This document contains notes from chapters and corresponding topics covered in *Algorithm Design* in CS 4820 at Cornell.

Chapter 1: Introduction: Some Representative Problems

1.1 Stable matching problem

Given a set of preferences among employers and applicants, can we assign applicants to employers s.t. for every employer E, and every applicant A who is not scheduled to work for E, at least one of the following is the case

- E prefers every one of its accepted applicants to A
- A prefers her current situation over working for employer E

When this holds, the outcome is stable and individual self interest will prevent any applicant/employer deal from being made behind the scenes. There are some complications that arise with this general question, so we define it more clearly as follows. Set $M = \{m_1, \ldots, m_n\}$ of n men, set $W = \{w_1, \ldots, w_n\}$ of n women. $M \times W$ denotes the set of all possible ordered pairs of the form (m, w), $m \in M$, $w \in W$.

Matching S is a set of ordered pairs, each from $M \times W$, with the property that each member of M and each member of W appears in at most one pair in S.

Perfect matching S' is a matching with the property that each member of M and each member of W appears in *exactly one* pair in S'.

Above a perfect matching is a way to pair men with women, s.t. everyone ends up with a partner and nobody is married to more than one person. Now to add the notion of *preferences* to the setting. Each man $m \in M$ ranks all women. Say that m *prefers* w to w' if m ranks w higher than w'. The ordered ranking of m is his *preference list*. Ties are not allowed in the ranking. Define analogously for each women.

Instability When a pair $(m, w') \notin S$ but each m and w' prefers each other over their partner in S.

Stable matching Say a matching S is stable if it is perfect and has no instabilities.

Gale-Shapley Algorithm

while While there is a free man **do**
    pick a man
    man proposes to highest ranked woman he hasn't already proposed to
    if woman is free **then**
        she tentatively accepts proposal
    else
        she compares her current tentatively accepted proposal to the man who is proposing and tentatively accepts the man she prefers in her ranking
    end if
end while
return all tentatively accepted proposals once there are no more unmatched men

Fact The G-S algorithm terminates after at most n^2 iterations of the while loop.

Fact Consider an execution of the G-S algorithm that returns a set S of pairs. That set S is a stable matching.

Fact Let S^* denote the set of pairs $\{(m, best(m)) : m \in M\}$. Every execution of the G-S algorithm results in the set S^*, i.e. as state above, the G-S algorithm favors males.

1.2 Five Representative Problems

Graph Think of graph G as a way to encode pairwise relationships among a set of objects. G consists of a pair of sets, (V, E), V is a collection of nodes, E is a collection of edges, each edge joins two of the nodes. Edge $e \in E$ is a two-element subset of $V : e = \{u, v\}$ for some $u, v \in V$.

Five problems:

- interval scheduling - greedy algorithms
- weighted interval scheduling - dynamic programming
- bipartite matching - network flow problems
- independent set - NP-completeness
- competitive facility location - PSPACE-complete

Chapter 4: Greedy Algorithms

4.1 Interval scheduling, greedy stays ahead

General idea is to use a simple rule to select first possible request, delete incompatible requests, repeat until you run out of requests.

The correct greedy rule is to **select the request with the smallest finishing time**.

Proofs with *greedy stays ahead*.

Algorithm can be made to run in $O(n \log n)$ time. $O(n \log n)$ to sort requests, $O(n)$ to iterate through them.

4.2 Scheduling to minimize lateness, exchange argument

Earliest deadline first sort jobs in increasing order of their deadlines and schedule them in this order.

Use exchange argument to prove correctness.

4.5 Minimum spanning tree problem

Say there is a set of locations $V = \{v_1, \ldots, v_n\}$, you want to connect them all (path between every node) as cheaply as possible (min number of paths). For certain pairs (v_i, v_j) you can build a path connecting them at cost $c(v_i, v_j) > 0$. We represent the set of possible links as a graph $G = (V, E)$, with a positive cost c_e corresponding to each edge $e = (v_i, v_j) \in E$. Problem is to find a subset of edges $T \subseteq E$ so that the graph (V, T) is connected and total cost $\sum_{e \in T} c_e$ is as small as possible.

Fact Let T be a minimum cost solution to problem posed above, (V, T) is a tree.

We call the subset $T \subseteq G$ a **spanning tree** of G if (V, T) is a tree. Thus the problem posed above can be rephrased as a problem of finding a *minimum spanning tree*.

Kruskal's Start with no edges, build a spanning tree by successively inserting edges from E in order of increasing cost. Insert edge e as long as it does not create a cycle.

Fact Kruskal's can be implemented on a graph with n nodes and m edges to run in $O(m \log n)$ time.

Prim's Start with root node s and greedily build spanning tree from that. At each step, add node to spanning tree that minimizes attachment cost ($\min_{e=(u,v):u \in S}$) and include edge (u, v) that achieves this minimum in the minimum spanning tree.

Fact Using priority queue, you can implement Prim's to run on graph with n nodes and m edges to run in $O(m)$ time.

Reverse delete Backward version of Kruskal. Start with full graph $G = (V, E)$ and delete edges in order of decreasing cost. Delete edge as long as the deletion does not disconnect the current graph.

Borukva ?

Chapter 5: Divide and Conquer

Divide and Conquer is a class of algorithmic techniques in which you break the input into smaller parts and solve the problem on each part recursively, then combine the solutions to the subproblems to generate the solution to the overall problem.

Template Divide input into two pieces of equal size, solve subproblems on these pieces separately by recursion, combine results into overall solution, spend only linear time for division and recombining.

Recurrence relation For any divide and conquer that generally fits the *template*, let $T(n)$ denote the worse case running time on input of size n, supposing that n is even, the algo spends $O(n)$ time to divide the input into pieces of size $n/2$ and then spends time $T(n/2)$ to solve each one, and finally spends $O(n)$ time to combine the solutions of each one. This gives rise to the following *recurrence relation* (for some constant c)

$$T(n) \leq 2T(n/2) + cn \tag{1}$$

when $n > 2$ and $T(2) \leq c$. You could also write without c as

$$T(n) \leq 2T(n/2) + O(n)$$

but its better to make c explicit.

To solve a recurrence:

- "unroll" the recursion. i.e. accout for the running time for first few levels, identify pattern for recurrence expanding, sum running times over all levels of recursion and arrive at total running time.
- Start with guess for solution, substitute into recurrence relation, check that it works. Justify the "plugging" in argument via induction on n.

Fact Any function $T(\cdot)$ satisfying (1) is bounded by $O(n \log n)$, when $n > 1$.

