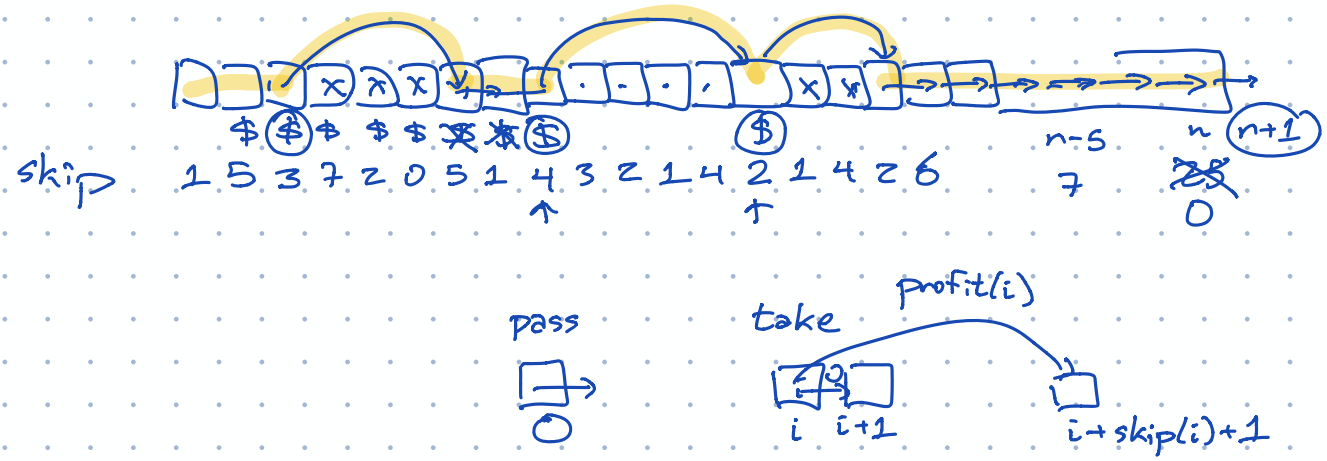
1. ***[Spring 2016]*** After the Revolutionary War, Alexander Hamilton's biggest rival as a lawyer was Aaron Burr. (Sir!) In fact, the two worked next door to each other. Unlike Hamilton, Burr cannot work non-stop; every case he tries exhausts him. The bigger the case, the longer he must rest before he is well enough to take the next case. (Of course, he is willing to wait for it.) If a case arrives while Burr is resting, Hamilton snatches it up instead.

   Burr has been asked to consider a sequence of n upcoming cases. He quickly computes two arrays $profit[1..n]$ and $skip[1..n]$, where for each index i,

   - $profit[i]$ is the amount of money Burr would make by taking the ith case, and
   - $skip[i]$ is the number of consecutive cases Burr must skip if he accepts the ith case. That is, if Burr accepts the ith case, he cannot accept cases $i+1$ through $i+skip[i]$.

   $skip[i] \le n-i$

   Design and analyze an algorithm that determines the maximum total profit Burr can secure from these n cases, using his two arrays as input.



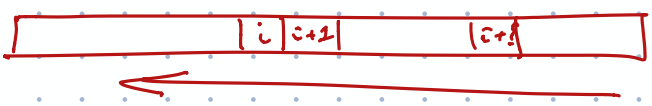
Graph: Verts = cases (+n+1)
Edges = $i \to i+1$ (skip) weight 0
DAG!
topologically sorted! $i \to \min\{n+1, i+skip[i]+1\}$ (take) weight profit[i]

Problem: Longest path in G From vertex 1 to n+1

DP [we did this in class/book] : $O(V+E) = O(n)$ time

MaxProfit(i) = max profit Burr can earn with cases i..n

$$\text{MaxProfit}(i) = \begin{cases} 0 & \text{if } i > n \\ \max\{\text{MaxProfit}(i+1),\ profit[i] + \text{MaxProfit}(i+skip[i]+1)\} & \text{o/w} \end{cases}$$



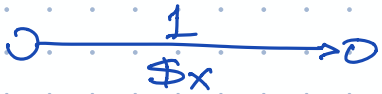
$O(n)$ time

2. ***[Spring 2015]*** Let $G = (V, E)$ be a directed graph, in which every edge has capacity equal to 1 and some arbitrary cost. Edge costs could be positive, negative, or zero. Suppose you have just finished computing the minimum-cost circulation in this graph. Unfortunately, after all that work, now you realize that you recorded the direction of one of the edges incorrectly!

