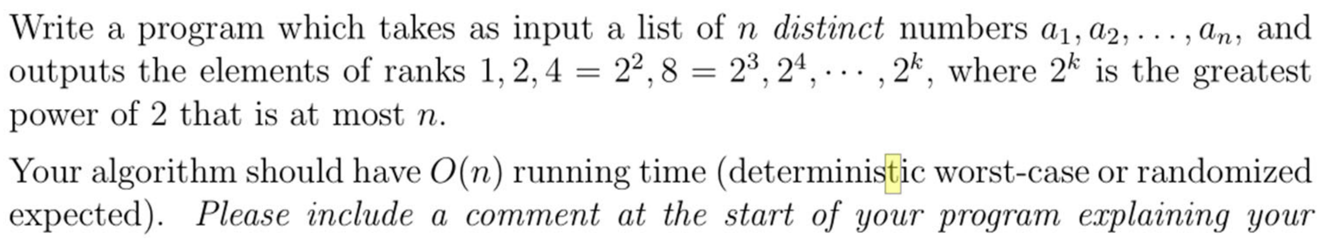
Lecture 1 and 2: Upper & Lower Bounds

- Linear time selection
  - solving recurrences, linearity of expectation
- Tight lower bounds for sorting, max-finding, sorting-by-swaps
  - adversary method, comparison-tree method

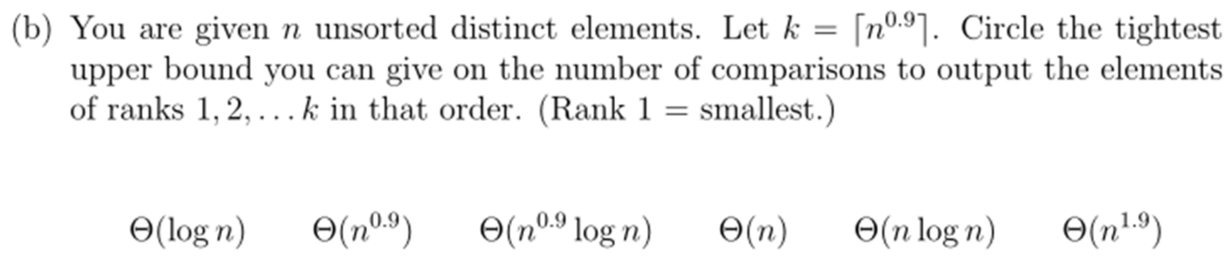


Write a program which takes as input a list of n *distinct* numbers $a_1, a_2, \ldots, a_n$, and outputs the elements of ranks $1, 2, 4 = 2^2, 8 = 2^3, 2^4, \cdots, 2^k$, where 2^k is the greatest power of 2 that is at most n.

Your algorithm should have $O(n)$ running time (deterministic worst-case or randomized expected). *Please include a comment at the start of your program explaining your*

Lecture 1 and 2: Upper & Lower Bounds

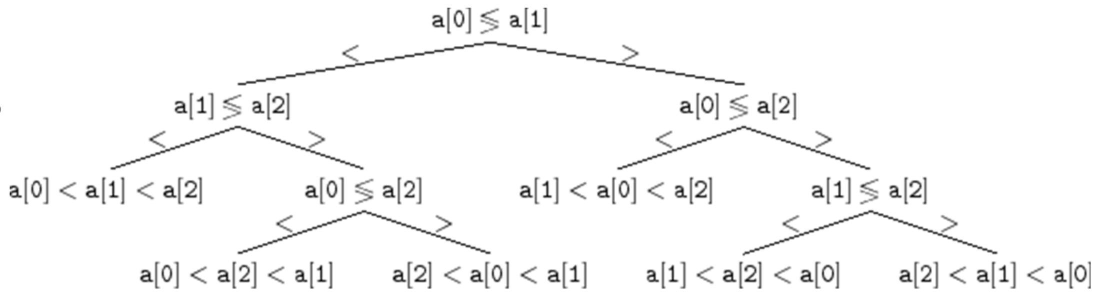
- Linear time selection
  - solving recurrences, linearity of expectation
- Tight lower bounds for sorting, max-finding, sorting-by-swaps
  - adversary method, comparison-tree method



(b) You are given n unsorted distinct elements. Let $k = \lceil n^{0.9} \rceil$. Circle the tightest upper bound you can give on the number of comparisons to output the elements of ranks $1, 2, \ldots k$ in that order. (Rank 1 = smallest.)

