤ max_{e out of s} c_e
    While Δ ≥ 1
      While there is an s-t path in the graph G_f(Δ)
        Let P be a simple s-t path in G_f(Δ)
        f' = augment(f, P)
        Update f to be f' and update G_f(Δ)
      Endwhile
      Δ = Δ/2
    Endwhile
Return f
```

BIPARTITE MATCHING

A GRAPH G IS BIPARTITE IF THE NODES CAN BE PARTITIONED INTO TWO SUBSETS L AND R S.T. EVERY EDGE CONNECTS A NODE IN L TO ONE IN R.

GIVEN A BIPARTITE GRAPH $G = (L \cup R, E)$ FIND A MAX CARDINALITY MATCHING.

- CREATE $G' = (L \cup R \cup \{s, t\}, E')$
- ADD SOURCE s AND UNIT CAPACITY EDGES FROM s TO L.
- ADD SINK t AND UNIT CAPACITY EDGES FROM R TO t.
  WE COMPUTE THE VALUE OF MAX FLOW IN G'



MAX CARDINALITY OF A MATCHING IN G = MAX FLOW IN G'

G HAS A **PERFECT MATCHING** IFF $|N(S)| \geq |S|$ FORALL SUBSETS $S \subseteq L$.

DISJOINT PATH

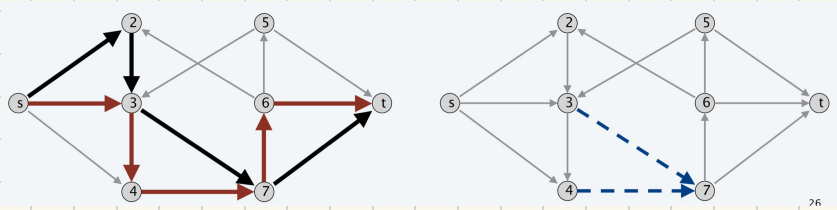
TWO PATHS ARE DISJOINT IF THEY HAVE NO EDGE IN COMMON.
GIVEN A GRAPH $G=(V, E)$ AND TWO NODES s AND t, FIND THE MAX NUMBER OF EDGE DISJOINT s-t PATHS.

- ASSIGN UNIT CAPACITY TO EVERY EDGE
- WE COMPUTE THE VALUE OF MAX FLOW IN G'

THE MAX NUMBER OF EDGE DISJOINT s-t PATHS = VALUE OF MAX FLOW

A SET OF EDGES $F \subseteq E$ DISCONNECTS t FROM s IF EVERY s-t PATH USES AT LEAST ONE EDGE IN F.

THE MAX NUMBER OF EDGE DISJOINT s-t PATHS = min NUMBER OF EDGES WHOSE REMOVAL DISCONNECTS t FROM s.

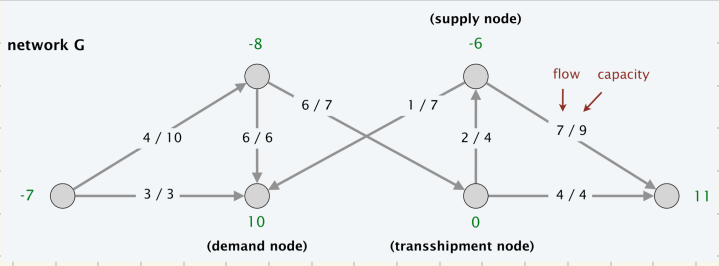


IN AN UNDIRECTIONAL GRAPH WE REPLACE EACH EDGE WITH TWO ANTI PARALLEL EDGES AND ASSIGN UNIT CAPACITY TO EVERY EDGE

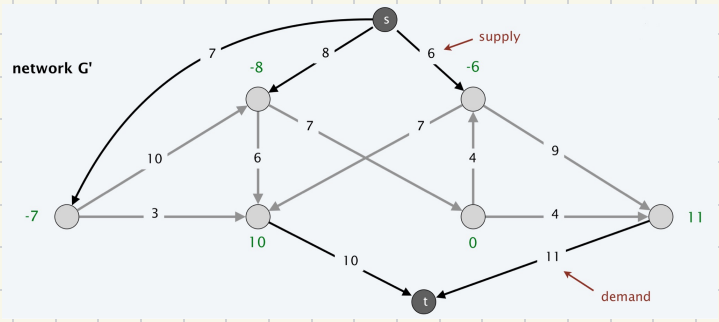
EXTENSION TO MAXFLOW

WE HAVE A NETWORK G LIKE THIS.

$d(v) > 0$: NODE REQUIRES FLOW
$d(v) < 0$: NODE PROVIDES FLOW
$d(v) = 0$: TRANSIT NODE



- ADD SOURCE s AND SINK t.
- FOR EACH v WITH $d(v) < 0$, ADD EDGE (s, v) WITH CAPACITY $-d(v)$.
- FOR EACH v WITH $d(v) > 0$, ADD EDGE (v, t) WITH CAPACITY $d(v)$.



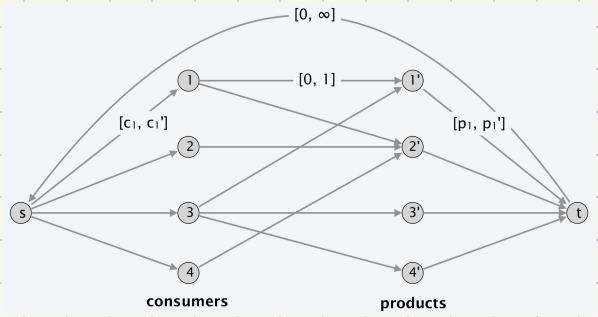
G HAS CIRCULATION IFF G' HAS MAX FLOW OF VALUE $D = \sum_{d(v)>0} d(v) = \sum_{d(v)<0} -d(v)$.

ALL EDGES SATURATED

SURVEY DESIGN

WE ASK n CONSUMERS ABOUT m PRODUCTS.

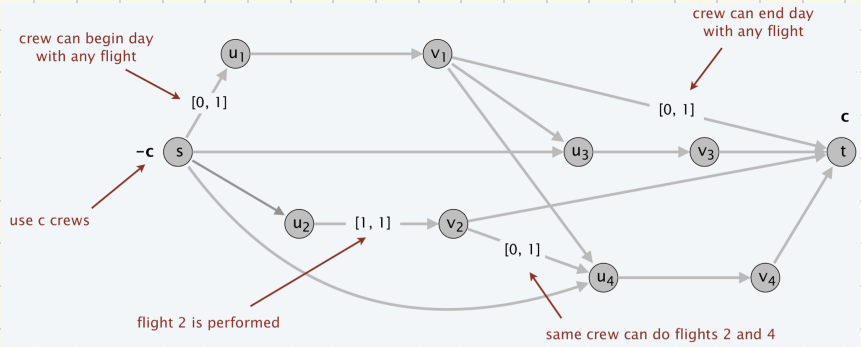
- ADD EDGE (i, j) IF CONSUMER j OWNS PRODUCT i.
- ADD EDGE FROM s TO CONSUMER j.
- ADD EDGE FROM PRODUCT i TO t.
- ADD EDGE FROM t TO s.



AIRLINE SCHEDULING

WE HAVE A SET OF k FLIGHTS IN A DAY. EACH FLIGHT i LEAVES ORIGIN o_i AT TIME s_i AND ARRIVES IN DESTINATION d_i AT TIME t_i. WE HAVE TO MINIMIZE THE NUMBER OF FLIGHT CREWS BY REUSING THEM OVER MULTIPLE FLIGHT.

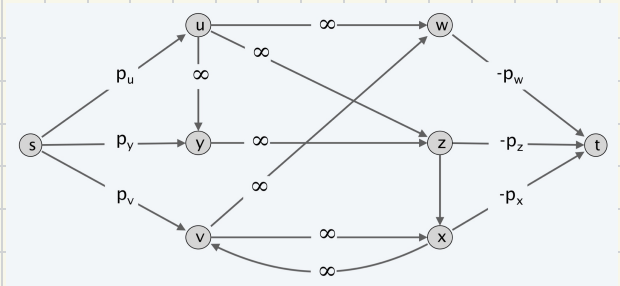
- FOR EACH FLIGHT i, INCLUDE TWO NODES u_i AND v_i.
- ADD SOURCE s WITH DEMAND $-c$, AND EDGES (s, u_i) WITH CAPACITY 1.
- ADD SINK t WITH DEMAND c, AND EDGES (v_i, t) WITH CAPACITY 1.
- FOR EACH i, ADD EDGE (u_i, v_i) WITH LOWERBOUND AND CAPACITY 1.
- IF FLIGHT j REACHABLE FROM i, ADD EDGE (v_i, u_j) WITH CAPACITY 1.



PROJECT SELECTION

GIVEN A SET OF PROJECTS P AND PREREQUISITIES E, CHOOSE A FEASIBLE SUBSET OF PROJECTS TO MAXIMIZE REVENUE.

- ASSIGN ∞ TO ALL PREREQUISITE EDGE.
- ADD EDGE (s, v) WITH CAPACITY p_v IF $p_v > 0$.
- ADD EDGE (v, t) WITH CAPACITY $-p_v$ IF $p_v < 0$.
- ADD EDGE (v, w) IF CAN'T DO v WITHOUT ALSO DOING w.
- DEFINE $p_s = p_t = 0$

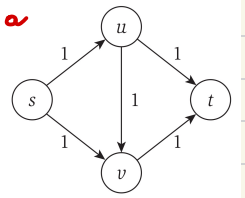


EXERCISE

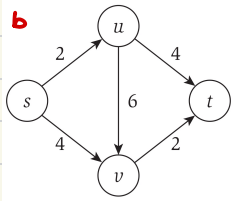
1. **(a)** List all the minimum *s*-*t* cuts in the flow network pictured in Figure 7.24. The capacity of each edge appears as a label next to the edge.

(b) What is the minimum capacity of an *s*-*t* cut in the flow network in Figure 7.25? Again, the capacity of each edge appears as a label next to the edge.