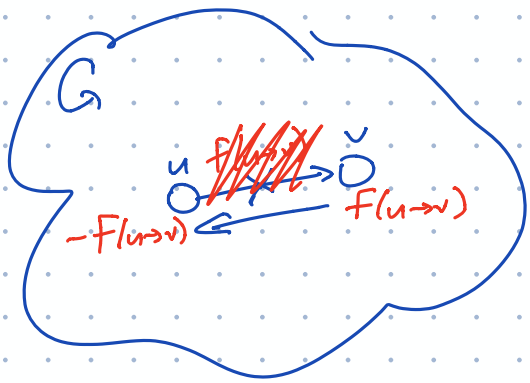
Assume integer valued

Describe and analyze an algorithm to update the minimum-cost circulation in G when the direction of an arbitrary edge in G is reversed. The input to your algorithm consists of the directed graph G, the costs of edges in G, the minimum-cost circulation in G, and the edge to be reversed. Your algorithm should be faster than recomputing the minimum-cost circulation from scratch.

(min-cost flows)



All cap 1 ⇒ all f(e) = 0 or 1

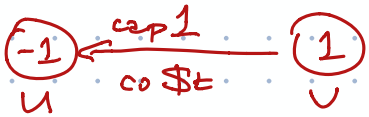


Update min-cost ~~flow~~ circulation

(1) Delete edge u→v

b(u) ← −f(u→v) [supply]
b(v) ← f(u→v) [demand]

(2) Add edge v→u
$(v→u) = $(u→v)
c(v→u) = 1



IF f(u→v) = 1 :

Push 1 unit of flow along shortest path from u to v

Bellman-Ford → O(VE) time

Restored balance

f(u→v) = 0:

Look in G_f for a neg cycle containing v→u

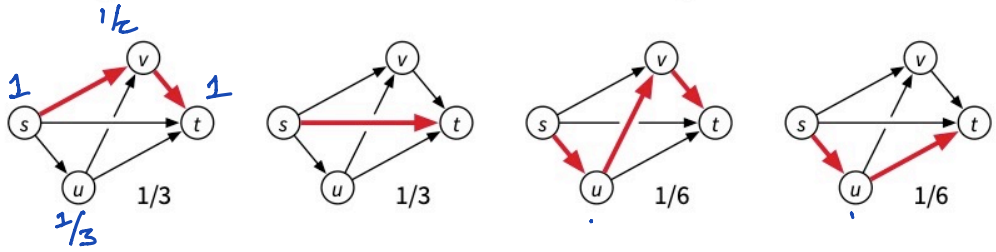
O(VE) time

Compute shortest path from u to v in G_f

if length + $(v→u) < 0
push flow around cycle
else
don't

3. ***[Spring 2017]*** Let $G = (V, E)$ be an arbitrary dag with a unique source s and a unique sink t. Suppose we compute a random walk from s to t, where at each node v (except t), we choose an outgoing edge $v \to w$ uniformly at random to determine the successor of v.

For example, in the following four-node graph, there are four walks from the unique source s to the unique sink t, chosen with the indicated probabilities:

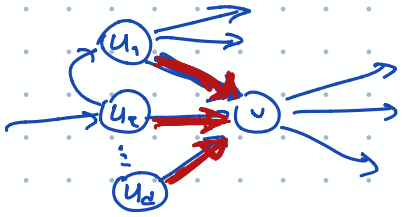


(a) Describe and analyze an algorithm to compute, for every vertex v, the probability that the random walk visits v. For example, in the graph shown above, a random walk visits the source s with probability 1, the bottom vertex u with probability $1/3$, the top vertex v with probability $1/2$, and the sink t with probability 1.

(b) Describe and analyze an algorithm to compute the expected number of edges in the random walk. For example, given the graph shown above, your algorithm should return the number $1 \cdot 1/3 + 2 \cdot (1/2 + 1/6) + 3 \cdot 1/6 = 11/6$.