$\Theta(\log n)$ $\Theta(n^{0.9})$ $\Theta(n^{0.9} \log n)$ $\Theta(n)$ $\Theta(n \log n)$ $\Theta(n^{1.9})$

Lecture 2: Lower Bounds



- Theorem: Any deterministic comparison-based sorting algorithm must perform $\lg(n!)$ comparisons to sort n elements in the worst case, i.e., for any sorting algorithm A and $n \geq 2$, there is an input I of size n so that A makes $\geq \lg(n!) = \Omega(n \log n)$ comparisons to sort I.

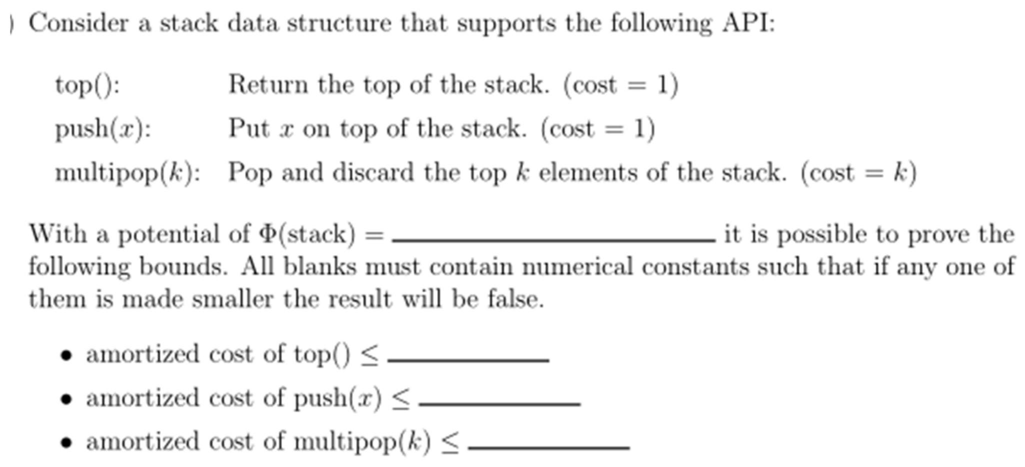
- Proof: Suppose there is a problem with M possible outputs
  - For sorting $M = n!$ since for each output permutation π, there is an input with output π

- Suppose for each possible output, there is an input for which that output is the only correct answer
  - For sorting there are inputs for which π is the only correct answer

- Then there is a lower bound of $\lg M$
  - Consider a set of inputs in one-to-one correspondence with the M possible outputs. Algorithm needs to find which of the M outputs is correct for a given input, and each comparison can be answered in a way that removes at most half of the possible inputs

Lecture 3-4: Amortized Analysis & Union-Find

- Binary counter, maintaining array under inserts and deletes
  - amortized analysis
  - banker’s method and potential function (aka total money in bank)
  - “when is the next expensive operation coming, save enough to pay for it”



) Consider a stack data structure that supports the following API:

top():	Return the top of the stack. (cost = 1)
push(x):	Put x on top of the stack. (cost = 1)
multipop(k):	Pop and discard the top k elements of the stack. (cost = k)

With a potential of $\Phi(\text{stack}) =$ ____________________ it is possible to prove the following bounds. All blanks must contain numerical constants such that if any one of them is made smaller the result will be false.

- amortized cost of top() $\leq$ __________
- amortized cost of push(x) $\leq$ __________
- amortized cost of multipop(k) $\leq$ __________

Lecture 5: Hashing

- h: U -> {0, 1, 2, …, M-1} is universal if for all $x \neq y$,

$$\Pr_{h \leftarrow H}[h(x) = h(y)] \leq \frac{1}{M}$$

- Theorem: If H is universal, then for any set $S \subseteq U$ with |S| = N, for any $x \in S$, if we choose h at random from H, the **expected** number of collisions between x and other elements in S is less than N/M.
  - Ax mod 2 is universal

- Definition: A hash function family H is k-universal if for every set of k distinct keys $x_1, \dots, x_k$ and every set of k values $v_1, \dots, v_k \in \{0, 1, \dots, M-1\}$,

$$\Pr[h(x_1) = v_1 \text{ AND } h(x_2) = v_2 \text{ AND } \dots \text{ AND } h(x_k) = v_k] = \frac{1}{M^k}$$

- Perfect Hashing: can use O(N) space to achieve constant worst-case lookup time to store a set of N keys. Based on2-level hashing scheme

Lecture 6: Data Stream Model

- Stream of elements $\mathrm{a_1, \dots, a_t}$

- $\mathrm{S_t}$ is the multiset of items at time t, so $\mathrm{S_0 = \emptyset, S_1 = \{a_1\}, \dots, S_i = \{a_1, \dots, a_i\}}$,
$$\mathrm{count_t(e) = \{i \in \{1, 2, \dots, t\} \text{ such that } a_i = e\}}$$