5.5 Integer multiplication

Problem is the multiplication of two integers (also assume we are multiplying in base 2, but it doesn't matter overall). The traditional way to do this takes $O(n^2)$.

function RECURSIVE-MULTIPLY(x, y)
$x = x_1 * 2^{n/2} + x_0$
$y = y_1 * 2^{n/2} + y_0$
Compute $x_1 + x_0$ and $y_1 + y_0$
$p =$ RECURSIVE-MULTIPLY($x_1 + x_0$,$y_1 + y_0$)
$x_1y_1 =$ RECURSIVE-MULTIPLY(x_1,y_1)
$x_0y_0 =$ RECURSIVE-MULTIPLY(x_0,y_0)
return $x_1y_1 \cdot 2^n + (p - x_1y_1 - x_0y_0) \cdot 2^{n/2} + x_0y_0$
end function

Fact The running time of Recursive-multiply on two n-bit factors is $O(n^{\log_2 3}) = O(n^{1.59})$

Linear-space sequence allignment

Same sequence alignment question that is outlined in DP section, but we want to do better than $O(mn)$ space. We will develop algo to complete get the matching in $O(mn)$ time (same as before) but in $O(m + n)$ space.

Idea is to only store recent values needed to calculate next step of recurrence. Problem is that if you only do this, you achieve $O(m)$ space but you can't implement a "find-solution" type of algorithm to backtrack and return the alignment once the algorithm completes.

We accomplish the solution with a *backward* formulation of the dynamic program

Backward recurrence For $i < m$ and $j < n$ we have

$$g(i, j) = \min(\alpha_{x_{i+1}, y_{j+1}} + g(i + 1, j + 1), \delta + g(i, j + 1), \delta + g(i + 1, j))$$

You then use divide and conquer and both the forward (described in DP section) and backward recurrences (on each half of substrings) to find the optimal alignment, maintain a global list P that

holds nodes on the "corner to corner" path, P will have at most $m+n$ entries, thus $O(m+n)$ space complexity.

Space-efficient strong matching

todo

Chapter 6: Dynamic Programming

General DP outline

To develop DP algorithm for a problem, need to have subproblems derived from original problem with the following conditions

- only polynomial number of subproblems
- solution to original problem can easily be computed from solution to subproblems, original problem may actually be a subproblem/could be considered a subproblem
- $\exists$ a natural ordering of subproblems with an easy to compute recurrence, that allows you to compute solution to subproblem from the solutions to some number of smaller subproblems

Its never clear that a collection of subproblems will be useful until one finds the recurrence, but its hard to find a recurrence without any subproblems.

Essential components of solution (check this over)

- basecase
- recurrence
- final output, how to arrive there

6.1 Weighted interval scheduling

This is the more general case of interval scheduling problem, each interval now comes with a certain value (or weight), want to accept a set of intervals of maximum value. To refresh, we have n requests labeled $1, \ldots, n$, with each request i specifying start time s_i, finish time f_i, and value v_i. Intervals are compatible if they do not overlap. Goal is to select a subset $S \subseteq \{1, \ldots, n\}$ of mutually compatible intervals, as to maximize the sum of values of selected intervals $\sum_{i \in S} v_i$. Suppose requests are sorted in order of nondecreasing finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$. Define $p(j)$ for an interval j, to be the largest index $i < j$ (i before j) s.t. intervals i and j are disjoint. i is the leftmost interval that ends before j begins. $p(j) = 0$ if no i satisfies.

For any value j say $opt(j)$ is the optimal solution of the problem consisting of requests $\{1, \ldots, j\}$

Recurrence relation is

$$opt(j) = \max\Big(v_j + opt(p(j)), opt(j-1)\Big)$$

This will correctly compute $opt(j)$ for $j = 1, \ldots, n$.

Problem: As is, this runs in exponential time. **Solution**: Memoize the results to make it run faster.

Memoization Store previous *opt*() computations in a data structure (typically an array) to use in future computations of *opt*().

Fact Running time of memoized version (assuming input intervals are sorted by finishing times) is $O(n)$.

Compute solution w/ value you can "work backwards" to find the solution along with the optimal value (find the specific intervals to schedule), can be done in $O(n)$ time.

To get solution by working backwards:

function FIND-SOLUTION(j)
  if $j = 0$ **then**
    return null
  else if $v_j + M[p(j)] \geq M[j-1]$ **then**
    return j together with the result of FIND-SOLUTION($p(j)$)
  else
    return the result of FIND-SOLUTION($j-1$)
  end if
end function

6.3 Segmented least squares

This problem involves multi-way choices in the recurrence (not just binary). The fundamental issue is *change detection*, identifying points in a sequence at which discrete change occurs.

Given a set of points $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ with $x_1 < x_2 < \cdots < x_n$. Use p_i to denote point (x_i, y_i). Must partition P into some number of segments. Each segment is a subset of P that represents a contiguous set of x coordinates, i.e. a subset of the form $\{p_i, p_{i+1}, \ldots, p_{j-1}, p_j\}$ for indices $i \leq j$. Then for each segment S in partition of P, compute the line minimizing error wrt points in S, according to formulas in text. Penalty of partition is a sum of following terms:

- number of segments into which we partition P, times a fixed, given multiplier $C > 0$
- for each segment, the error value of the optimal line through that segment

Let $opt(i)$ denote the optimum solution for the points $p_1, \ldots, p_i$ and let $e_{i,j}$ denote the minimum error of any line with respect to $p_i, p_{i+1}, \ldots, p_j$. We know then that

Fact If the last segment of the optimal partition is $p_i, \ldots, p_n$ then the value of the optimal solution is $opt(n) = e_{i,n} + C + opt(i-1)$

Recurrence For the subproblem on the points $p_1, \ldots, p_j$

$$opt(j) = min_{1 \leq i \leq j}(e_{i,j} + C + opt(i-1))$$

and the segment $p_i, \ldots, p_j$ is used in an optimum solution for the subproblem iff the minimum is obtained using index i.

This algorithm runs in $O(n^2)$ time.

6.6 Sequence alignment (a.k.a. edit distance)

Suppose we are given two strings X and Y, where each is a sequence of symbols $X = x_1, x_2, \ldots, x_m$ and $Y = y_1, y_2, \ldots, y_n$. Consider the sets $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ representing positions in X and Y, a matching is a set of ordered pairs with property that each item occurs in at most one pair. Say the matching M is an alignment if there are no crossing pairs. Want to find the optimal alignment between X ane Y. According to following criteria

- Parameter $\delta > 0$ defines gap penalty. For each position of X and Y that is not matched in M (each gap), incur a cost of δ.
- For each pair of letters p, q, there is a mismatch cost of $\alpha_{p,q}$ for lining up p with q.
- The cost of M is the sum of the gap and mismatch costs, we seek an alignment of min cost.