Assume all relevant arithmetic operations can be performed exactly in $O(1)$ time.

(a) $\Pr[\text{visiting } v]$



Dynamic programming

Recurrence!

$$\Pr[\text{visiting } v] = \sum_{u \to v} \Pr[\text{traversing } u \to v]$$

$$= \sum_{u \to v} \Pr[\text{visiting } u] \cdot \Pr[u \to v \mid \text{visiting } u]$$

$$= \sum_{u \to v} \Pr[\text{visiting } u] \cdot \frac{1}{\text{outdeg}(u)}$$

Base case: $\Pr[\text{visiting } s] = 1$ and $\Pr[\text{visiting } t] = 1$

Memoize into $v.\text{prob}$

v depends on predecessors $\longrightarrow$ evaluate in top. order

$O(V+E)$ time

(b) $E[\text{length}] = E[\#\text{edges}] = \sum_e \Pr[\text{traversing } e]$

Previous algo computes this prob for all edges in $O(V+E)$ time

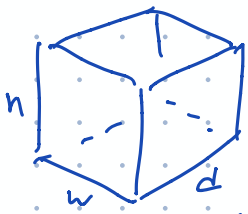
Sum in $O(E)$ time

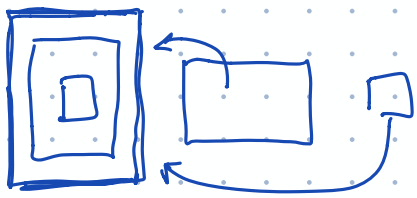
$O(V+E)$ time

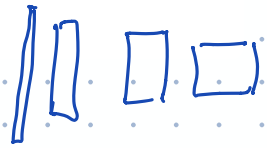
$= \sum_e E[\underbrace{[\text{traverse } e]}_{0 \text{ or } 1}]$

4. ***[Spring 2016]*** Suppose we are given a set of n rectangular boxes, each specified by their height, width, and depth in centimeters. All three dimensions of each box lie strictly between 10cm and 20cm, and all $3n$ dimensions are distinct. As you might expect, one box can be nested inside another if the first box can be rotated so that is is smaller in every dimension than the second box. Boxes can be nested recursively, but two boxes cannot be nested side-by-side inside a third box. A box is *visible* if it is not nested inside another box.

Describe and analyze an algorithm to nest the boxes, so that the number of visible boxes is as small as possible.







DAG!!

Define graph $G=(V,E)$

- V = boxes $\quad |V|=n$
- $E = u \to v$ if box u fits inside box v. $\quad |E|=O(n^2)$

We can build G in $O(n^2)$ time by brute force ($O(1)$ per pair of boxes — try all 6 perms) ☆

$\max\{u.ht, u.wd, u.dp\} < \max\{v.ht, v.wd, v.dp\}$
and med $<$
min $<$

What are we doing with G?

Actual one set nested boxes ⟶ path in G

last box on path is visible
= biggest
= outermost

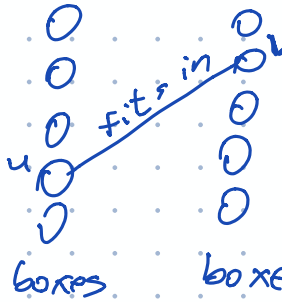
#visible boxes = #paths

Disjoint path cover!

$O(VE)$ time (see book) ⟶ $O(n^3)$

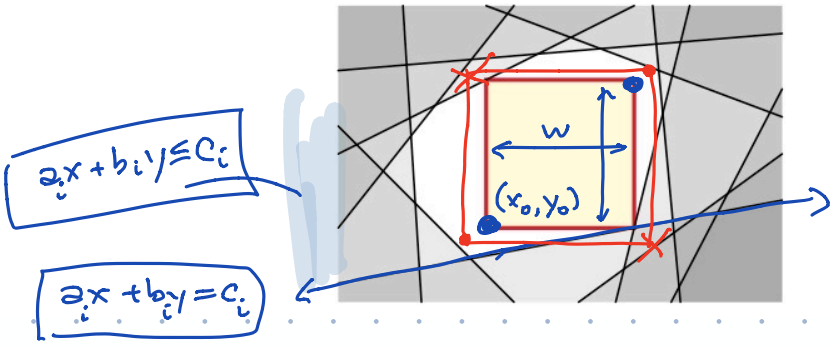
Assign each box to next larger box

Bipartite matching



$O(VE) = O(n^3)$

5. ***[Spring 2015]*** Suppose we are given a sequence of n linear inequalities of the form $a_i x + b_i y \leq c_i$; the set of all points (x, y) that satisfy these inequalities is a convex polygon P in the (x, y)-plane. Describe a linear program whose solution describes the largest square with horizontal and vertical sides that lies inside P. (You can assume that P is non-empty.)