- $\mathrm{e} \in \Sigma$ is an ϵ-heavy hitter at time t if $\mathrm{count_t(e)} > \epsilon \cdot \mathrm{t}$

- Maintain $1/\epsilon$ identities and counters. Every time we discard an update, we decrement $1/\epsilon$ counts, so an ϵ-heavy hitter will survive

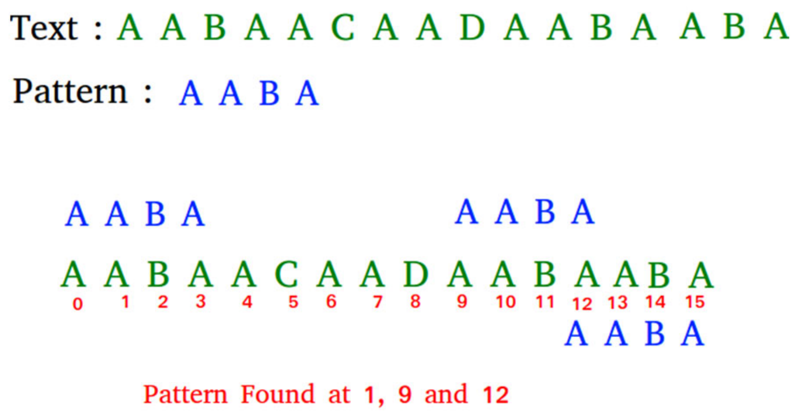
Lecture 6: Data Stream Model

- Heavy hitters with stream deletions
- Query "What is $\mathrm{count}_t(\mathrm{e})$?", should output $\mathrm{est}_t(\mathrm{e})$ with:
$$\Pr[|\mathrm{est}_t(\mathrm{e}) - \mathrm{count}_t(\mathrm{e})| \leq \epsilon |S_t|] \geq 1 - \delta$$
- Want space close to our previous O(1/ ϵ) (log(Σ) + log t) bits
- Let $\mathrm{h}: \Sigma \to \{0,1,2, \ldots, 10/\epsilon\}$ be a 2-universal hash function
- Maintain an array A[0, 1, …, k-1] to store non-negative integers

```
when update $a_t$ arrives:
    **if** $a_t = (\mathrm{add}, \mathrm{e})$ **then** $A[h(e)]++$
    **else** $a_t = (\mathrm{del}, \mathrm{e})$, and $A[h(e)]--$
```

- $\mathrm{est}_t(\mathrm{e}) = A[h(e)]$

Lecture 7: Fingerprinting

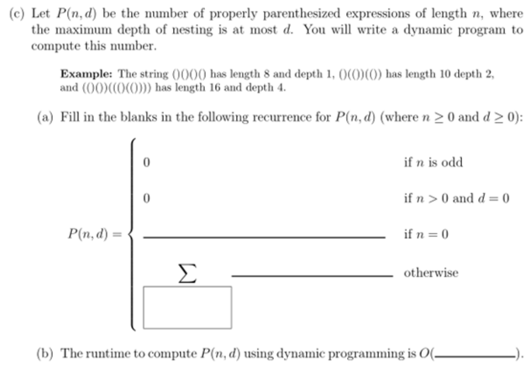


Text : A A B A A C A A D A A B A A B A
Pattern : A A B A
A A B A
A A B A
A A B A A C A A D A A B A A B A
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A A B A
Pattern Found at 1, 9 and 12

- In the string-matching problem, we have
  - A text T of length m
  - A pattern P of length n
- Goal: output all occurrences of the pattern P inside the text T
- Karp-Rabin: $h_p(x) = x \bmod p$ for $x \in \{0,1\}^n$, think of x as an integer
- Create x' by dropping the most significant bit of x, and appending a bit to the right
  - E.g., if x = 0011001, then x' could be 0110010 or 0110011
- Given $h_p(x) = z$, can we compute $h_p(x')$ quickly?
- Suppose x'_{lb} is the lowest-order bit of x', and x_{hb} is the highest order bit of x
- $x' = 2(x - x_{hb} \cdot 2^{n-1}) + x_{lb}'$
- Since $h_p(a+b) = \left(h_p(a) + h_p(b)\right) \bmod p$, and $h_p(2a) = 2h_p(a) \bmod p$,
$$h_p(x') = \left(2h_p(x) - x_{hb} \cdot h_p(2^n) + x'_{lb}\right) \bmod p$$
- *Given $h_p(x)$ and $h_p(2^n)$, this is just O(1) arithmetic operations mod p. O(m+n) total time.*

Lecture 8: Dynamic Programming

- Store solutions to subproblems, avoid resolving repeatedly
- Top-down (via memorization), Bottom-up (via filling table)
    - standard table DP: longest common subsequence, knapsack
    - tree DP (for independent set), subset DP (for TSP)



(c) Let $P(n, d)$ be the number of properly parenthesized expressions of length n, where the maximum depth of nesting is at most d. You will write a dynamic program to compute this number.