Recurrence Let $opt(i, j)$ denote the minimum cost of an alignment between $x_1 x_2 \cdots x_i$ and $y_1 y_2 \cdots y_j$. The minimum alignment costs satisfy the following recurrence for $i \geq 1$ and $j \geq 1$:

$$opt(i, j) = \min(\alpha_{x_i, y_j} + opt(i-1, j-1), \delta + opt(i-1, j), \delta + opt(i, j-1))$$

(i, j) is an optimal alignment M for this subproblem iff the minimum is achieved by the first of these values.

Output $opt(m, n)$ is the value we are seeking in the problem.

Running time $O(mn)$, because the array has $O(mn)$ entries and at worst we spend constant time on each.

Knapsack problem

Starting with subset sum problem...

Given n items $\{1, \ldots, n\}$, and each has a given non-negative weigth w_i (for $i = 1, \ldots, n$). Also given bound W. Would like to select subset S of items so that $\sum_{i \in S} w_i \leq W$ and subject to this restriction, $\sum_{i \in S} w_i$ is maximized. This is the *subset sum problem.*

Denote the optimal solution using subset of items $\{1, \ldots, i\}$ with maximum allowed weight w as $opt(i, w)$

$$opt(i, w) = max_s$$

Recurrence If $w < w_i$ then $opt(i, w) = opt(i-1, w)$, else

$$opt(i, w) = \max(opt(i-1, w), w_i + opt(i-1, w - w_i))$$

Fact This algorithm correctly computes the optimal value and runs in $O(nW)$ time.

Fact Given a table M of optimal values of subproblems, the optimal set S can be found in $O(n)$ time.

The subset sum problem above is a special case of the more general *Knapsack problem*, where each request i has both a value v_i and a weight w_i. The goal is to now select a subset S of maximum total value $\sum_{i \in S} v_i$, subject to the restriction that its total weight does not exceed W: $\sum_{i \in S} w_i \leq W$.

Recurrence If $w < w_i$ then $opt(i, w) = opt(i-1, w)$, else

$$opt(i, w) = \max(opt(i-1, w), v_i + opt(i-1, w-w_i))$$

Fact The knapsack problem can be solved in $O(nW)$ time.

Bellman-Ford shortest path algorithm

Let $G = (V, E)$ be a directed graph. Assume that each edge $(i, j) \in E$ has weight c_{ij}. We will try to use dynamic programming to solve the problem of finding a shortest path from s to t when there are negative cost edges, but no negative cycles.

Fact If G has no negative cycles, then $\exists$ a shortest path from s to t that is simple (i.e. does not repeat nodes), and hence has at most $n-1$ edges.

Say $opt(i, v)$ is the minimum cost of a $v - t$ path using at most i edges.

Recurrence If $i > 0$ then

$$opt(i, v) = \min(opt(i-1, v), \min_{w \in V} opt(i-1, w) + c_{vw})$$

Fact This algo correctly computes minimum cost of an $s-t$ path in any graph that has no negative cycles and runs in $O(n^3)$ time.

Fact Algo can be implemented in $O(mn)$ time.

Chapter 7: Network Flow

7.1 Max Flow and Ford Fulkerson

Flow networks a directed graph $G = (V, E)$ with the following features

- each edge $e \in E$ has non-negative capacity c_e
- single source node $s \in V$
- single sink node $t \in V$
- nodes other than s and t are internal nodes

No edge enters the source s and no edge leaves the sink t. Also there is at least one edge incident to each internal node and all capacities are integer valued.

Flow this is an abstract representation of the traffic being carried by the network. Say that an $s - t$ flow is a function f that maps edge e to a non-negative real number, $f : E \to \mathbb{R}^+$, the value $f(e)$ represents the amount of flow being carried by edge e and f must satisfy

- (capacity condition) $\forall\ e \in E$, $0 \leq f(e) \leq c_e$
- (conservation condition) $\forall\ v \in V : v \neq s, t$,

$$\sum_{e \text{ into } V} f(e) = \sum_{e \text{ out of } V} f(e)$$

Value of flow f the amount of flow generated at the source

$$v(f) = \sum_{e \text{ out of } S} f(e)$$

$$f^{\text{out}}(v) = \sum_{e \text{ out of } v} f(e)$$

$$f^{\text{in}}(v) = \sum_{e \text{ into } v} f(e)$$

For a set of vertices $S \subseteq V$

$$f^{\text{out}}(S) = \sum_{e \text{ out of } S} f(e)$$

$$f^{\text{in}}(S) = \sum_{e \text{ into } S} f(e)$$

Max flow problem Given a flow network, find a flow of maximum possible value. Max flow is related to min cut (max flow value is same as minimum capacity along min cut division).

Residual graph Given flow network G and flow f on G, define G_f to be the residual graph of G wrt f as follows

- node set G_f is same as G
- $\forall$ edges $e = (u, v)$ of G on which $f(e) < c_e$, add edge $e = (u, v)$ to G_f with capacity $c - f(e)$ (left over units of flow that can be pushed forward), these edges added are forwards edges
- $\forall$ edges $e = (u, v)$ of G on which $f(e) > 0$, add edge $e' = (v, u)$ to G_f with capacity $f(e)$ (flow we can undo and push backwards), these edges added are called backwards edges

Augmenting paths Say P is a simple $s-t$ path in G_f, P does not visit any node more than once. Define bottleneck(P, f) to be the minimum residual capacity of any edge on P wrt flow f. Say an augmenting path is any $s-t$ path in residual graph. Follow this to make augmenting path

function AUGMENT(f,P)
  let b = bottleneck(P, f)
  for edge $(u, v) \in P$ **do**
    if $e = (u, v)$ is a forward edge **then**
      increase $f(e)$ in G by b
    else
      ((u, v) is a backward edge, say $e = (v, u)$), decrease $f(e)$ in G by b
    end if
  end for
  return f
end function

There will be a new flow as a result of the function.

Ford Fulkerson Algorith to compute $s - t$ flow in G

function MAX-FLOW

```
    Initialize f(e) = 0 ∀ e ∈ G
    while there is an s − t path in residual graph G_f do
        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'
    end while
    return f
end function
```

Fact At every intermediate stage of Ford Fulkerson, the flow values $f(e)$ and the residual capacities in G_f are integers

Fact Suppose that all capacities in the flow network G are integers. Then the Ford Fulkerson algorithm terminates in at most C iterations of the while loop, where $C = \sum_{e \text{ out of } s} c_e$.