a



b



a) MINIMUM CAPACITY = 2

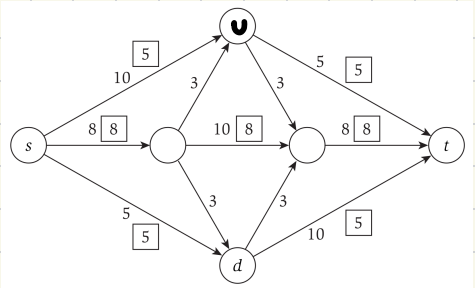
$(\{s\}, \{u, v, t\})$, $(\{s, v\}, \{u, t\})$, $(\{s, u, v\}, \{t\})$

b) WITH THE CUT $(\{s, v\}, \{u, t\})$ THE MINIMUM CAPACITY IS 4

2. Figure 7.26 shows a flow network on which an *s*-*t* flow has been computed. The capacity of each edge appears as a label next to the edge, and the numbers in boxes give the amount of flow sent on each edge. (Edges without boxed numbers—specifically, the four edges of capacity 3—have no flow being sent on them.)

(a) What is the value of this flow? Is this a maximum (s,t) flow in this graph?

(b) Find a minimum *s*-*t* cut in the flow network pictured in Figure 7.26, and also say what its capacity is.



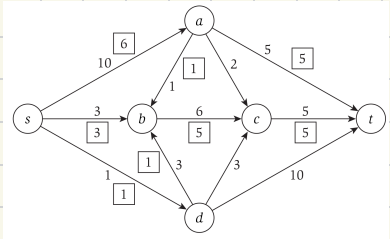
a) $v(f) = 18$ BUT IT'S NOT THE MAXIMUM.

b) A MINIMUM CUT IS S AND U IN A AND THE REST IN B, WITH CAPACITY 21.

3. Figure 7.27 shows a flow network on which an *s*-*t* flow has been computed. The capacity of each edge appears as a label next to the edge, and the numbers in boxes give the amount of flow sent on each edge. (Edges without boxed numbers have no flow being sent on them.)

(a) What is the value of this flow? Is this a maximum (s,t) flow in this graph?

(b) Find a minimum *s*-*t* cut in the flow network pictured in Figure 7.27, and also say what its capacity is.



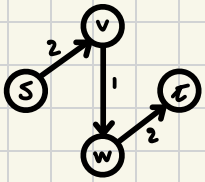
a) $v(f) = 10$ BUT IT'S NOT THE MAXIMUM.

b) A MINIMUM CUT IS $(\{s, a, b, c\}, \{d, t\})$ WITH CAPACITY 11.

4. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

Let G be an arbitrary flow network, with a source s, a sink t, and a positive integer capacity c_e on every edge e. If f is a maximum s-t flow in G, then f saturates every edge out of s with flow (i.e., for all edges e out of s, we have $f(e) = c_e$).

FALSE.

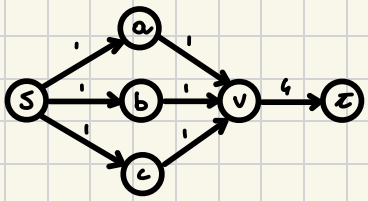


HERE MAXIMUM FLOW IS 1 AND DOESN'T SATURATE S-V.

5. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

Let G be an arbitrary flow network, with a source s, a sink t, and a positive integer capacity c_e on every edge e; and let (A, B) be a mimimum s-t cut with respect to these capacities $\{c_e : e \in E\}$. Now suppose we add 1 to every capacity; then (A, B) is still a minimum s-t cut with respect to these new capacities $\{1 + c_e : e \in E\}$.

FALSE.



$(\{s\}, \{a,b,c,v,t\})$ IS A CUT WITH MINIMUM CAPACITY 3 IF WE ADD 1 TO ALL EDGES THE MINIMUM CUT IS $(\{s,a,b,c,v\}, \{t\})$ WITH 5 INSTEAD OF 6.

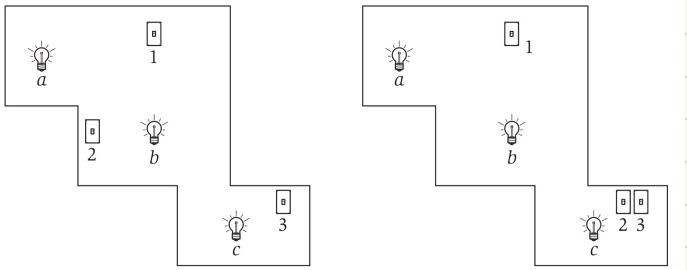
6. Suppose you're a consultant for the Ergonomic Architecture Commission, and they come to you with the following problem.

They're really concerned about designing houses that are "user-friendly," and they've been having a lot of trouble with the setup of light fixtures and switches in newly designed houses. Consider, for example, a one-floor house with n light fixtures and n locations for light switches mounted in the wall. You'd like to be able to wire up one switch to control each light fixture, in such a way that a person at the switch can *see* the light fixture being controlled.

Sometimes this is possible and sometimes it isn't. Consider the two simple floor plans for houses in Figure 7.28. There are three light fixtures (labeled a, b, c) and three switches (labeled 1, 2, 3). It is possible to wire switches to fixtures in Figure 7.28(a) so that every switch has a line of sight to the fixture, but this is not possible in Figure 7.28(b).

Let's call a floor plan, together with n light fixture locations and n switch locations, *ergonomic* if it's possible to wire one switch to each fixture so that every fixture is visible from the switch that controls it. A floor plan will be represented by a set of m horizontal or vertical line segments in the plane (the walls), where the i^{th} wall has endpoints $(x_i, y_i), (x_i', y_i')$. Each of the n switches and each of the n fixtures is given by its coordinates in the plane. A fixture is *visible* from a switch if the line segment joining them does not cross any of the walls.

Give an algorithm to decide if a given floor plan is ergonomic. The running time should be polynomial in m and n. You may assume that you have a subroutine with $O(1)$ running time that takes two line segments as input and decides whether or not they cross in the plane.



WE BUILD A BIPARTITE GRAPH $G = (V, E)$

$V = X \cup Y$ → $x_i \in X$ = SWITCH i
$y_j \in Y$ = FIXTURE j

(x_i, y_j) IS AN EDGE IFF THE LINE SEGMENT FROM x_i TO y_j DOESN'T INTERSECT ANY OF THE m WALLS.

7. Consider a set of mobile computing clients in a certain town who each need to be connected to one of several possible *base stations*. We'll suppose there are n clients, with the position of each client specified by its (x, y) coordinates in the plane. There are also k base stations; the position of each of these is specified by (x, y) coordinates as well.

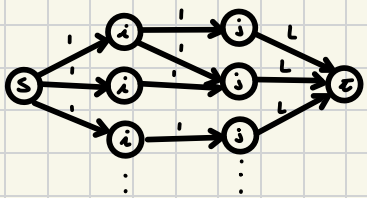
For each client, we wish to connect it to exactly one of the base stations. Our choice of connections is constrained in the following ways. There is a *range parameter* r—a client can only be connected to a base station that is within distance r. There is also a *load parameter* L—no more than L clients can be connected to any single base station.

Your goal is to design a polynomial-time algorithm for the following problem. Given the positions of a set of clients and a set of base stations, as well as the range and load parameters, decide whether every client can be connected simultaneously to a base station, subject to the range and load conditions in the previous paragraph.

WE BUILD THIS FLOW NETWORK:

THERE IS A NODE v_i FOR EACH CLIENT i, A NODE w_j FOR EACH BASE j, AND AN EDGE (v_i, w_j) OF CAPACITY 1 IF i IS WITHIN RANGE OF BASE STATION j. WE THEN CONNECT A SUPER SOURCE s TO EACH i WITH CAPACITY 1, AND EACH j TO A SUPERTANK OF CAPACITY L.

THERE IS A FEASIBLE WAY TO CONNECT ALL CLIENTS TO BASE STATIONS IFF THERE IS AN s-t FLOW OF VALUE n.



8. Statistically, the arrival of spring typically results in increased accidents and increased need for emergency medical treatment, which often requires blood transfusions. Consider the problem faced by a hospital that is trying to evaluate whether its blood supply is sufficient.

The basic rule for blood donation is the following. A person's own blood supply has certain *antigens* present (we can think of antigens as a kind of molecular signature); and a person cannot receive blood with a particular antigen if their own blood does not have this antigen present. Concretely, this principle underpins the division of blood into four *types*: A, B, AB, and O. Blood of type A has the A antigen, blood of type B has the B antigen, blood of type AB has both, and blood of type O has neither. Thus, patients with type A can receive only blood types A or O in a transfusion, patients with type B can receive only B or O, patients with type O can receive only O, and patients with type AB can receive any of the four types.[4]

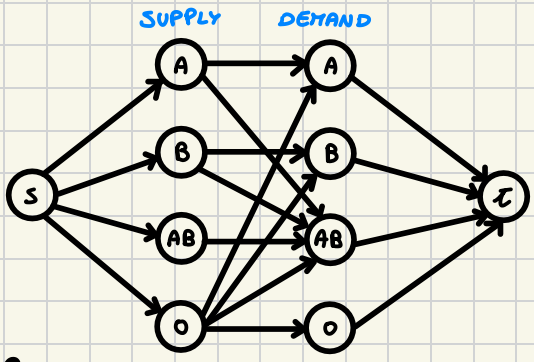
(a) Let s_O, s_A, s_B, and s_{AB} denote the supply in whole units of the different blood types on hand. Assume that the hospital knows the projected demand for each blood type d_O, d_A, d_B, and d_{AB} for the coming week. Give a polynomial-time algorithm to evaluate if the blood on hand would suffice for the projected need.

(b) Consider the following example. Over the next week, they expect to need at most 100 units of blood. The typical distribution of blood types in U.S. patients is roughly 45 percent type O, 42 percent type A, 10 percent type B, and 3 percent type AB. The hospital wants to know if the blood supply it has on hand would be enough if 100 patients arrive with the expected type distribution. There is a total of 105 units of blood on hand. The table below gives these demands, and the supply on hand.