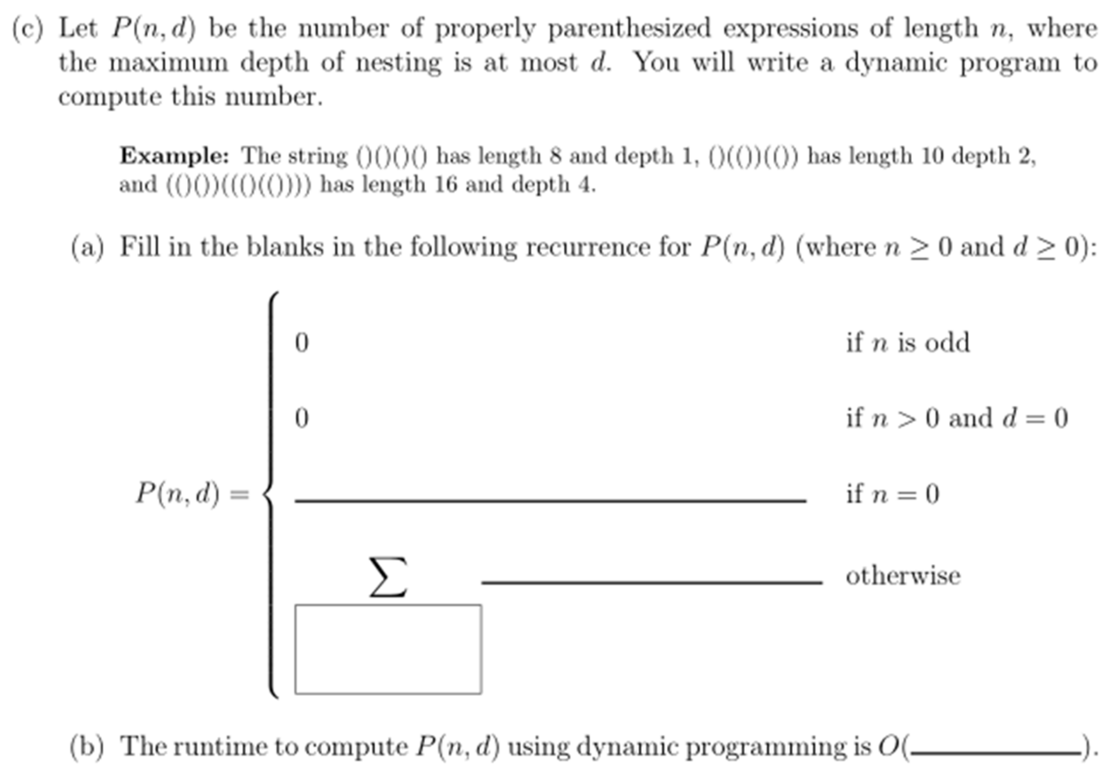
Example: The string ()()()() has length 8 and depth 1, ()(())(()) has length 10 depth 2, and (()())((()(()))) has length 16 and depth 4.

(a) Fill in the blanks in the following recurrence for $P(n, d)$ (where $n \geq 0$ and $d \geq 0$):

$$P(n,d) = \begin{cases} 0 & \text{if } n \text{ is odd} \\ 0 & \text{if } n > 0 \text{ and } d = 0 \\ \rule{6cm}{0.4pt} & \text{if } n = 0 \\ \displaystyle\sum_{\boxed{}} \rule{4cm}{0.4pt} & \text{otherwise} \end{cases}$$

(b) The runtime to compute $P(n, d)$ using dynamic programming is $O(\rule{2cm}{0.4pt})$.

Lecture 8: Dynamic Programming



(c) Let $P(n, d)$ be the number of properly parenthesized expressions of length n, where the maximum depth of nesting is at most d. You will write a dynamic program to compute this number.

Example: The string ()()()() has length 8 and depth 1, ()(())(()) has length 10 depth 2, and (()())((()(()))) has length 16 and depth 4.

(a) Fill in the blanks in the following recurrence for $P(n, d)$ (where $n \geq 0$ and $d \geq 0$):

$$P(n,d) = \begin{cases} 0 & \text{if } n \text{ is odd} \\ 0 & \text{if } n > 0 \text{ and } d = 0 \\ \underline{\hspace{8cm}} & \text{if } n = 0 \\ \displaystyle\sum_{\boxed{\hspace{2.5cm}}} \underline{\hspace{5cm}} & \text{otherwise} \end{cases}$$

(b) The runtime to compute $P(n, d)$ using dynamic programming is $O(\underline{\hspace{3cm}})$.