Fact Suppose that all capacities in the flow network G are integers and m is number of edges. Then the Ford Fulkerson algorithm can be implemented in $O(mC)$ time.

7.2 Max flows and min cuts in a network

Fact Let f be any $s-t$ flow, and (A, B) any $s-t$ cut. Then $v(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$.

Fact Let f be any $s-t$ flow, and (A, B) any $s-t$ cut. Then $v(f) = f^{\text{in}}(B) - f^{\text{out}}(B)$.

Fact Let f be any $s-t$ flow, and (A, B) any $s-t$ cut. Then $v(f) \leq c(A, B)$, i.e. the value of every flow is upper-bounded by the capacity of every cut.

Fact If f is an $s-t$ flow s.t. there is no $s-t$ path in the residual graph G_f, then there is an $s-t$ cut (A^*, B^*) in G for which $v(f) = c(A^*, B^*)$. Consequently, f has the maximum value of any flow in G, and (A^*, B^*) has the minimum capacity of any $s-t$ cut in G.

Fact The flow $\bar{f}$ returned by Ford Fulkerson is a maximum flow.

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

Fact In every flow network, $\exists$ a flow f and cut (A, B) s.t. $v(f) = c(A, B)$.

Fact In every flow network, the maximum value of an $s-t$ flow is equal to the minimum capacity of an $s-t$ cut.

Fact If all capacities in the flow network are integers, then $\exists$ a maximum flow f for which every flow value $f(e)$ is an integer.

Note Note that if you allow real-valued capacities in the flow network, then it is possible for the Ford Fulkerson algo to never terminate.

7.5 First application: Bipartite matching problem

Recall a *bipartite graph* $G = (V, E)$ is undirected graph whose node set can be partitioned as $V = X \cup Y$ s.t. every edge $e \in E$ has one end in X and the other end in Y. A *matching* M in G is

a subset of the edges $M \subseteq E$ s.t. each node appears in at most one edge in M. Bipartite matching problem wants to find a matching M in G of the largest possible size.

To make a flow network G' from G, make all edges from X to Y directed, then add node s and add edge $(s, x)\ \forall\ x \in X$, then add node t and add edge $(y, t)\ \forall\ y \in Y$, give each edge a capacity of 1. From the textbook:

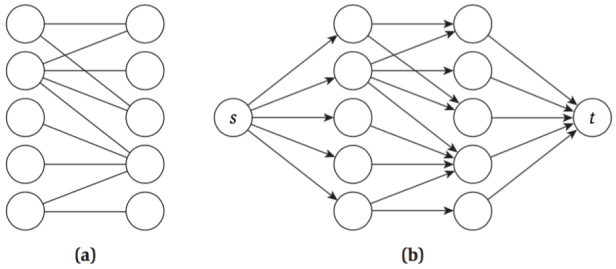

Figure 1: How to create flow network from bipartite graph

Compute maximum flow on this network, value of the max flow is equal to the size of the maximum matching in G, and we can use the max flow to recover the max matching.

Fact The size of max matching in G is equal to the value of the maximum flow in G'; and the edges in such a matching in G are the edges that carry flow from X to Y in G'.

Fact Ford Fulkerson algorithm can be used to find a maximum matching in a bipartite graph in $O(mn)$ time, m = number of edges in bipartite graph, $n = |X| = |Y|$.

Define $\Gamma(A)$ to be the set of all nodes that are adjacent to nodes in A.

Fact If a bipartite graph $G = (V, E)$ with two sides X and Y has a perfect matching, then $\forall$ $A \subseteq X$ we must have $|\Gamma(A)| \geq |A|$.

Fact Assume that the bipartite graph $G = (V, E)$ has two sides X and Y s.t. $|X| = |Y|$. Then graph G either has a perfect matching or $\exists$ a subset $A \subseteq X$ s.t. $|\Gamma(A)| \geq |A|$. A perfect matching or an appropriate subset A can be found in $O(mn)$ time.

7.7 Extensions to max flow problem

Its really useful that we are able to reduce problems with a non-trivial combinatorial search component to a problem involving find a maximum flow (or minimum cut) in a directed graph, as this gives us the ability to solve in polynomial time.

Circulations with demands

We now have a set S of sources and a set T of sinks, each $s \in S$ generates supply $-d_v$ and each $t \in T$ has demand d_v. We want to ship flow from s's to t's to satisfy the supplies and demands of the sources and sinks. Say that a circulation with demands $\{d_v\}$ is a function f that assigns a non-negative real number to each edge and satisfies

- (Capacity conditions) $\forall\ e \in E$, $0 \leq f(e) \leq c_e$
- (Demand conditions) $\forall\ v \in V$, $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$

Fact If $\exists$ a feasible circulation with demands d_v then $\sum_v d_v = 0$

Reduction to max flow add a super source s^* and super sink t^*. For each $u \in S$ add edge (s^*, u) with capacity $-d_u$. For each $v \in T$ add edge (v, t^*) with capacity d_v.

Fact There is a feasible circulation with demands $\{d_v\}$ in G iff the maximum $s^* - t^*$ in G' has value D. If all capacities and demands in G are integers and there is a feasible circulation, then there is a feasible circulation that is integer valued.

$$D = \sum_{v:d_v>0} d_v = \sum_{v:d_v<0} -d_v$$

Circulations with demands and lower bounds

Similar to above but we also want to force the flow to make use of certain edges. We do this with lower bounds (l_e on each edge e in flow network). Flow network $G = (V, E)$, $0 \leq l_e \leq c_e$ for each $e \in E$, and also each $v \in V$ has demand d_v. Circulation in this new network now must satisfy

- (Capacity conditions) $\forall\ e \in E$, $l_e \leq f(e) \leq c_e$
- (Demand conditions) $\forall\ v \in V$, $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$

We want to know if $\exists$ a feasible circulation. We will reduce this problem to the problem above where we have no lower bounds on capacity (which then gets reduced again). To construct our reduction lets start defining initial circulation f_0 via $f_0(e) = l_e\ \forall e$ and by defining L_v via

$$L_v = f_0^{\text{in}}(v) - f_0^{\text{out}}(v) = \sum_{e \text{ into } v} l_e - \sum_{e \text{ out of } v} l_e$$

Now make G' similarly to our previous reduction but capacity of each edge is now $c_e - l_e$ and demand for each node is $d_v - L_v$. This is now equivalent to the instance above with demands but no lower bounds, and we can use the algorithm from that to solve it.