blood type	supply	demand
O	50	45
A	36	42
B	11	8
AB	8	3

Is the 105 units of blood on hand enough to satisfy the 100 units of demand? Find an allocation that satisfies the maximum possible number of patients. Use an argument based on a minimum-capacity cut to show why not all patients can receive blood. Also, provide an explanation for this fact that would be understandable to the clinic administrators, who have not taken a course on algorithms. (So, for example, this explanation should not involve the words *flow*, *cut*, or *graph* in the sense we use them in this book.)

a)



b) NO BECAUSE 36 (A) + 5 (O) ARE NOT SUFFICIENT TO SATISFY 42 UNITS OF A.

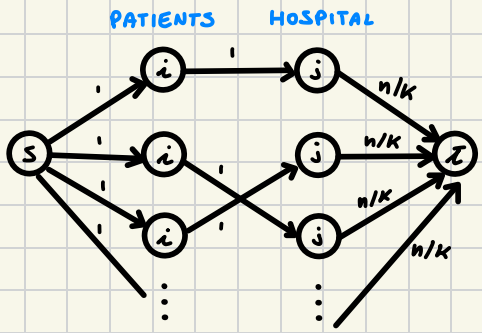
9. Network flow issues come up in dealing with natural disasters and other crises, since major unexpected events often require the movement and evacuation of large numbers of people in a short amount of time.

Consider the following scenario. Due to large-scale flooding in a region, paramedics have identified a set of n injured people distributed across the region who need to be rushed to hospitals. There are k hospitals in the region, and each of the n people needs to be brought to a hospital that is within a half-hour's driving time of their current location (so different people will have different options for hospitals, depending on where they are right now).

At the same time, one doesn't want to overload any one of the hospitals by sending too many patients its way. The paramedics are in touch by cell phone, and they want to collectively work out whether they can choose a hospital for each of the injured people in such a way that the load on the hospitals is *balanced*: Each hospital receives at most $\lceil n/k \rceil$ people.

Give a polynomial-time algorithm that takes the given information about the people's locations and determines whether this is possible.

(i, j) IF i IS WITHIN A HALF HOUR DRIVE OF HOSPITAL j



THERE IS A FEASIBLE WAY TO SEND ALL i TO HOSPITALS IFF THERE IS AN S-T FLOW OF VALUE n.

NP PROBLEMS

POLYNOMIAL TIME REDUCTION

THE KEY IDEA IS TO COMPARE THE DIFFICULTY OF DIFFERENT PROBLEMS

"PROBLEM X IS AT LEAST AS HARD AS PROBLEM Y"

CAN ARBITRARY INSTANCES OF PROBLEM Y BE SOLVED USING A POLYNOMIAL NUMBER OF STANDARD COMPUTATIONAL STEPS + A POLYNOMIAL NUMBER OF CALLS TO A BLACK BOX THAT SOLVES PROBLEM X?

↳ IF THE ANSWER IS YES WE DEFINE $Y \leq_p X$ AS:

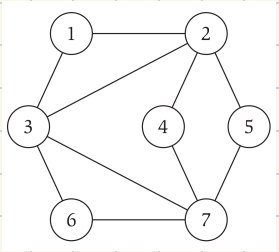
"Y IS POLYNOMIAL-TIME REDUCIBLE TO X. IF X CAN BE SOLVED IN POLYNOMIAL TIME, THEN Y TOO"

"IF $Y \leq_p X$ AND X CAN'T BE SOLVED IN POLYNOMIAL TIME, NEITHER Y."

IF Y IS KNOW TO BE HARD, AND WE SHOW $Y \leq_p X$, X IS AS HARD AS Y.

INDEPENDENT SET AND VERTEX COVER

IN A GRAPH $G = (V, E)$ WE SAY A SET OF NODES $S \subseteq V$ IS INDEPENDENT IF NO TWO NODES IN S ARE JOINED BY AN EDGE. IT'S EASY TO FIND A SMALL SET, BUT NO FOR LARGE SETS. SO IT'S MORE CONVENIENT TO WORK WITH YES/NO PROBLEMS.



> GIVEN A GRAPH G AND A NUMBER K, DOES G CONTAIN AN INDEPENDENT SET OF SIZE AT LEAST K?

IF WE CAN SOLVE FOR THE LARGEST K FOR WHICH THE ANSWER IS YES, WE FIND THE LARGEST INDEPENDENT SET IN G.

GIVEN A GRAPH $G = (V, E)$, WE SAY THAT A SET OF NODES $S \subseteq V$ IS A VERTEX COVER IF EVERY EDGE $e \in E$ HAS AT LEAST ONE END IN S. IT'S EASY TO FIND A LARGE VERTEX COVER, BUT DIFFICULT FOR SMALL.

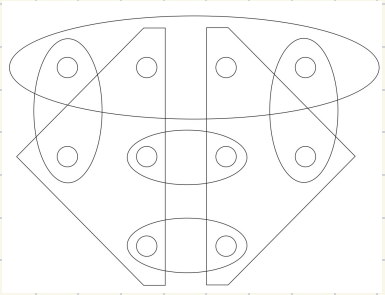
> GIVEN A GRAPH G AND A NUMBER K, DOES G CONTAIN A VERTEX COVER OF SIZE AT MOST K?

$G = (V, E) \rightarrow S$ IS AN INDEPENDENT SET IFF V-S IS A VERTEX COVER.

INDEPENDENT SET $\leq_p$ VERTEX COVER
VERTEX COVER $\leq_p$ INDEPENDENT SET

INDEPENDENT SET AND VERTEX COVER GENERALIZATIONS

SET COVER IS A COVERING PROBLEM IN WHICH WE SEEK TO COVER A SET OF OBJECTS USING A COLLECTION OF SMALLER SETS.



VERTEX COVER $\leq_p$ SET COVER

THE SET PACKING PROBLEM IS A GENERALIZATION OF THE INDEPENDENT SET PROBLEM. HERE THE AIM IS TO PACK A LARGE NUMBER OF SETS TOGETHER, WITH THE CONSTRAINTS THAT NO TWO OF THEM ARE OVERLAPPING.

INDEPENDENT SET $\leq_p$ SET PACKING

WITH $Y \leq_p X$ WE CAN TRASFORM THE INSTANCE OF Y TO A SINGLE INSTANCE OF X, INVOKING X'S BLACK BOX ON THIS INSTANCE, AND REPORTING THE BOX'S ANSWER AS THE ANSWER FOR THE INSTANCE OF Y.

SAT PROBLEM

GIVEN A SET X OF n BOOLEAN VARIABLES $x_1, \dots x_n$, A TRUTH ASSIGNMENT FOR X IS AN ASSIGNMENT OF THE VALUE 0 OR 1 TO EACH x_i.

A CLAUSE IS A DISJUNCTION OF DISTINCT TERMS: $\mathcal{I}_1 \vee \mathcal{I}_2 \vee \mathcal{I}_3 \dots$

AN ASSIGNMENT SATISFIES A CLAUSE C IF IT CAUSES C TO EVALUATE TO 1 (AT LEAST ONE TERM = 1).
AN ASSIGNMENT SATISFIES A COLLECTION OF CLAUSES IF $C_1 \wedge C_2 \dots \wedge C_k = 1$.

> GIVEN A SET OF CLAUSES $C_1 \dots C_k$ OVER A SET OF VARIABLES $X = \{x_1 \dots x_n\}$, DOES THERE EXIST A SATISFYING THRUTH ASSIGNMENT?

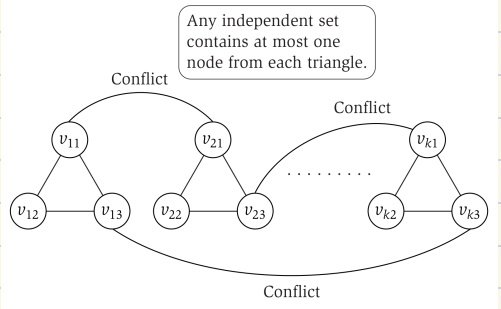
THERE IS A SPECIAL CASE OF SAT IN WHICH ALL CLAUSES CONTAIN EXACTLY 3 TERMS (3 SAT):

> GIVEN A SET OF CLAUSES $C_1 \dots C_k$, EACH OF LENGTH 3, OVER A SET OF VARIABLES $X = \{x_1 \dots x_n\}$, DOES THERE EXIST A SATISFYING THRUTH ASSIGN?

3-SAT REDUCTION

3-SAT $\leq_p$ INDEPENDENT SET

THE DIFFICULTY IS TO ENCODE ALL BOOLEAN CONSTRAINTS IN THE NODES AND EDGES OF A GRAPH, SO THAT SATISFIABILITY CORRESPONDS TO THE EXISTENCE OF A LARGE INDEP SET.



TWO TERMS CONFLICT IF ONE IS EQUAL TO x_i AND THE OTHER $\bar{x}_i$.
IF WE AVOID CONFLICTING TERMS, WE CAN FIND A TRUTH ASSIGNMENT THAT MAKES THE SELECTED TERMS FROM EACH CLAUSE EVALUATE TO 1.

THE ORIGINAL 3-SAT INSTANCE IS SATISFIABLE IFF THE GRAPH G HAS AN INDEPENDENT SET OF SIZE AT LEAST K.
IF THE 3-SAT INSTANCE IS SATISFIABLE, THEN EACH TRIANGLE IN G CONTAINS AT LEAST ONE NODE = 1 LET S BE A SET CONSISTING OF ONE SUCH NODE FROM EACH TRIANGLE, THEN S IS INDEPENDENT.

NP

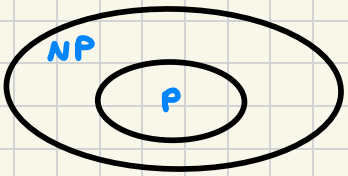
WE DON'T WANT TO FIND A GOOD ALGORITHM, BUT TO CHECK A PROPOSED SOLUTION IN POLYNOMIAL TIME.

IN ORDER TO CHECK A SOLUTION, WE NEED THE INPUT STRING s AND A SEPARATE CERTIFICATE t THAT CONTAINS THE EVIDENCE THAT s IS A "YES" INSTANCE OF X. Y IS AN EFFICIENT CERTIFIER FOR A PROBLEM X IF:

- Y IS A POLYNOMIAL ALGORITHM THAT TAKES TWO INPUT s AND t.