BR
TR
BR
TL

$$\max\ 1w + 0x_0 + 0y_0$$

$$\text{s.t. } a_i x_0 + b_i y_0 \leq c_i \quad \text{for all } i$$

$$a_i(x_0 + w) + b(y_0 + w) \leq c_i \quad \text{for all } i$$

$$a_i(x_0 + w) + b\, y_0 \leq c_i \quad \text{for all } i$$

$$a_i x_0 + b_i(y_0 + w) \leq c_i \quad \text{for all } i$$

Variables:
- x_0 = left side
- y_0 = bottom side
- w = width

Objective:
- maximize w

Constraints: $(4n)$

"square(x_0, y_0, w) inside P"

6. ***[Spring 2017]*** You are applying to participate in this year's Trial of the Pyx, the annual ceremony where samples of all British coinage are tested, to ensure that they conform as strictly as possible to legal standards. As a test of your qualifications, your interviewer at the Worshipful Company of Goldsmiths has given you a bag of n commemorative Alan Turing half-guinea coins, exactly two of which are counterfeit. One counterfeit coin is very slightly lighter than a genuine Turing; the other is very slightly heavier. Together, the two counterfeit coins have *exactly* the same weight as two genuine coins. Your task is to identify the two counterfeit coins.

   The weight difference between the real and fake coins is too small to be detected by anything other than the Royal Pyx Coin Balance. You can place any two disjoint sets of coins in each of the Balance's two pans; the Balance will then indicate which of the two subsets has larger total weight, or that the two subsets have the same total weight. Unfortunately, each use of the Balance requires completing a complicated authorization form (in triplicate), submitting a blood sample, and scheduling the Royal Bugle Corps, so you *really* want to use the Balance as few times as possible.

   (a) Suppose you *randomly* choose $n/2$ of your n coins to put on one pan of the Balance, and put the remaining $n/2$ coins on the other pan. What is the probability that the two subsets have equal weight?

   (b) Describe and analyze a randomized algorithm to identify the two fake coins. What is the expected number of times your algorithm uses the Balance? To simplify the algorithm, you may assume that n is a power of 2.

$n = 2^k$ coins
one light
one heavy

$\binom{n}{k} = \binom{n}{n-k}$

(a) $\Pr[\text{equal wt}] = \dfrac{\#\text{good splits}}{\#\text{splits}} = \dfrac{2\binom{n-2}{n/2}}{\binom{n}{n/2}}$

$\#\text{left good splits} = \binom{n-2}{n/2-2}$

$\#\text{right good splits} = \binom{n-2}{n/2}$

$$\frac{2\binom{n-2}{n/2}}{\binom{n}{n/2}} = \frac{2(n-2)!}{\left(\frac{n}{2}\right)!\left(\frac{n}{2}-2\right)!} \cdot \frac{\left(\frac{n}{2}\right)!\left(\frac{n}{2}\right)!}{n!} = \frac{2 \cdot \left(\frac{n}{2}-1\right)}{n(n-1)}$$

$$= \frac{n-2}{2n-2} = \frac{1}{2} - \frac{1}{2n-2}$$

(b)

Randomly split $\frac{n}{2} \mid \frac{n}{2}$

- L heavy → Binary search for heavy coin in L
- balanced → start over!
- R heavy → symmetric

$O(\log n)$

$L \to \frac{n}{4} \mid \frac{n}{4}$

- L → recurse L
- R → recurse R

→ heavy coin

Binary search R for light coin similar $O(\log n)$ time

Overall expected # trials

$\leq \underbrace{2}_{\text{\#flips to get heads}} + 2\log_2 \frac{n}{2} = \boxed{O(\log n)}$