Fact There is a feasible circulation in G iff there is a feasible circulation in G'. If all demands, capacities, and lower bounds in G are integers and $\exists$ a feasible circulation, then $\exists$ a feasible circulation that is integer valued.

7.10 Image Segmentation

Segmenting an image into various coherent regions. We want to label each pixel as belonging to either the foreground or background. Let V be the set of pixels and E be set of all pairs of neighboring pixels. This gives undirected graph $G = (V, E)$. Each pixel i has a likelihood a_i that it is in the foreground and b_i that it is in the backgroud. For each pair of pixels (i, j) there is a separation penalty p_{ij} for placing one in one ground and the other in the other. Segmentation problem is then to find a partition of the set of pixels into A (foreground) and B (background) to maximize

$$q(A, B) = \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{(i,j) \in E, |A \cap (i,j)|=1} p_{ij}$$

which is the same as minimizing

$$q'(A, B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{(i,j) \in E, |A \cap (i,j)|=1} p_{ij}$$

To make a flow network $G' = (V', E')$ we first add a super source and super sink, s and t respectively. The node set consists of V from before combined with the new s and t nodes. For each neighboring pair i and j of nodes, we add two edges (i, j) and (j, i), each with capacity p_{ij}. For each pixel i we add edge (s, i) with capacity a_i and edge (i, t) with capacity b_i. An $s - t$ cut (A, B) corresponds to partition of pixels into sets A and B. We can now solve the min cut problem (as we know how to do) and we have an optimal model of foreground/background segmentation.

Fact The solution to the Segmentation Problem can be obtained by a minimum-cut algorithm on G' constructed above. For a minimum cut (A', B'), the partition (A, B) obtained by deleting s^* and t^* maximizes the segmentation value of $q(A, B)$.

Chapter 8: NP and Computational Intractability

8.1 Polynomial time reductions

Reduction a particular problem X is at least as hard has problem Y when if we have a black box capable of solving X, then we can also use that to solve Y. Can arbitrary instances of Y be solved using a polynomial number of standard computational steps plus a polynomial number of calls to a black box that solves problem X? If yes, then $Y \leq_P X$.

Fact Suppose $Y \leq_P X$. If X can be solved in polynomial time, then Y can be solved in polynomial time.

Fact Suppose $Y \leq_P X$. If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.

Independent Set Given $G = (V, E)$, set of nodes $S \subseteq V$ are independent if no two nodes in S are joined by an edge. Also given k, question is whether G contains an independent set of size at least k.

Vertex Cover Given $G = (V, E)$, say a set of nodes $S \subseteq V$ is a vertex cover if every edge $e \in E$ has at least one end in S. Vertices do the covering, edges get covered. Also given k, question is does G contain a vertex cover of size at most k.

Fact Let $G = (V, E)$ be a graph. Then S is an independent set iff its complement $V - S$ is a vertex cover.

Fact Independent Set (IS) $\leq_P$ Vertex Cover (VC)

Fact Vertex Cover (VC) $\leq_P$ Independent Set (IS)

Set Cover Given a set U of n elements, a collection $S_1, \ldots, S_m$ of subsets of U, and a number k, does there exist a collection of at most k of these sets whose union is equal to all of U.

Fact Vertex Cover (VC) $\leq_P$ Set Cover (SC)

Set Packing Given a set U of n elements, a collection $S_1, \ldots, S_m$ of U, and a number k, does there exist a collection of at least k of these sets with the property that no two of them intersect.

Fact Independent Set (IS) $\leq_P$ Set Packing (SP)

8.2 Reductions via gadgets: Satisfiability problem

SAT and 3-SAT problems

Suppose we are given a set X of n booleans $x_1, \ldots, x_n$; each take either 0 or 1. By a *term* in X, we mean one of the variables x_i or its negation $\bar{x_i}$. A *clause* is a disjunction of distinct terms

$$t_1 \vee t_2 \vee \cdots \vee t_l$$

With each $t_i \in \{x_1, x_2, \ldots, x_n, \bar{x_1}, \ldots, \bar{x_n}\}$

A *truth assignment* for X is an assignment of the value 0 or 1 to each x_i, i.e. a function $v : X \to \{0, 1\}$. Assignment v implicitly gives $\bar{x_i}$ the opposite truth value from x_i. An assignment *satisfies* a clause C if it causes C to evaluate to 1 under boolean logic, i.e. at least one term in C is set to 1. An assignment satisfies a collection of clauses $C_1, \ldots, C_k$ if it causes all of the C_i to evaluate to 1; i.e. it causes the conjunction

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k$$

to evaluate to 1. When this happens, we say v is a *satisfying agreement* wrt $C_1, \ldots, C_k$ and the set of clauses $C_1, \ldots, C_k$ is *satisfiable*.

Satisfiability SAT problem Given a set of clauses $C_1, \ldots, C_k$ over a set of variables $X = \{x_1, \ldots, x_n\}$, does there exist a satisfying truth assignment?

3-SAT Given a set of clauses $C_1, \ldots, C_k$, each of length 3, over a set of variables $X = \{x_1, \ldots, x_n\}$, does there exist a satisfying truth assignment?

SAT and 3-SAT are fundamental combinatorial search problems.

Fact 3-SAT $\leq_P$ Independent Set

Fact If $Z \leq_P Y$ and $Y \leq_P X$, then $Z \leq_P X$ (reductions are transitive).

8.3 Efficient certification and definition of NP

$\mathcal{P}$ is set of all problems X for which $\exists$ algorithm A with polynomial run time that solves X.

Efficient certifier B is efficient certifier for problem X if

- B is polynomial time algorithm that takes two input arguments s and t, s is the guess of correct solution, t that is the "certificate" string, gives a way to check s is correct.
- $\exists$ polynomial function p s.t. for every string s, we have $s \in X$ iff $\exists$ string t s.t. $|t| \leq p(|s|)$ and $B(s, t) =$ yes.

Efficient certifier can be used as the core of a brute force algorithm. Try all strings and keep checking with the certifier.

$\mathcal{NP}$ is the set of problems for which $\exists$ an efficient certifier.

Fact $\mathcal{P} \subseteq \mathcal{NP}$

Million dollar question Is there a problem in $\mathcal{NP}$ that does not belong to $\mathcal{P}$? Does $\mathcal{P} = \mathcal{NP}$?

8.4 NP-Complete problems

NP-Complete A natural way to define the hardest problem. Such a problem X should have the following two properties

- $X \in \mathcal{NP}$
- $\forall\ Y \in \mathcal{NP}$, $Y \leq_P X$. I.e. every problem in $\mathcal{NP}$ can be reduced to X.