- THERE IS A POLYNOMIAL FUNCTION p SO THAT FOR EVERY STRING s, WE HAVE $s \in X$ IFF THERE EXISTS A STRING t: $|t| \leq p(|s|)$ AND $Y(s,t) = YES$.

WE DEFINE P AS THE SET OF ALL PROBLEMS X FOR WHICH THERE EXISTS AN ALGORITHM WITH A POLYNOMIAL RUNNING TIME THAT SOLVES X.
WE DEFINE NP AS THE SET OF ALL PROBLEMS FOR WHICH THERE EXISTS AN EFFICIENT CERTIFIER.

$P \subseteq NP$



- 3-SAT: THE CERTIFICATE t IS AN ASSIGNMENT OF TRUTH VALUES TO THE VARIABLES. THE CERTIFIER Y EVALUATES THE GIVEN SET OF CLAUSES WITH RESPECT TO THIS ASSIGNMENT.
- INDEPENDENT SET: THE CERTIFICATE t IS THE IDENTITY OF A SET OF AT LEAST K VERTICES. THE CERTIFIER Y CHECKS THAT, FOR THESE VERTICES, NO EDGE JOINS ANY PAIR OF THEM.
- SET COVER: THE CERTIFICATE t IS A LIST OF K SETS FROM THE GIVEN COLLECTION. THE CERTIFIER Y CHECKS THAT THE UNION OF THESE SETS IS EQUAL TO THE UNDERLAYING SET U.

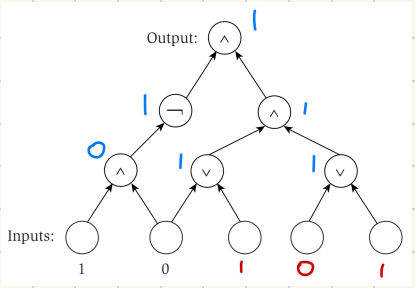
NP-COMPLETE

WHAT ARE THE HARDEST PROBLEMS IN NP?

i $X \in NP$
ii $\forall Y \in NP \cdot Y \leq_p X$ $\Rightarrow$ X IS NP COMPLETE

SUPPOSE X IS NP-COMPLETE. X IS SOLVABLE IN POLYNOMIAL TIME IFF $P = NP$ IMPOSSIBLE

CIRCUIT SATISFIABILITY



ANY ALGORITHM THAT TAKES A FIXED NUMBER n OF BITS AS INPUT AND PRODUCES A YES/NO OUTPUT CAN BE REPRESENTED BY A CIRWIT.

CIRCUIT SATISFIABILITY IS NP-COMPLETE

3-SAT INDEPENDENT SET SET PACKING VERTEX COVER SET COVER	NP-COMPLETE

NP-COMPLETE STRATEGY

GIVEN A NEW PROBLEM X:

1. PROVE THAT $X \in NP$
2. CHOOSE A PROBLEM Y THAT IS KNOW TO BE NP-COMPLETE
3. PROVE THAT $Y \leq_p X$

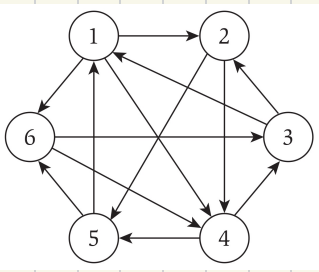
FOR 3. CONSIDER AN ARBITRARY INSTANCE S_Y OF THE PROBLEM Y, AND SHOW HOW TO CONSTRUCT, IN POLYNOMIAL TIME, AN INSTANCE S_X OF THE PROBLEM X THAT SATISFIES:

i. IF S_Y IS A YES INSTANCE OF Y, THEN S_X IS A YES INSTANCE OF X.
ii. IF S_X IS A YES INSTANCE OF X, THEN S_Y IS A YES INSTANCE OF Y.

S_X AND S_Y SHOULD HAVE THE SAME ANSWER.

THE HAMILTONIAN CYCLE PROBLEM

ANOTHER TYPE OF COMPUTATIONALLY HARD PROBLEMS INVOLVES SEARCHING OVER THE SET OF ALL PERMUTATIONS OF A COLLECTION.



GIVEN A DIRECTED GRAPH $G = (V, E)$, WE SAY THAT A CYCLE C IS A HAMILTONIAN CYCLE IF IT VISITS EACH NODE EXACTLY ONCE SO IT CONSTITUTES A TOUR WITHOUT REPETITIONS.

HAMILTONIAN CYCLE IS NP-COMPLETE

SUBSET SUM

GIVEN A SET $x_1 \dots x_n$, AND A TARGET W, IS THERE A SUBSET OF $\{x_1, \dots x_n\}$ THAT ADDS UP TO PRECISELY W?

SUBSET SUM IS NP-COMPLETE

GRAPH COLORING

WE SEEK TO ASSIGN A COLOR TO EACH NODE OF A GRAPH G SO THAT IN AN EDGE (u, v), u AND v ARE ASSIGNED DIFFERENT COLORS, WHILE USING A SMALL SET OF COLORS.

> GIVEN A GRAPH G AND A BOUND K. DOES G HAVE A K-COLORING?

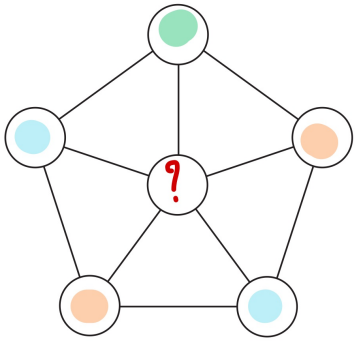


Figure 8.10 A graph that is not 3-colorable.

A GRAPH G IS 2-COLORING IFF IT'S BIPARTITE WITH 3 COLORS THING BECOME GETTING HARDER.

K-COLORING IS NP-COMPLETE $(k \geq 3)$

A 3-SAT INSTANCE IS SATISFIABLE IFF G' HAS A 3-COLORING.

WHEN $K > 3$ WE REDUCE THE 3-COLORING PROBLEM TO K-COLORING, BY TAKING AN INSTANCE OF 3-COLORING, REPRESENTED WITH A GRAPH, ADDING $K-3$ NODES AND JOINING THEM TO EACH OTHER AND TO EVERY NODE IN G. THE RESULTING GRAPH G' IS K-COLORABLE IFF G IS 3-COLORING.

SCHEDULING PROBLEMS

WE HAVE A SET OF n JOBS THAT MUST BE RUN ON A SINGLE MACHINE. EACH JOB i HAS A RELEASE TIME r_i WHEN IT'S FIRST AVAILABLE FOR PROCESSING, A DEADLINE d_i AND A DURATION t_i. IN ORDER TO BE COMPLETED, JOB i MUST BE ALLOCATED IN A SLOT OF t_i UNITS BETWEEN $[r_i, d_i]$. THE MACHINE CAN RUN ONLY ONE JOB AT A TIME.

> CAN WE SCHEDULE ALL JOBS SO THAT EACH COMPLETES BY ITS d_i?

SCHEDULING WITH r_i AND d_i IS NP-COMPLETE

CO-NP

WHEN WE HAVE A YES INSTANCE, THERE EXISTS A SHORT PROOF OF THIS, BUT WITH A NO INSTANCE WE CAN'T HAVE A PROOF.

$\forall$ PROBLEM X $\exists$ A COMPLEMENTARY $\overline{X}$. $\forall$ INPUT s. $s \in \overline{X}$ IFF $s \notin X$.
IF $X \in P \rightarrow \overline{X} \in P$. IF $X \in NP \rightarrow \overline{X} \in NP$?

A PROBLEM X BELONGS TO CO-NP IFF $\overline{X} \in NP$. NP = CO-NP? BOH

IF A PROBLEM $X \in NP \wedge$ CO-NP IT HAS A GOOD CHARACTERIZATION: WHEN THE ANSWER IS YES, THERE IS A SHORT PROOF, AND WHEN THE ANSWER IS NO, THERE IS A SHORT PROOF TOO.

EXERCISE

1. For each of the two questions below, decide whether the answer is (i) "Yes," (ii) "No," or (iii) "Unknown, because it would resolve the question of whether $\mathcal{P} = \mathcal{NP}$." Give a brief explanation of your answer.

(a) Let's define the decision version of the Interval Scheduling Problem from Chapter 4 as follows: Given a collection of intervals on a time-line, and a bound k, does the collection contain a subset of nonoverlapping intervals of size at least k?

Question: Is it the case that Interval Scheduling $\leq_P$ Vertex Cover?

(b) Question: Is it the case that Independent Set $\leq_P$ Interval Scheduling?

a) YES. INTERVAL SCHEDULING IS IN NP, AND ANYTHING IN NP CAN BE REDUCED TO VERTEX COVER.

b) P=NP. IF P=NP, THEN INDEPENDENT SET CAN BE SOLVED IN POL-TIME, AND SO INDEPENDENT SET $\leq_P$ INT SCHED. BUT INDEPENDENT SET IS NP-COMPLETE, SO SOLVING IT POL-TIME IMPLY P=NP.

2. A store trying to analyze the behavior of its customers will often maintain a two-dimensional array A, where the rows correspond to its customers and the columns correspond to the products it sells. The entry $A[i, j]$ specifies the quantity of product j that has been purchased by customer i.

Here's a tiny example of such an array A.

	liquid detergent	beer	diapers	cat litter
Raj	0	6	0	3
Alanis	2	3	0	0
Chelsea	0	0	0	7

One thing that a store might want to do with this data is the following. Let us say that a subset S of the customers is *diverse* if no two of the of the customers in S have ever bought the same product (i.e., for each product, at most one of the customers in S has ever bought it). A diverse set of customers can be useful, for example, as a target pool for market research.

We can now define the Diverse Subset Problem as follows: Given an $m \times n$ array A as defined above, and a number $k \leq m$, is there a subset of at least k of customers that is *diverse*?

Show that Diverse Subset is NP-complete.