Lecture 9: Shortest Paths

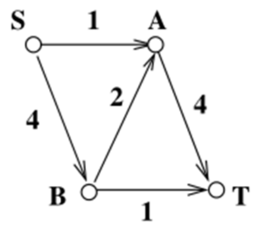
- SSSP : Dijkstra (for non-negative weights)
  : Bellman-Ford (for negative weights, detects negative cycle)

- APSP : Floyd-Warshall
  : Johnson’s algorithm (BF, make edge-lengths non-neg, n Dijkstras)
  : Matrix Multiplication

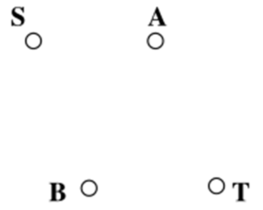
Lecture 10-11: Network Flows

- s-t Flow Network: capacities on edges
- Ford-Fulkerson. Residual graph.

(a) What is the value of the maximum S-T flow in the graph below? ____________



(b) Draw the residual graph that results after running the Ford-Fulkerson algorithm to completion on the above graph. Label the edges with their residual capacities. (Vertices shown below for your convenience).



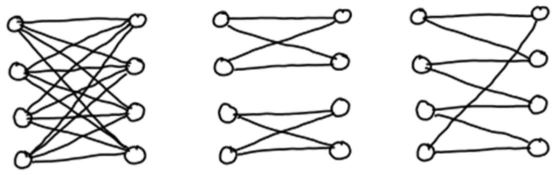
Lecture 10-11: Network Flows

- s-t Flow Network: capacities on edges
- Ford-Fulkerson. Residual graph.

- Max-flow Min-Cut theorem. Integer Max-flow if capacities are integers.
- Solves Bipartite matching.

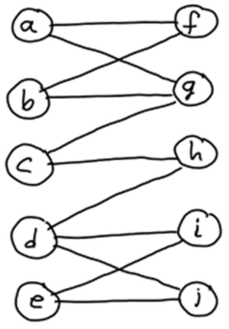
- Poly-time rules: Edmonds-Karp fattest path, E-K shortest augmenting path.
- Min-cost max-flow

Lecture 10-11: Network Flows

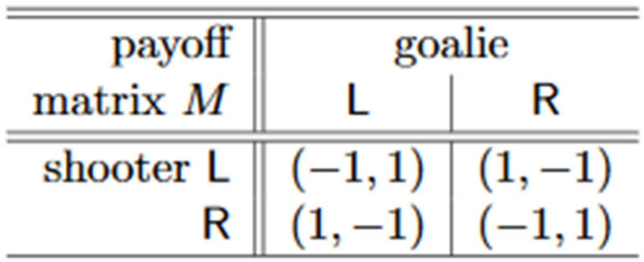


In lecture we saw how to compute a maximum matching in a bipartite graph using a maximum flow algorithm. Let's adapt this idea to find a 2-factor in a bipartite graph (or determine that no 2-factor exists).

(a) Think of a construction to convert any bipartite graph G into a flow network G' such that *G has a 2-factor if and only if G' has a maximum flow that is "large enough"*. Illustrate your construction on the graph below by adding two additional vertices (s and t), and adding some edges. For each edge give it a direction. Label each edge with a capacity.



Lecture 12: Game Theory



payoff matrix M	goalie L	R
shooter L	$(-1, 1)$	$(1, -1)$
R	$(1, -1)$	$(-1, 1)$

- Assume players have independent randomness
- $V_R(p, q) = \sum_{i,j} \Pr[\text{row player plays i, column player plays j}] \cdot R_{i,j} = \sum_{i,j} p_i q_j R_{i,j}$
- $V_C(p, q) = \sum_{i,j} \Pr[\text{row player plays i, column player plays j}] \cdot C_{i,j} = \sum_{i,j} p_i q_j C_{i,j}$
- What is $V_R(p, q)$ + $V_C(p, q)$?
  - 0, since zero-sum game

If p = (.5, .5) and q = (.5, .5) what is V_R?