Fact Suppose X is an NP-Complete problem. Then X is solvable in polynomial time iff $\mathcal{P} = \mathcal{NP}$. If there is any NP-complete problem that cannot be solved in polynomial time, then no NP-complete problem can be solved in polynomial time.

Fact Circuit Satisfiability is NP-complete.

Fact If Y is an NP-complete problem, and X is a problem in $\mathcal{NP}$ with the property that $Y \leq_P X$, then X is NP-complete.

Fact 3-SAT is NP-complete.

Fact Independent set, set packing, vertex cover, and set cover are all NP-complete.

General Strategy to Prove New Problem is NP-Complete

Given a new problem X general strategy to show it is NP-complete:

1. Prove $X \in \mathcal{NP}$
2. Choose problem Y that is known to be NP-complete
3. Prove $Y \leq_P X$ i.e. consider an arbitrary instance s_Y of problem Y, and show how to construct, in polynomial time, an instance of s_X of problem X that satisfies
   - If s_Y is a yes instance of Y, then s_X is a yes instance of X
   - If s_X is a yes instance of X, then s_Y is a yes instance of Y (or contrapositve: if s_Y is a no instance of Y, then s_X is a no instance of X)

   In other words this establishes that s_X and s_Y have the same answer.

8.5 Sequencing problems

Type of computationally hard problem involving searching over the set of all *permutations* of a collection of objects.

Traveling Salesman Problem Given a set of distance on n cities and a bound D, is there a tour of length at most D? (Must visit all n cities and return to home city).

Hamiltonian Cycle Problem Given a directed graph $G = (V, E)$, say that a cycle C in G is a *Hamiltonian cycle* if it visits each vertex exactly once. Given a directed graph G, does it contain a Hamiltonian cycle?

Fact Hamiltonian cycle is NP-complete.

Fact Traveling Salesman is NP-complete.

Hamiltonian Path Problem Given a directed graph $G = (V, E)$, say a path P in G is a *Hamiltonian path* if it contains each vertex exactly once. (Path can start and end at any node). Given a direct graph G, does it contain a Hamiltonian path?

Fact Hamiltonian path is NP-complete.

8.6 Partioning Problems

3-Dimensional Matching Problem

Given disjoint sets X, Y, and Z, each of size n, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, does $\exists$ a set of n triples in T s.t. each element of $X \cup Y \cup Z$ is contained in exactly one of these triples.

Easy to show 3-D matching $\leq_P$ Set Cover and 3-D matching $\leq_P$ Set Packing but hard to show the other way (hard to use this to show NP-complete).

Fact 3-D Matching is NP-complete, use 3-SAT for reduction.

8.8 Numerical Problems

Subset Sum Problem

This is a special case of the Knapsack problem. Given natural numbers $w_1, \ldots, w_n$, and a target number W, is there a subset of $\{w_1, \ldots, w_n\}$ that adds up to precisely W?

We have seen an algorithm that can solve this problem already, the algorithm was *psuedo-polynomial*, i.e. it runs in time polynomial in the magnitude of input numbers, but not polynomial in the size of their respresentation.

Fact Subset sum is NP-complete, use 3-D matching for reduction.

Fact Scheduling with Release Times and Deadlines is NP-complete, use Subset sum for reduction.

Caveat: Subset sum with polynomially bounded numbers Consider special case of subset sum, with n input numbers, in which W is bounded by a polynomial function of n. Assuming $\mathcal{P} \neq \mathcal{NP}$, this special case is *not* NP-complete.

Component Grouping Given a graph that is not connected, and a number k, does there exist a subset of its connected components whose union has size exactly k? *It is incorrect to claim that Component Grouping is NP-complete.* You cannot establish that Subset sum $\leq_P$ Component Grouping, because it takes more than polynomial time to reduce.

8.9 Co-NP and the Asymmetry of NP

The definition of efficient certification is fundamentally asymmetric. Relating to intuition about NP, when we have a "yes" its a relatively short proof, but when we have a "no", no corresponding short proof is guaranteed by definition. Answer is simply no because we can't find a string that will serve as a proof.

For every problem X, there is a natural *complementary* $\bar{X}$: For all input strings s, we say $s \in \bar{X}$ iff $s \notin X$. Note if $X \in P$, $\bar{X} \in P$, because if we have algo A that solves X, we can simply produce $\bar{A}$ that runs A and flips the answer. This doesn't make it clear that if $X \in NP$, then $\bar{X} \notin NP$. This is because its hard to show that the $\exists$ in the definition of $\in NP$ transfers to become $\forall$ in the definition of $\notin NP$.

Co-NP Class of problems parallel to NP. A problem X belongs to Co-NP iff the complementary problem $\bar{X}$ belongs to NP.

$$\text{Does } NP = Co - NP \text{ ?}$$

Widespread belief is that $NP \neq Co - NP$. Proving $NP \neq Co - NP$ would be an even bigger step than proving $P \neq NP$ because:

Theorem If NP $\neq$ Co-NP, then P $\neq$ NP.

Good Characterizations: The Class NP $\cap$ Co-NP

If problem X belongs to both NP and Co-NP, when the answer is yes, there is a short proof of correctness, when the answer is no, there is also a short proof of correctness.

Good Characterization Problems that belong to the intersection NP $\cap$ Co-NP are said to have good characterization, there is always a nice certificate for the solution. The following have good characterization:

- Whether or not network flow contains flow of value at least v
- Bipartite perfect matching problem

If a problem in P, then it belongs to both NP and Co-NP, thus P $\subseteq$ NP $\cap$ Co-NP. Open question:

$$\text{Does } P = NP \cap Co - NP \text{ ?}$$

General opinions are very mixed on this problem.

8.10 Partial Taxonomy of Hard Problems

Packing Problems

You're given a collection of objects, and you want to choose at least k of them; making your life difficult is a set of conflicts among the objects, preventing you from choosing certain groups simultaneously.

Independent Set Given a graph G and a number k, does G contain an independent set of size at least k.

Set Packing Given a set U of n elements, a collection $S_1, \ldots, S_m$ of subsets of U, and a number k, does there exist a collection of at least k of these sets with the property that no two of them intersect?

Covering Problems

You're given a collection of objects, and you want to choose a subset that collectively achieves a certain goal; the challenge is to achieve this goal while choosing only k of the objects.

Vertex Cover Given a graph G and a number k, does G contain a vertex cover of size at most k?

Set Cover Given a set U of n elements, a collection $S_1, \ldots, S_m$ of subsets of U, and a number k, does there exist a collection of at most k of these sets whose union is equal to all of U?

Partitioning Problems

Partitioning problems involve a search over all ways to divide up a collection of objects into subsets so that each object appears in exactly one of the subsets.