THE PROBLEM IS IN NP BECAUSE WE CAN EXHIBIT A SET OF K CUSTOMERS, AND IN POL-TIME WE CAN CHECK THAT NO TWO BOUGHT ANY PRODUCT IN COMMON.

INDEPENDENT SET $\leq_P$ DIVERSE

GIVEN A GRAPH G AND A NUMBER K, WE CONSTRUCT A CUSTOMER FOR EACH NODE OF G, AND A PRODUCT FOR EACH EGE. WE BUILD AN ARRAY THAT SAYS: CUSTOMER U BOUGHT PRODUCT E IF EGE E IS INCIDENT TO U, AND WE ASK WHETHER THIS ARRAY HAS A DIVERSE SUBSET OF SIZE K. THIS HOLDS IFF G HAS AN INDEPENDENT SET OF SIZE K. IF THERE IS A DIVERSE SUBSET OF SIZE K, THEN THE CORRESPONDING SET OF NODES HAS THE PROPERTY THAT NO TWO ARE INCIDENT TO THE SAME EDGE.

3. Suppose you're helping to organize a summer sports camp, and the following problem comes up. The camp is supposed to have at least one counselor who's skilled at each of the n sports covered by the camp (baseball, volleyball, and so on). They have received job applications from m potential counselors. For each of the n sports, there is some subset of the m applicants qualified in that sport. The question is: For a given number $k < m$, is it possible to hire at most k of the counselors and have at least one counselor qualified in each of the n sports? We'll call this the *Efficient Recruiting Problem*.

Show that Efficient Recruiting is NP-complete.

IT'S NP SINCE, GIVEN A SET OF K COUNSELORS, WE CAN CHECK THAT THEY COVER ALL THE SPORTS.
WE CAN USE VERTEX COVER. GIVEN A GRAPH G AND A K, WE DEFINE A SPORT S_e FOR EACH EDGE, AND A COUNSELOR C_u FOREACH NODE U. C_u IS QUALIFIED IN S_e IFF E HAS AN END EQUAL TO U. THUS, G HAS A VERTEX COVER OF SIZE AT MOST K IFF THE INSTANCE OF EFFICIENT RECRUITING THAT WE CREATE CAN BE SOLVED WITH AT MOST K COUNSELORS.

4. Suppose you're consulting for a group that manages a high-performance real-time system in which asynchronous processes make use of shared resources. Thus the system has a set of n *processes* and a set of m *resources*. At any given point in time, each process specifies a set of resources that it requests to use. Each resource might be requested by many processes at once; but it can only be used by a single process at a time. Your job is to allocate resources to processes that request them. If a process is allocated all the resources it requests, then it is *active*; otherwise it is *blocked*. You want to perform the allocation so that as many processes as possible are active. Thus we phrase the *Resource Reservation Problem* as follows: Given a set of processes and resources, the set of requested resources for each process, and a number k, is it possible to allocate resources to processes so that at least k processes will be active?

Consider the following list of problems, and for each problem either give a polynomial-time algorithm or prove that the problem is NP-complete.

(a) The general Resource Reservation Problem defined above.

(b) The special case of the problem when $k = 2$.

(c) The special case of the problem when there are two types of resources—say, people and equipment—and each process requires at most one resource of each type (In other words, each process requires one specific person and one specific piece of equipment.)

(d) The special case of the problem when each resource is requested by at most two processes.

a) IT'S NP SINCE, GIVEN A SET OF K PROCESSES, WE CAN CHECK IN POL-TIME THAT NO RESOURCE IS REQUESTED BY MORE THAN ONE.

INDEPENDENT SET $\leq_p$ RR

GIVEN A GRAPH G AND K, WE CREATE AN INSTANCE RR EQUIVALENT TO AN INSTANCE OF IND SET. THE PROCESSES ARE NODES, WHILE RESOURCES ARE EDGES.
IF THERE ARE K PROCESSES WHOSE REQUESTED RESOURCES ARE DISJOINT, THEN THE K NODES OF THIS PROCESSES FORM AN IND SET.

b) WE CAN SOLVE THIS BY BRUTE FORCE. WE JUST TRY ALL $O(n^2)$ PAIRS OF PROCESSES AND WE SEE WETHER ANY PAIR HAS DISJOINT RESOURCE REQUIREMENTS.

c) IT'S A BIPARTITE MATCHING PROBLEM. WE DEFINE A NODE FOREACH PERSON AND EQUIPMENT, AND EACH PROCESS IS AN EDGE (PERSON, EQUIPMENT). A SET OF PROCESSES WITH DISJOINT RESOURCE REQUIREMENTS IS A SET OF EDGES WITH DISJOINT ENDS (A MATCHING).

d) NP-COMPLETE SINCE IT'S THE SAME AS a. EACH EDGE (RESOURCE) IS REQUESTED ONLY BY THE TWO NODES (PROCESSES) THAT FORM ITS ENDS.

5. Consider a set $A = \{a_1, \ldots, a_n\}$ and a collection $B_1, B_2, \ldots, B_m$ of subsets of A (i.e., $B_i \subseteq A$ for each i).

We say that a set $H \subseteq A$ is a *hitting set* for the collection $B_1, B_2, \ldots, B_m$ if H contains at least one element from each B_i—that is, if $H \cap B_i$ is not empty for each i (so H "hits" all the sets B_i).

We now define the *Hitting Set Problem* as follows. We are given a set $A = \{a_1, \ldots, a_n\}$, a collection $B_1, B_2, \ldots, B_m$ of subsets of A, and a number k. We are asked: Is there a hitting set $H \subseteq A$ for $B_1, B_2, \ldots, B_m$ so that the size of H is at most k?

Prove that Hitting Set is NP-complete.

IT'S NP SINCE, GIVEN A SET H, WE CAN CHECK IN POL-TIME THAT H HAS SIZE OF AT MOST K

VERTEX COVER $\leq_p$ HITTING SET

GIVEN A GRAPH G AND K, WE CREATE AN INSTANCE OF HITTNG EQUIVALENT TO AN INSTANCE OF VERTEX COVER. IN VERTEX COVER, WE ARE TRYING TO CHOOSE AT MOST K NODES TO FORM A VERTEX COVER. IN HITTING SET, WE ARE TRYING TO CHOOSE AT MOST K ELEMENTS TO FORM A HITTING SET. SO WE DEFINE THE SET A IN HITTING SET INSTANCE TO BE THE V OF NODES IN VERTEX COVER INSTANCE. THERE IS A HITTING SET OF SIZE AT MOST K IFF G HAS A VERTEX COVER OF SIZE AT MOST K.

6. Consider an instance of the Satisfiability Problem, specified by clauses $C_1, \ldots, C_k$ over a set of Boolean variables $x_1, \ldots, x_n$. We say that the instance is *monotone* if each term in each clause consists of a nonnegated variable; that is, each term is equal to x_i, for some i, rather than $\overline{x_i}$. Monotone instances of Satisfiability are very easy to solve: They are always satisfiable, by setting each variable equal to 1.

For example, suppose we have the three clauses

$$(x_1 \vee x_2), (x_1 \vee x_3), (x_2 \vee x_3).$$

This is monotone, and indeed the assignment that sets all three variables to 1 satisfies all the clauses. But we can observe that this is not the only satisfying assignment; we could also have set x_1 and x_2 to 1, and x_3 to 0. Indeed, for any monotone instance, it is natural to ask how few variables we need to set to 1 in order to satisfy it.

Given a monotone instance of Satisfiability, together with a number k, the problem of *Monotone Satisfiability with Few True Variables* asks: Is there a satisfying assignment for the instance in which at most k variables are set to 1? Prove this problem is NP-complete.

VERTEX COVER $\leq_p$ MONSAT

GIVEN A GRAPH G AND K, WE CREATE AN INSTANCE OF MONSAT EQUIVALENT TO AN INSTANCE OF VERTEX COVER. WE HAVE A VARIABLE x_i FOR EACH VERTEX v_i. FOR EACH EDGE $e_j = (v_a, v_b)$ WE CREATE THE CLAUSE $C_j = (x_a \vee x_b)$. SO THERE ARE $C_1 \ldots C_m$ CLAUSES, ONE FOR EACH EDGE, AND WE WANT TO KNOW IF THEY CAN ALL BE SATISFIED BY SETTING AT MOST K VARIABLES TO 1
THERE IS A VERTEX COVER WITH AT MOST K VARIABLES SETTED TO 1 IFF THERE IS A MONSAT WITH AT MOST K VARIABLES SETTED TO 1.

7. Since the 3-Dimensional Matching Problem is NP-complete, it is natural to expect that the corresponding 4-Dimensional Matching Problem is at least as hard. Let us define *4-Dimensional Matching* as follows. Given sets W, X, Y, and Z, each of size n, and a collection C of ordered 4-tuples of the form (w_i, x_j, y_k, z_ℓ), do there exist n 4-tuples from C so that no two have an element in common?

Prove that 4-Dimensional Matching is NP-complete.

4D IS NP SINCE, GIVEN A SET OF n 4-TUPLES, WE CAN CHECK THAT ARE DISJOINT.

3D $\leq_p$ 4D

WE DEFINE A COLLECTION C' OF 4-TUPLES SO THAT FOR EVERY $(x_j, y_k, z_\ell) \in C$, AND EVERY $1 \leq i \leq n$, THERE IS A 4-TUPLE (w_i, x_j, y_k, z_ℓ). IF $A = (x_j, y_k, z_\ell)$ IS A TRIPLE IN C, WE DEFINE $f(A) \in C'$ TO BE THE 4-TUPLE (w_i, x_j, y_k, z_ℓ). IF $B = (w_i, x_j, y_k, z_\ell) \in C'$, $f'(B) = (x_j, y_k, z_\ell) \in C$. GIVEN A SET OF n DISJOINT TRIPLES A_i IN C, $f(A_i)$ IS A SET OF n 4-TUPLES IN C'.

8. Your friends' preschool-age daughter Madison has recently learned to spell some simple words. To help encourage this, her parents got her a colorful set of refrigerator magnets featuring the letters of the alphabet (some number of copies of the letter A, some number of copies of the letter B, and so on), and the last time you saw her the two of you spent a while arranging the magnets to spell out words that she knows.