$$V_R = .25 \cdot (-1) + .25 \cdot 1 + .25 \cdot 1 + .25 \cdot (-1)$$

If p = (.75, .25) and q = (.6, .4) what is V_R? $V_R = -0.1$

Von Neumann: Given a finite 2-player zero-sum game,

$$\text{lb} = \max_p \min_q V_R(p, q) = \min_q \max_p V_R(p, q) = \text{ub}$$

Lecture 13: Linear Programming

Variables: S, P, E

Objective: Maximize 2P + E subject to

Constraints: $S + P + E = 168$

$E \geq 56$

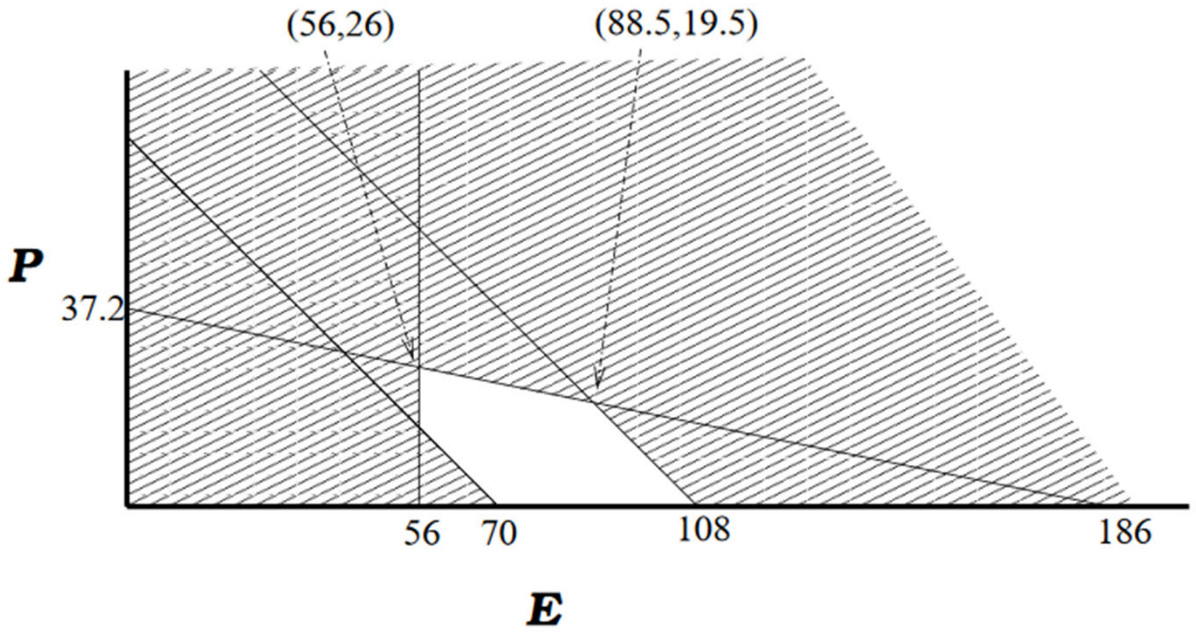
$S \geq 60$

$2S + E - 3P \geq 150$

$P + E \geq 70$

$P \geq 0$

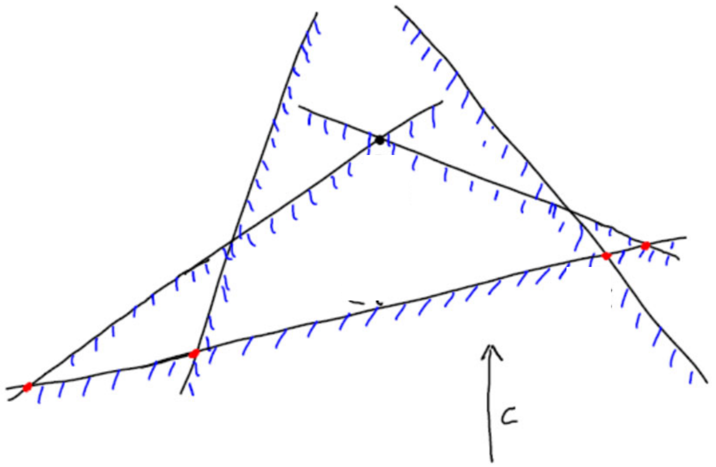
Lecture 13: Linear Programming



Start at vertex of the feasible region (polyhedron in high dimensions)
Look at cost of objective function at each neighbor
Move to neighbor of minimum cost
Always make progress, but could take exponential time (in high dimensions)

Lecture 14: Seidel's Algorithm

- Order constraints randomly: $a_{i_1} \cdot x \leq b_{i_1}, \dots, a_{i_{m-2}} \cdot x \leq b_{i_{m-2}}, c_1, c_2$
  - Leave c_1, c_2 at the end
- Recursively find the optimum x^* of $a_{i_2} \cdot x \leq b_{i_2}, \dots, a_{i_{m-2}} \cdot x \leq b_{i_{m-2}}, c_1, c_2$
- Case 1: If $a_{i_1} \cdot x^* \leq b_{i_1}$, then x^* is overall optimum
  - O(1) time
- Case 2: If $a_{i_1} \cdot x^* > b_{i_1}$, then we need to intersect the line $a_{i_1} \cdot x = b_{i_1}$ with each other line $a_{i_j} \cdot x = b_{i_j}$ and solve a 1-dimensional problem in O(m) time