3-Dimensional Matching Given disjoint sets X, Y, and Z, each of size n, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, does there exist a set of n triples in T so that each element of $X \cup Y \cup Z$ is contained in exactly one of these triples?

Graph Coloring Given a graph G and a bound k, does G have a k-coloring?

Sequencing Problems

Searching over the set of all permutations of a collection of objects.

Hamiltonian Cycle Given a directed graph G, does it contain a Hamiltonian cycle?

Hamiltonian Path Given a directed graph G, does it contain a Hamiltonian path?

Traveling Salesman Given a set of distances on n cities, and a bound D, is there a tour of length at most D? (must hit all cities).

Numerical Problems

Subset Sum Given natural numbers $w_1, \ldots, w_n$, and a target number W, is there a subset of $\{w_1, \ldots, w_n\}$ that adds up to precisely W?

Constraint Satisfaction Problems

3-SAT Given a set of clauses $C_1, \ldots, C_k$, each of length 3, over a set of variables $X = \{x_1, \ldots, x_n\}$, does there exist a satisfying truth assignment? (allegedly most useful for reductins, SAT is also good).

Chapter 11: Approximation Algorithms

How to design algorithms for problems where polynomial time is probably unattainable? Approximation algorithms run in polynomial time and find solutions that are *guaranteed* to be *close* to the optimal solution. In order to prove our approximation is guaranteed to be close to optimal, we need to compare it to the optimal solution - which is very hard to find, making these kinds of problems inherently tricky.

Four general techniques

- greedy algorithms
- pricing method (aka primal-dual technique)
- linear programming, integer programming
- dynamic programming on rounded version of input

11.2 The Center Selection Problem

Given an integer k, a set S of n sites (towns), and a distance function that satisfies the following natural properties

- $dist(s, s) = 0 \; \forall \; s \in S$
- distance is symmetric: $dist(s, z) = dist(z, s) \; \forall z, s \in S$
- triangle inequality: $dist(s, z) + dist(z, h) \geq dist(s, h)$

Let C be a set of centers. Assume that people in a given town will shop at the closest mall. Define the distance from a site s to the centers as $dist(s, C) = \min_{c \in C} dist(s, c)$. Say that the set of centers C forms an r cover if each site is within distance at most r from the centers ($dist(s, C) \leq r$ for all sites $s \in S$). The minimum r for which C can r-cover will be called the *covering radius* of C and is denoted $r(C)$ (farthest anyone needs to travel to get to his or her nearest center). Goal is to select a set C of k centers for which $r(C)$ is as small as possible.

Simple greedy algorithm Put the first center at the best possible location for a single center, then keep adding centers so as to reduce the covering radius each time by as much as possible.

This is too simple to be effective. Consider only two sites and $k = 2$, greedy selects halfway between first, but the optimal is to put a center on top of each site.

Better greedy algorithm that relies on knowing optimal ahead of time

S' is set of sites that still need to be covered
Initialize $S' = S$
Let $C = \emptyset$
while $S' \neq \emptyset$ **do**
    Select any site $s \in S'$ and add s to C
    Delete all sites from S' that are at distance at most $2r$ from s
end while
if $|C| \leq k$ **then**
    return C as the selected set of sites
else
    Claim (correctly) that there is no set of k centers with covering radius r
end if

Fact Any set of centers C returned by the algorithm has covering $r(C) \leq 2r$

Fact Suppose the algorithm selects more than k centers. Then, for any set C^* of size at most k, the covering radius is $r(C^*) > r$

Problem is that algorithm above relies on knowing optimal radius r.

Eliminating the assumption of knowing optimal radius Iteratively maintain two r values, $r_0 < r_1$, s.t. optimal radius is always $> r_0$ but less than $2r_1$. Run algorithm above with radius $r = (r_0 + r_1)/2$.

Greedy algorithm that works Select site s that is farthest away from all previously selected centers. If there is any site at least $2r$ away from previously chosen centers, then this farthest site s must be one of them.

Assume $k \leq |S|$

Select any site s and let $C = \{s\}$
Let $C = \emptyset$
while $|C| < k$ **do**
  Select any site $s \in S$ that maximizes $dist(s, C)$
  Add site s to C
end while
return C as the selected set of sites

Fact This greedy algorithm returns a set C of k points s.t. $r(C) \leq 2r(C^*)$, where C^* is an optimal set of points

This is a 2-approximation.

11.6 Linear Programming and Rounding: An Application to Vertex Cover

Linear Programming as a General Technique

Consider the problem of determining a vector x that satisifies $Ax \geq b$. Given a region defined by $Ax \geq b$ linear programming seeks to minimize a linear combination of the coordinates of x over all x belonging to the region. Such a linear combination can be written as $c^t x$, where c is a vector of coefficients, $c^t x$ denotes inner product of the two vectors.

Standard form for linear programming optimization problem is as follows:

> 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 the following optimization problem
>
> $$\min(c^t x \text{ s.t. } x \geq 0;\ Ax \geq b)$$

$c^t x$ is often called the *objective function* and $Ax \geq b$ is the set of constraints.

Can phrase linear programming as a decision problem via

> Given a matrix A, vectors b and c, and a bound γ, does $\exists\ x$ s.t. $x \geq 0$, $Ax \geq b$, and $c^t x \leq \gamma$.

Computational complexity of linear programming

The decision version of linear programming is in NP and also in Co-NP, this follows from linear programming duality. Most widely used algorithm for linear programming is the *simplex method* but worst case run time is known to be exponential. Linear programming is good example for thinking about the limits of polynomial time as a formal definition of efficiency.

Vertex Cover as Integer Program

Recall weighted vertex cover as typically defined.

We make decision variable x_i for each node $i \in V$ to model choice of whether to include node i in the vertex cover; $x_i = 0$ means node i is not in the vertex cover, $x_i = 1$ means node i is in vertex cover. Create single n-dimensional vector x in which the ith coordinate corresponds to the ith decision variable x_i. Linear inequalities to encode requirement that nodes form a vertex cover and use objective function to encode goal of minimizing total weight. For each edge $(i, j) \in E$, it must have one end in the vertex cover, we write this as $x_i + x_j \geq 1$. To express minimization, we

write the set of node weights as n dimensional vector w, with the ith coordinating to w_i, and then minimize $w^t x$. This looks like

$$\text{(VC. IP) } \min \sum_{i \in V} w_i x_i$$

$$s.t. \;\; x_i + x_j \geq 1 \;\; (i,j) \in E$$

$$x_i \in (0,1) \; i \in V$$

Fact S is a vertex cover in G iff the vector x, defined as $x_i = 1$ for $i \in S$, and $x_i = 0$ for $i \notin S$, satisfies the constraints in (VC.IP). Further, we have $w(S) = w^t x$.