Somehow with you and Madison, things always end up getting more elaborate than originally planned, and soon the two of you were trying to spell out words so as to use up all the magnets in the full set—that is, picking words that she knows how to spell, so that once they were all spelled out, each magnet was participating in the spelling of exactly one of the words. (Multiple copies of words are okay here; so for example, if the set of refrigerator magnets includes two copies each of C, A, and T, it would be okay to spell out CAT twice.)

This turned out to be pretty difficult, and it was only later that you realized a plausible reason for this. Suppose we consider a general version of the problem of *Using Up All the Refrigerator Magnets*, where we replace the English alphabet by an arbitrary collection of symbols, and we model Madison's vocabulary as an arbitrary set of strings over this collection of symbols. The goal is the same as in the previous paragraph.

Prove that the problem of Using Up All the Refrigerator Magnets is NP-complete.

THIS IS IN NP.

3D $\leq_p$ MAG

AN INSTANCE OF 3D CONSISTS OF 3 SETS $|X| = |Y| = |Z| = n$, AND A SET OF TUPLES M. WE WANT TO FIND n TUPLES FROM M S.T. EACH ELEMENT IS COVERED BY EXACTLY ONE TUPLE.
AN INSTANCE OF MAG CONSISTS OF: EACH ELEMENT X, Y OR Z BECOMES A MAGNET WITH A UNIQUE LETTER (3n LETTERS), AND EVERY TUPLE (x_i, y_j, z_k) BECOMES A WORD THAT MADISON KNOWS.
THERE IS A PERFECT MATCHING IN 3D IFF ALL THE MAGNETS ARE USED.

9. Consider the following problem. You are managing a communication network, modeled by a directed graph $G = (V, E)$. There are c *users* who are interested in making use of this network. User i (for each $i = 1, 2, \ldots, c$) issues a *request* to reserve a specific path P_i in G on which to transmit data.

You are interested in accepting as many of these path requests as possible, subject to the following restriction: if you accept both P_i and P_j, then P_i and P_j cannot share any nodes.

Thus, the *Path Selection Problem* asks: Given a directed graph $G = (V, E)$, a set of requests $P_1, P_2, \ldots, P_c$—each of which must be a path in G—and a number k, is it possible to select at least k of the paths so that no two of the selected paths share any nodes?

Prove that Path Selection is NP-complete.

THIS IS IN NP SINCE WE CAN EASILY CHECK A GIVEN SOLUTION.

3D $\leq_p$ PATH

WE CONSTRUCT A DIRECT GRAPH G ON THE NODE SET $X \cup Y \cup Z$. FOR EACH $T_i = (x_i, y_j, z_k)$ WE ADD EDGES (x_i, y_j) AND (y_j, z_k) TO G. FOR EACH i WE DEFINE A PATH P_i THAT PASSES THROUGH THE NODES T_i. THERE ARE PATHS AMONG $P_1 \ldots P_m$ SHARING NO NODES IFF THERE EXIST n DISJOINT TRIPLES AMONG $T_1 \ldots T_m$.

10. Your friends at WebExodus have recently been doing some consulting work for companies that maintain large, publicly accessible Web sites—contractual issues prevent them from saying which ones—and they've come across the following *Strategic Advertising Problem*.

A company comes to them with the map of a Web site, which we'll model as a directed graph $G = (V, E)$. The company also provides a set of t *trails* typically followed by users of the site; we'll model these trails as directed paths $P_1, P_2, \ldots, P_t$ in the graph G (i.e., each P_i is a path in G).

The company wants WebExodus to answer the following question for them: Given G, the paths $\{P_i\}$, and a number k, is it possible to place advertisements on at most k of the nodes in G, so that each path P_i includes at least one node containing an advertisement? We'll call this the Strategic Advertising Problem, with input G, $\{P_i : i = 1, \ldots, t\}$, and k.

Your friends figure that a good algorithm for this will make them all rich; unfortunately, things are never quite this simple.

(a) Prove that Strategic Advertising is NP-complete.

(b) Your friends at WebExodus forge ahead and write a pretty fast algorithm $\mathcal{S}$ that produces yes/no answers to arbitrary instances of the Strategic Advertising Problem. You may assume that the algorithm $\mathcal{S}$ is always correct.

Using the algorithm $\mathcal{S}$ as a black box, design an algorithm that takes input G, $\{P_i\}$, and k as in part (a), and does one of the following two things:

- Outputs a set of at most k nodes in G so that each path P_i includes at least one of these nodes, *or*
- Outputs (correctly) that no such set of at most k nodes exists.

Your algorithm should use at most a polynomial number of steps, together with at most a polynomial number of calls to the algorithm $\mathcal{S}$.

a) IT'S IN NP SINCE WE CAN EASILY CHECK A GIVEN SOLUTION.

VERTEX COVER $\leq_p$ SA

GIVEN AN UNDIRECTED GRAPH $G = (V, E)$ AND A NUMBER K, PRODUCE A DIRECTED GRAPH $G' = (V, E')$ BY DIRECTING EACH EDGE OF G. THEN DEFINE A PATH P_i FOR EACH EDGE IN E'. G' HAS A VALID SET OF AT MOST K ADV IFF G HAS A VERTEX COVER OF SIZE AT MOST K.

b)

APPROXIMATION ALGORITHMS

THESE ALGORITHMS RUN IN POLYNOMIAL-TIME AND FIND SOLUTIONS THAT ARE CLOSE TO THE OPTIMAL, FOR PROBLEMS WHERE POL-TIME IS IMPOSSIBLE.

THERE ARE FOUR GENERAL TECHNIQUE:

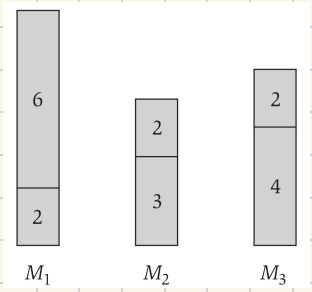
- GREADY ALGORITHMS: FIND A GREADY RULE THAT LEADS TO A SOLUTION CLOSE TO THE OPTIMAL
- PRICING METHOD: WE CONSIDER A PRICE ONE HAS TO PAY TO ENFORCE EACH CONSTRAINT OF THE PROBLEM.
- LINEAR PROGRAMMING AND ROUNDING: WE EXPLOITS THE RELATION BETWEEN THE COMPUTATIONAL FEASIBILITY OF LP AND THE EXPRESSIVE POWER OF INTEGER PROGRAMMING.
- DYNAMIC PROGRAMMING OF A ROUNDED VERSION OF THE INPUT.

LOAD BALANCING WITH GREEDY

GIVEN A SET OF m MACHINES $M_1, \dots M_m$ AND A SET OF n JOBS (EACH JOB HAS A PROCESSING TIME t_j), WE SEEK TO ASSIGN EACH JOB TO ONE OF THE MACHINES SO THAT THE LOADS PLACED ON ALL MACHINES ARE AS BALANCED AS POSSIBLE

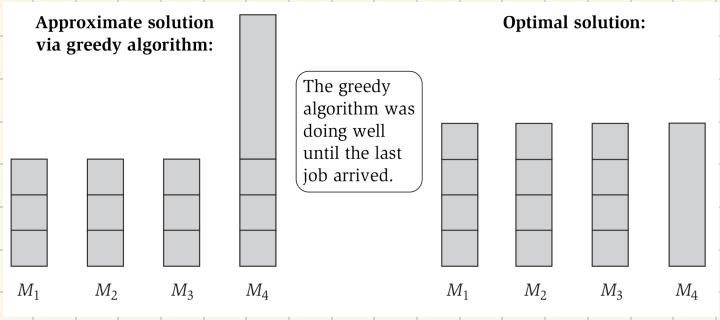
$A(i)$ DENOTE THE SET OF JOBS ASSIGNED TO MACHINE M_i AND M_i NEEDS TO WORK FOR A TOTAL TIME OF $T_i = \sum_{j \in A(i)} t_j$.

```
Greedy-Balance:
Start with no jobs assigned
Set T_i = 0 and A(i) = ∅ for all machines M_i
For j = 1, ..., n
  Let M_i be a machine that achieves the minimum min_k T_k
  Assign job j to machine M_i
  Set A(i) ← A(i) ∪ {j}
  Set T_i ← T_i + t_j
EndFor
```



IT GOES ON JOBS ORDER, AND IT ASSIGNS EACH JOB j TO THE MACHINE WITH THE SMALLEST LOAD.

BUT SOMETIME IS NOT SO GOOD



WE CAN IMPROV IT BY FIRST SORTING THE JOBS IN DECREASING ORDER OF PROCESSING TIME AND THEN PROCEDING AS BEFORE.

```
Sorted-Balance:
Start with no jobs assigned
Set T_i = 0 and A(i) = ∅ for all machines M_i
Sort jobs in decreasing order of processing times t_j
Assume that t_1 ≥ t_2 ≥ ... ≥ t_n
For j = 1, ..., n
  Let M_i be the machine that achieves the minimum min_k T_k
  Assign job j to machine M_i
  Set A(i) ← A(i) ∪ {j}
  Set T_i ← T_i + t_j
EndFor
```

SET COVER

GIVEN A SET U OF n ELEMENTS AND A LIST $S_1, \dots S_m$ OF SUBSETS OF U, WE SAY THAT A SET COVER IS A COLLECTION OF THESE SETS WHOSE UNION IS EQUAL TO ALL OF U
IN THIS VERSION EACH SET S_i HAS AN ASSOCIATED WEIGHT $w_i \geq 0$. THE GOAL IS TO FIND A SET COVER C TO MINIMIZE $\sum_{S_i \in C} w_i$.

```
Greedy-Set-Cover:
Start with R = U and no sets selected
While R ≠ ∅
  Select set S_i that minimizes w_i/|S_i ∩ R|
  Delete set S_i from R
EndWhile
Return the selected sets
```

IT COVERS ONE SET AT A TIME. TO CHOOSE ITS NEXT SET, IT LOOKS FOR ONE THAT SEEMS TO MAKE THE MOST PROGRESS TOWARD THE GOAL.