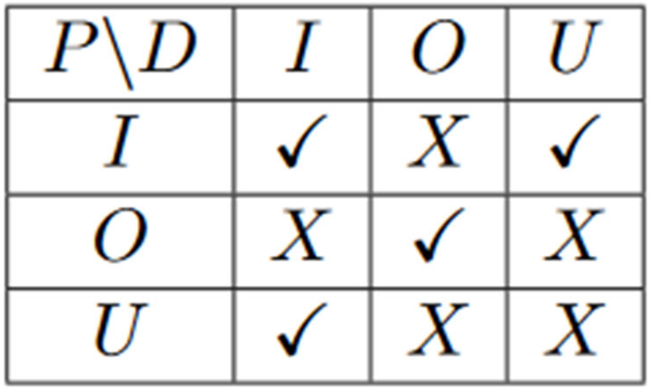
Lecture 15: Duality

$$P = \max(2x_1 + 3x_2)$$
$$\text{s.t.} \quad 4x_1 + 8x_2 \le 12$$
$$2x_1 + x_2 \le 3$$
$$3x_1 + 2x_2 \le 4$$
$$x_1, x_2 \ge 0$$

$$4y_1 + 2y_2 + 3y_3 \ge 2$$
$$8y_1 + y_2 + 2y_3 \ge 3$$
$$y_1, y_2, y_3 \ge 0$$
$$\text{and we seek } \min(12y_1 + 3y_2 + 4y_3)$$

- Dual of the dual is the primal!
- Can we get better upper/lower bounds by looking at more complicated combinations of the inequalities, not just linear combinations?

Lecture 15: Duality



$P \backslash D$	I	O	U
I	✓	X	✓
O	X	✓	X
U	✓	X	X

I means infeasible
O means feasible and bounded
U means unbounded

Check means possible
X means impossible

Lecture 16: NP-completeness

- P: (decision) problems solvable in poly-time
- NP: (dec.) probs with short “proofs” that can be verified in poly-time
- NP-complete: prob in NP, and every problem in NP reduces to it
  - circuit-SAT, 3SAT, CLIQUE, IND-SET, Vertex Cover, Set Cover, 3-Coloring, Hamilton Path, Partition

- to show hardness:

  reduce the known NP-complete problem to unknown problem

  “so the new problem is at least as hard as some NP-complete problem”

The students of 15-451/651 like forming clubs. Each club consists of exactly 3 distinct members. Students may belong to multiple clubs (or may belong to none at all.) We want to pick a subset of students to designate as "officers", such that each club contains at least one officer. A set S of students is *good* if every club intersects S.

> E.g., if there is a student who belongs to every club, we can just designate her as officer, and all clubs will contain an officer. If all clubs are disjoint, we need to pick one officer per club.

The decision problem OFFICERS is: given the roster R of the class (with n students), a list of clubs $C_1, \ldots, C_m$ (each a subset of R of size 3), and an integer $K > 0$, does there exist a *good* set of size K?

Show this problem is **NP** complete. (Hint: One way is to use VERTEX COVER.)

OFFICERS is in **NP** because a verifier can efficiently check a proposed solution as follows:

____

To prove that OFFICERS is **NP**-hard, we can reduce from

$Q =$ ____________________ to $Q' =$ ____________________.

Specifically, given an instance I of Q we create an instance $f(I)$ of Q' as follows:

____

____

Now we show that

$$I \text{ is a YES-instance of } Q \iff f(I) \text{ is a YES-instance of } Q'$$

as follows:

____

____

Finally, time taken to write down $f(I)$ is __________.

Lec 17: Approx Algos

- Derive some “surrogate” for OPT value.
- Show we pay not too much more.
- E.g., p_max, $\frac{\sum_j p_j}{m}$ are lower bounds for OPT makespan
- Can write integer LP, relax to get LP, “round” LP to get integer sol.

Lec 18-19: Online Algos and Experts

- Algorithms that work without knowing future
- Ski-rental, List Update (MoveToFront)
  - potential function capturing difference between us and other algo B.
- Multiplicative Weights:
  - #mistakes <= (1+eps)(#mistakes of best expert) + O(\log N/eps).