PROGRESS = $w_i / |S_i|$ → BY SELECTING S_i, WE COVER $|S_i|$ ELEMENTS AT A COST OF w_i, AND SO THIS RATIO GIVES THE COST PER ELEMENT COVERED

WE WILL MAINTAIN THE SET R OF REMAINING UNCOVERED ELEMENTS AND CHOOSE THE SET S_i THAT MINIMIZE $w_i / |S_i \cap R|$

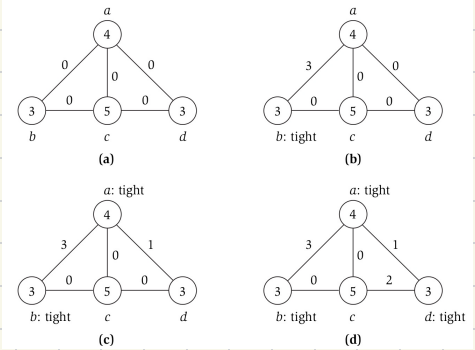
VERTEX COVER - PRICING METHOD

A VERTEX COVER IN A GRAPH $G=(V,E)$, IS A SET $S \subseteq V$ SO THAT EACH EDGE HAS AT LEAST ONE END IN S.
IN THIS VERSION EACH EDGE VERTEX $i \in V$ HAS A WEIGHT $w_i \geq 0$ AND $w(s) = \sum_{i \in S} w_i$. THE GOAL IS TO FIND A VERTEX COVER S MINIMIZING $w(s)$.

VERTEX COVER $\leq_p$ SET COVER BUT IT'S NOT GOOD TO USE SET COVER'S ALG.

WE WILL THINK w_i AS THE COST OF USING VERTEX i IN THE COVER.
WE WILL THINK EDGE e AS AN AGENT WHO IS WILLING TO PAY SOMETHING TO THE NODE THAT COVERS IT. THE ALG WILL NOT ONLY TO FIND A VERTEX COVER S, BUT ALSO DETERMINE PRICES $p_e \geq 0$ FOR EACH EDGE e. THE ALG USES THIS PRICES TO DRIVE THE WAY IT SELECTS NODES.
SELECTING A VERTEX i COVERS ALL INCIDENT EDGES, SO IT WOULD BE UNFAIR TO CHARGE THESE EDGES IN TOTAL MORE THAN THE COST OF VERTEX i. SO PRICES p_e ARE FAIR IF, FOR EACH VERTEX i, THE EDGES ADJACENT TO i DON'T HAVE TO PAY MORE THAN THE COST OF VERTEX: $\sum_{e(i,j)} p_e \leq w_i$

```
Vertex-Cover-Approx(G, w):
  Set p_e = 0 for all e ∈ E
  While there is an edge e = (i, j) such that neither i nor j is tight
     Select such an edge e
     Increase p_e without violating fairness
  EndWhile
  Let S be the set of all tight nodes
  Return S
```



LINEAR PROGRAMMING AND ROUNDING

GIVEN A REGION DEFINED BY $Ax \geq b$, LP SEEKS TO MINIMIZE A LINEAR COMBINATION $c^T x$, WHERE c IS A VECTOR OF COEFFICIENTS, AND $c^T x$ DENOTES THE INNER PRODUCT OF TWO VECTORS

> GIVEN AN $m \times n$ MATRIX A, AND VECTORS $b \in R^m$ AND $c \in R^n$, FIND A VECTOR $x \in R^n$ TO SOLVE THIS OPTIMIZATION PROBLEM:
>
> $$\min (c^T x \text{ s.t. } x \geq 0; Ax \geq b)$$

> GIVEN A MATRIX A, VECTORS c AND b, AND A BOUND y, DOES THERE EXISTS x SO THAT $x \geq 0$, $Ax \geq b$ AND $c^T x \leq y$?

EXERCISE

1. Suppose you're acting as a consultant for the Port Authority of a small Pacific Rim nation. They're currently doing a multi-billion-dollar business per year, and their revenue is constrained almost entirely by the rate at which they can unload ships that arrive in the port.

Here's a basic sort of problem they face. A ship arrives, with n containers of weight $w_1, w_2, \ldots, w_n$. Standing on the dock is a set of trucks, each of which can hold K units of weight. (You can assume that K and each w_i is an integer.) You can stack multiple containers in each truck, subject to the weight restriction of K; the goal is to minimize the number of trucks that are needed in order to carry all the containers. This problem is NP-complete (you don't have to prove this).

A greedy algorithm you might use for this is the following. Start with an empty truck, and begin piling containers $1, 2, 3, \ldots$ into it until you get to a container that would overflow the weight limit. Now declare this truck "loaded" and send it off; then continue the process with a fresh truck. This algorithm, by considering trucks one at a time, may not achieve the most efficient way to pack the full set of containers into an available collection of trucks.

(a) Give an example of a set of weights, and a value of K, where this algorithm does not use the minimum possible number of trucks.

(b) Show, however, that the number of trucks used by this algorithm is within a factor of 2 of the minimum possible number, for any set of weights and any value of K.

a) $n = 4 \quad w_i = \{2, 5, 3, 4\} \quad K = 5$

THE ALG WOULD USE 4 TRUCK, ONE FOR EACH CONTAINER, WHILE OPT IS 3 TRUCKS.

b) LET $W = \sum_i w_i$. SINCE EACH TRUCK HOLDS AT MOST K UNITS, W/K IS A LOWER BOUND ON THE NUMBER OF TRUCK NEEDED.

SUPPOSE THE NUMBER OF TRUCK USED BY THE GREEDY IS ODD $m = 2q+1$. WE DIVIDE THE TRUCKS USED INTO CONSECUTIVE GROUPS OF TWO, FOR A TOTAL OF $q+1$ GROUPS. IN EACH GROUP, BUT THE LAST, THE TOTAL WEIGHT OF CONTAINERS MUST BE STICTLY GREATER THAN K. SO $W > qK$, $W/K > q$. OPT USES AT LEAST $q+1$ TRUCKS, WHICH IS WITHIN A FACTOR OF 2 OF $m = 2q+1$.

2. At a lecture in a computational biology conference one of us attended a few years ago, a well-known protein chemist talked about the idea of building a "representative set" for a large collection of protein molecules whose properties we don't understand. The idea would be to intensively study the proteins in the representative set and thereby learn (by inference) about all the proteins in the full collection.

To be useful, the representative set must have two properties.

- It should be relatively small, so that it will not be too expensive to study it.
- Every protein in the full collection should be "similar" to some protein in the representative set. (In this way, it truly provides some information about all the proteins.)

More concretely, there is a large set P of proteins. We define similarity on proteins by a *distance function* d: Given two proteins p and q, it returns a number $d(p, q) \geq 0$. In fact, the function $d(\cdot, \cdot)$ most typically used is the *sequence alignment* measure, which we looked at when we studied dynamic programming in Chapter 6. We'll assume this is the distance being used here. There is a predefined distance cut-off Δ that's specified as part of the input to the problem; two proteins p and q are deemed to be "similar" to one another if and only if $d(p, q) \leq \Delta$.

We say that a subset of P is a *representative set* if, for every protein p, there is a protein q in the subset that is similar to it—that is, for which $d(p, q) \leq \Delta$. Our goal is to find a representative set that is as small as possible.

(a) Give a polynomial-time algorithm that approximates the minimum representative set to within a factor of $O(\log n)$. Specifically, your algorithm should have the following property: If the minimum possible size of a representative set is s^*, your algorithm should return a representative set of size at most $O(s^* \log n)$.

a) FOR EACH PROTEIN p WE DEFINE A SET S_p, CONSISTING OF ALL PROTEINS SIMILIAR TO IT, BY ENUMERATING ALL PROTEINS q FOR WHICH $d(p,q) \leq \Delta$. A REPRESENTATIVE SET $R \subseteq P$ IS A SET FOR WHICH $\{S_p : p \in R\}$ IF A SET COVER FOR P. TO APPROXIMATE THE SIZE OF THE SMALLEST REPRESENTATIVE SET, WE CAN USE THE APPR-ALG FOR SET COVER.

3. Suppose you are given a set of positive integers $A = \{a_1, a_2, \ldots, a_n\}$ and a positive integer B. A subset $S \subseteq A$ is called *feasible* if the sum of the numbers in S does not exceed B:

$$\sum_{a_i \in S} a_i \leq B.$$

The sum of the numbers in S will be called the *total sum* of S.

You would like to select a feasible subset S of A whose total sum is as large as possible.

Example. If $A = \{8, 2, 4\}$ and $B = 11$, then the optimal solution is the subset $S = \{8, 2\}$.

(a) Here is an algorithm for this problem.

```
Initially S = φ
Define T = 0
For i = 1, 2, ..., n
  If T + a_i ≤ B then
    S ← S ∪ {a_i}
    T ← T + a_i
  Endif
Endfor
```

Give an instance in which the total sum of the set S returned by this algorithm is less than half the total sum of some other feasible subset of A.

(b) Give a polynomial-time approximation algorithm for this problem with the following guarantee: It returns a feasible set $S \subseteq A$ whose total sum is at least half as large as the maximum total sum of any feasible set $S' \subseteq A$. Your algorithm should have a running time of at most $O(n \log n)$.

a) $a_1 = 2 \quad a_2 = 50 \quad B = 50$

THE ALG WOULD CHOOSE $\{a_1\}$, WHILE OPT: $\{a_2\}$

b) WE FIRST GO THROUGH ALL a_i AND DELETE ANY WHOSE VALUE EXCEEDS B (USELESS). WE THEN GO THROUGH ALL a_i UNTIL THE SUM EXCEEDS B. LET a_j = NUMBER ON WHICH THIS HAPPENS. WE HAVE THAT

$$\sum_{i=1}^{j} a_i \geq B \quad \sum_{i=1}^{j-1} a_i \leq B \rightarrow a_j \leq B$$

THUS THE SET $\{a_1 \ldots a_{j-1}\}$ OR $\{a_j\}$ IS AT LEAST B/2 AND AT MOST B.

RANDOMIZED ALGORITHMS

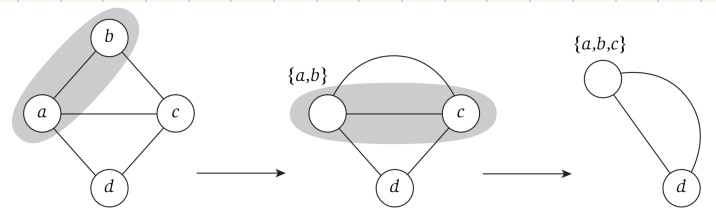
DESIGN RANDOMIZED ALG MAKE OUR MODEL POWERFUL

FINDING GLOBAL MIN CUT

GIVEN AN UNDIRECTED GRAPH $G=(V,E)$, WE DEFINE A CUT OF G TO BE A PARTITION OF V INTO TWO NON EMPTY SETS A AND B. THE OF THE CUT (A,B) IS THE NUMBER OF EDGES WITH ONE END IN A AND THE OTHER IN B. A **GLOBAL MIN CUT** IS A CUT WITH MINIMUM SIZE.

THE CONTRACTION ALG WORKS WITH A CONNECTED MULTIGRAPH $G=(V,E)$, AN UNDIRECTED GRAPH THAT CAN HAVE MULTIPLE PARALLEL EDGES BETWEEN THE SAME PAIR OF NODES.
IT BEGINS BY RANDOMLY CHOOSING AN EDGE $e=(u,v)$ AND CONTRACTING IT. WE PRODUCE A NEW G' IN WHICH u AND v BECOME A SINGLE NODE w, THE EDGE $e=(u,v)$ IS DELETED, AND EACH OTHER EDGE WITH AN END IN u OR v IS UPDATED WITH AN END IN w.
THE ALG TERMINATES WHEN THERE ARE ONLY 2 SUPERNODES v_1 AND v_2. THE SUBSETS $S(v_1)$ AND $S(v_2)$ FORM A PARTITION OF V, AND SO $(S(v_1), S(v_2))$ IS THE CUT RESULT.

```
The Contraction Algorithm applied to a multigraph G = (V, E):
  For each node v, we will record
        the set S(v) of nodes that have been contracted into v
      Initially S(v) = {v} for each v
  If G has two nodes v1 and v2, then return the cut (S(v1), S(v2))
  Else choose an edge e = (u, v) of G uniformly at random
      Let G' be the graph resulting from the contraction of e,
          with a new node z_uv replacing u and v
      Define S(z_uv) = S(u) ∪ S(v)
      Apply the Contraction Algorithm recursively to G'
  Endif
```



THE ALG RETURNS A GLOBAL MIN CUT WITH PROBABILITY AT LEAST $1/\binom{n}{2}$, AND AN UNDIRECTED G WITH n NODES HAS AT MOST $\binom{n}{2}$ GLOBAL MIN CUTS.

MAX 3-SAT

IF WE CAN'T FIND A TRUTH ASSIGNMENT THAT SATISFIES ALL CLAUSES, WE CAN TURN THE 3-SAT INSTANCE INTO AN OPTIMIZATION PROBLEM:

> GIVEN THE SET OF INPUT CLAUSES $C_1 \dots C_k$ FIND A TRUTH ASSIGNMENT THAT SATISFIES AS MANY AS POSSIBLE.

SUPPOSE WE SET EACH x_i TO 0 OR 1 WITH PROBABILITY $\frac{1}{2}$. WHAT IS THE EXPECTED NUMBER OF CLAUSES SATISFIED RANDOMLY?
LET Z DENOTE THE SUM OF THE RANDOM VARIABLES, $Z_i = 1$ IF $C_i = 1$ OR 0 OTHERWISE, SO $Z = Z_1 \dots Z_k$.
$E[Z_i]$ IS THE PROBABILITY THAT C_i IS SATISFIED:

$$E[Z] = E[Z_1] + E[Z_2] + \dots E[Z_k] = 1 - \left(\frac{1}{2}\right)^3 = \frac{7}{8}k$$

FOR EVERY INSTANCE OF 3-SAT, THERE IS A TRUTH ASSIGNMENT THAT SATISFIES AT LEAST A $7/8$ OF ALL CLAUSES

EXERCISE

1. *3-Coloring* is a yes/no question, but we can phrase it as an optimization problem as follows.

Suppose we are given a graph $G = (V, E)$, and we want to color each node with one of three colors, even if we aren't necessarily able to give different colors to every pair of adjacent nodes. Rather, we say that an edge (u, v) is *satisfied* if the colors assigned to u and v are different.

Consider a 3-coloring that maximizes the number of satisfied edges, and let c^* denote this number. Give a polynomial-time algorithm that produces a 3-coloring that satisfies at least $\frac{2}{3}c^*$ edges. If you want, your algorithm can be randomized; in this case, the *expected* number of edges it satisfies should be at least $\frac{2}{3}c^*$.

WE NEED AN UPPER BOUND: $c^* \le m = |E|$

WE COLOR EVERY NODE WITH PROB $1/3$

$$X_e = \begin{cases} 1 & \text{EDGE } e \text{ IS SATISFIED} \\ 0 & \text{OTHERWISE} \end{cases}$$

FORALL EDGE, THERE ARE 9 WAYS TO COLOR ITS TWO ENDS, WITH SAME PROB, AND 3 OF THEM ARE NOT SATISFYING

Y = # OF SATISFIED EDGES

$$E[X_e] = Pr[e \text{ IS SAT}] = 6/9 = 2/3$$

$$E[Y] = E\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} E[X_e] = \frac{2}{3}m \ge \frac{2}{3}c^*$$

2. Consider a county in which 100,000 people vote in an election. There are only two candidates on the ballot: a Democratic candidate (denoted D) and a Republican candidate (denoted R). As it happens, this county is heavily Democratic, so 80,000 people go to the polls with the intention of voting for D, and 20,000 go to the polls with the intention of voting for R.

However, the layout of the ballot is a little confusing, so each voter, independently and with probability $\frac{1}{100}$, votes for the wrong candidate—that is, the one that he or she *didn't* intend to vote for. (Remember that in this election, there are only two candidates on the ballot.)

Let X denote the random variable equal to the number of votes received by the Democratic candidate D, when the voting is conducted with this process of error. Determine the expected value of X, and give an explanation of your derivation of this value.

$$X_i = \begin{cases} 0 & \text{VOTE R} \\ 1 & \text{VOTE D} \end{cases} \qquad X = \sum_{i=1}^{100000} X_i$$

FOR $i \le 20000$

$$E[X_i] = 0.99 \cdot 0 + 0.01 \cdot 1 = 0.01$$

FOR $i > 20000$

$$E[X_i] = 0.01 \cdot 0 + 0.99 \cdot 1 = 0.99$$

$$E[X] = \sum_{i=1}^{100000} E[X_i] = 20000 \cdot 0.01 + 80000 \cdot 0.99 = 79400$$

7. In Section 13.4, we designed an approximation algorithm to within a factor of 7/8 for the MAX 3-SAT Problem, where we assumed that each clause has terms associated with three different variables. In this problem, we will consider the analogous MAX SAT Problem: Given a set of clauses $C_1, \ldots, C_k$ over a set of variables $X = \{x_1, \ldots, x_n\}$, find a truth assignment satisfying as many of the clauses as possible. Each clause has at least one term in it, and all the variables in a single clause are distinct, but otherwise we do not make any assumptions on the length of the clauses: There may be clauses that have a lot of variables, and others may have just a single variable.

(a) First consider the randomized approximation algorithm we used for MAX 3-SAT, setting each variable independently to *true* or *false* with probability 1/2 each. Show that the expected number of clauses satisfied by this random assignment is at least $k/2$, that is, at least half of the clauses are satisfied in expectation. Give an example to show that there are MAX SAT instances such that no assignment satisfies more than half of the clauses.

(b) If we have a clause that consists only of a single term (e.g., a clause consisting just of x_1, or just of $\overline{x_2}$), then there is only a single way to satisfy it: We need to set the corresponding variable in the appropriate way. If we have two clauses such that one consists of just the term x_i, and the other consists of just the negated term $\overline{x_i}$, then this is a pretty direct contradiction.

Assume that our instance has no such pair of "conflicting clauses"; that is, for no variable x_i do we have both a clause $C = \{x_i\}$ and a clause $C' = \{\overline{x_i}\}$. Modify the randomized procedure above to improve the approximation factor from 1/2 to at least .6. That is, change the algorithm so that the expected number of clauses satisfied by the process is at least $.6k$.

(c) Give a randomized polynomial-time algorithm for the general MAX SAT Problem, so that the expected number of clauses satisfied by the algorithm is at least a .6 fraction of the maximum possible.

(Note that, by the example in part (a), there are instances where one cannot satisfy more than $k/2$ clauses; the point here is that we'd still like an efficient algorithm that, in expectation, can satisfy a .6 fraction *of the maximum that can be satisfied by an optimal assignment*.)

a) CONSIDER A CLAUSE C_i WITH n VARIABLES. THE PROB THAT C_i IS NOT SAT IS $1/2^n$, SO THE PROB THAT C_i IS SAT IS $1 - 1/2^n$. THE WORST CASE IS C_i WITH $n=1$, IN WHICH THE PROB THAT C_i IS SAT IS $1/2$ SINCE THERE ARE K CLAUSES, THE EXPECTED NUMBER OF SATISFIED CLAUSES IS AT LEAST $K/2$.

b) LET THE PROB OF SETTING A SATISFIED VAR IN A SINGLE VAR CLAUSE $p \ge 1/2$. FOR CLAUSES WITH $n > 1$, THE PROB OF SAT AT WORST IS $(1 - 1/2^n) \ge (1 - p^2)$ SO $\ge 3/4$. WE SOLVE $p = 1 - p^2$ TO GET

$$p \approx 0.62 \qquad E[X] \ge 0.62K$$

c) K = TOTAL NUMBER OF CLAUSES. WE REMOVE ONE CONFLICTING SINGLE-VAR CLAUSE. THE MAX C WE COULD SAT IS $K - m$ (m REMOVED). WE CAN SAT $0.62 \cdot (K - 2m) + m$ CLUSES.