Lec 20: Gradient Descent

- Minimizing convex function:
  - Move small step in direction of negative gradient

$$x_{t+1} \leftarrow \mathrm{x_t} - \quad \eta_{\mathrm{t}} \ \nabla f(x_t)$$

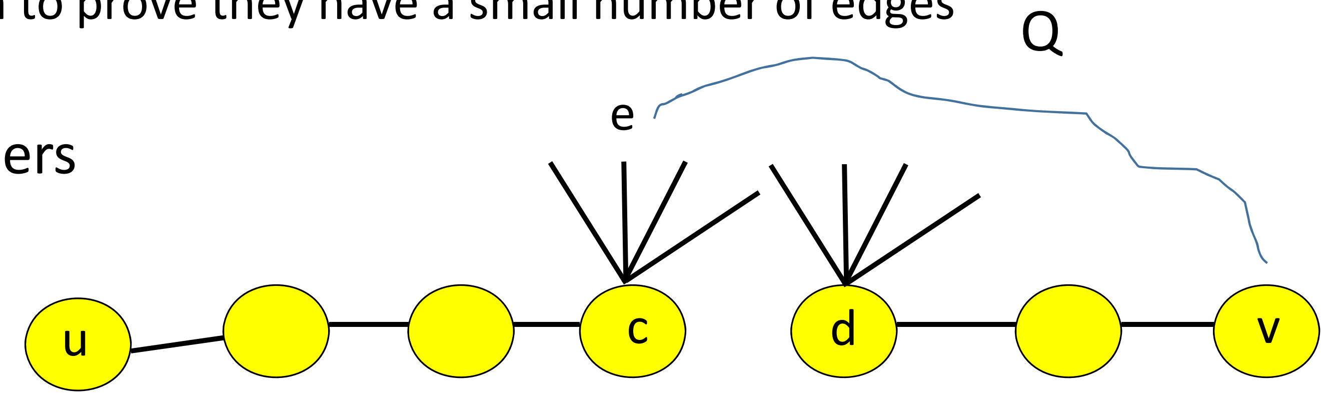
  - If done carefully, get within eps of optimum in time O(1/eps^2).

- In HW, approximately solve Ax=b quickly without Gaussian elim.

  (when A is “well-conditioned”).

Lecture 21: Graph Compression

- Multiplicative spanners
  - Greedy algorithm to find them
  - Use high girth to prove they have a small number of edges
- Additive spanners



- Q is a shortest path from e to v in G!
- $d_Q(e,v) \leq 1 + d_P(c,v)$
- $d_{P'}(u,v) = d_P(u,c) + 1 + d_Q(e,v) \leq d_P(u,c) + d_P(c,v) + 2 \; = d_P(u,v) + 2$

Lecture 22: Nearest Neighbor

Given a query point q and a database $\mathrm{p}_1, \ldots, \mathrm{p}_\mathrm{n}$ of n points in $\{0,1\}^d$, with some preprocessing one can build a data structure so that finding a 2-approximate near neighbor to q can be done with an expected query time of $\mathrm{n}^{.5}\mathrm{d}$!

Lecture 23-24: Regression and Linear Algebra

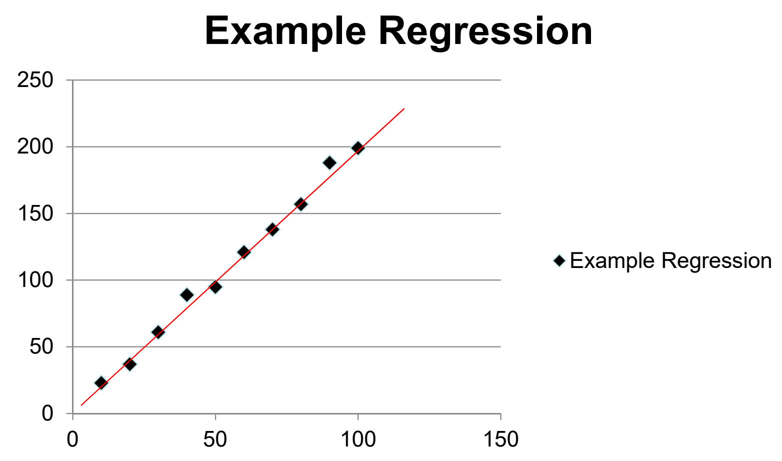
Linear Regression

- Understand linear dependencies between variables in the presence of noise.

Example

- Ohm's law V = R · I

- Find linear function that best fits the data



Use sketch and solve framework!

Lec 25: VCG

- Want to allocate items to maximize social welfare = sum of **values**.
- But players want to maximize their own utility = value – price.
- Use prices to incentivize players to bid truthfully
- Vickery (second-price auction), VCG.
    - Charge players = externality: how much they caused others value to decrease.

Lec 26: Polynomials

- Non-zero degree-d polynomial has at most d roots.
- Given d+1 points (a_i, b_i) there is unique poly P(x) passing thru them.
- Can use to error correct!
  - View message of length n+1 as poly of degree n.
  - Evaluate at n+2k+1 locations (2k more than necessary)
  - Interpolate correctly if at most k errors.