Integer programming Linear programming, but with the condition that solutions must be integer valued. Above is integer programming.

Fact Vertex Cover $\leq_P$ Integer Programming

Fact The special case of 0-1 integer programming is NP-complete.

Using Linear Programming for Vertex Cover

Take (VC.IP) - which was too hard to solve - and drop requirement that every x_i must be 0 or 1. This gives us an instance of (VC.LP). We can solve this in polynomial time because it is just normal linear programming, i.e. we can find a set of values $\{x_i^*\}$ between 0 and 1 s.t. $x_i^* + x_j^* \geq 1$ for each edge (i,j), and $\sum_i w_i x_i^*$ is minimized. Let x^* be this vector and $w_{LP} = w^t x^*$ denote the value of the objective function.

Fact Let S^* denote a vertex cover of minimum weight. Then $w_{LP} \leq w(S^*)$.

How can solving this linear program help us actually find a near-optimal vertex cover? Idea is to work with values x_i^* and infer a vertex cover S from them. Given a fractional solution $\{x_i^*\}$, define $S = \{i \in V : x_i^* \geq 1/2\}$ - round values at least 1/2 up, and those below 1/2 down.

Fact The set S defined in this way is a vertex cover, and $w(S) \leq w_{LP}$.

Fact The algorithm produces a vertex cover S of at most twice the minimum possible weight.

Chapter 13: Randomized Algorithms

Randomized min-cut

Given an undirected graph $G = (V, E)$, define a cut of G to be a partition of V into two non-empty sets A and B. For a cut (A, B), the size of (A, B) is the number of edges with one end in A and the other end in B. A *global minimum cut* is a cut of minimum size. We will use a contraction algorithm to find the min cut.

Contraction algorithm Start with a connected multigraph $G = (V, E)$; i.e. an undirected graph that is allowed to have multiple parrallel edges between the same pair of nodes. Choose an edge $e = (u, v)$ of G at random and *contract* it. From this we get a new graph G' where u and v are now a single node w, all other nodes the same. Edges that had one end equal to u and the other end equal to v are deleted from G'. Each other edge is preserved, updating its ends accordingly if need be. Algo continues recursively, choosing edge uniformly at random and contracting it and terminates when it reaches a graph with only two "super nodes" v_1

and v_2. Each node as a subset $S(v_i) \subseteq V$ consisting of nodes contracted into them, and the two subsets $S(v_1)$ and $S(v_2)$ form a partition of V. Output $(S(v_1), S(v_2))$ as the cut found by the algorithm.

Fact The contraction algo returns a global min cut of G with probability at least $1/\binom{n}{2}$

Fact An undirected graph $G = (V, E)$ on n nodes has at most $\binom{n}{2}$ global min-cuts.

Linear-time median finding

Suppose we are given a set of n numbers $S = \{a_1, a_2, \ldots, a_n\}$. With randomized approach we can find the median in $O(n)$ time, which is better than sorting in $O(n \log n)$ and finding it that way.

First step to this is to develop a way to select the kth largest element of a set of n numbers, S (k between 1 and n). Once we have this, we can call $select(S, n/2)$ or $select(S, (n+1)/2)$ to get the median of S. Algorith is as follows

function SELECT(S,k)
  Choose a splitter $a_i \in S$
  for each element a_j of S **do**
    Put a_j in S^- if $a_j < a_i$
    Put a_j in S^+ if $a_j > a_i$
  end for
  if $|S^-| = k - 1$ **then**
    The splitter a_i was in fact the answer
  else if $|S^-| \geq k - 1$ **then**
    The kth largest element lies in S^-
    Recursively call SELECT(S^-, k)
  else
    Suppose $|S^-| = l < k - 1$
    The kth largest element lies in S^+
    Recursively call SELECT$(S^+, k - 1 - l)$
  end if
end function

Fact Regardless of how the splitter was choosen, the algo returns the kth largest element of S.

Rule Choose splitter $a_i \in S$ uniformly at random.

Fact Expected running time of SELECT(n, k) is $O(n)$ when choosing splitter uniformly at random.

Hashing

There is a universe U of possible elements that is extremely large. Data structure is trying to keep track of a set $S \subseteq U$ whose size is generally a negligible fraction of U. Goal is to be able to insert and delete elements from S and quickly determine whether a given element belongs to S.

Hash function basic idea of hasing is to work with array of size $|S|$ instead of one the size of U. Suppose we want to be able to store a set S of size n. Set up an array H of size n to store

information, and use function $h : U \to \{0, 1, \ldots, n-1\}$ that maps elements of U to array positions. h is *hash function*, H is *hash table*.

- to add $u \in S$, place u at position $h(u) \in H$
- works perfectly if for all distinct $u, v \in S$, $h(u) \neq h(v)$, this is not true in real world, elements can "collide"

Good hash function Use randomization.

Fact Let $\mathcal{H}$ be a universal set of hash functions mapping universe U to set $\{0, 1, \ldots, n-1\}$, let S be arbitrary subset of U of size at most n, and let u be any element in U. Define X to be random variable equal to the number of elements $s \in S$ s.t. $h(s) = h(u)$, for a random choice of hash function $h \in \mathcal{H}$. S and u are fixed and randomness is the choice of $h \in \mathcal{H}$. Then $E[X] \leq 1$.

Universal class of hash functions Let $\mathcal{A}$ be set of all vectors of the form $a = (a_1, \ldots, a_r)$ where a_i is an integer in $[0, p-1]$ for each $i = 1, \ldots, r$. For each $a \in \mathcal{A}$, define linear function

$$h_a(x) = \Big(\sum_{i=1}^{r} a_i x_i \Big) \mod p$$

and then the family of hash functions is $\mathcal{H} = \{h_a : a \in \mathcal{A}\}$

Prime testing

Input is n a large number. To find a prime, select a random n and test if its prime. Challenge is to test if a number n is a prime.

Number theory If p prime and $a \not\equiv 0 \mod p$ then $\alpha \neq \beta \implies \alpha a \not\equiv \beta a \mod p$

Fermat's little theorem If p prime and $1 < a < p$ then $a^{p-1} \equiv 1 \mod p$

Prime testing using Fermat's theorem

- select $a \in [2, p-1]$
- compute $a^{p-1} \mod p$
- if $a^{p-1} \mod p \not\equiv 1 \mod p$ then p is not prime (but this doesn't mean that if it is $\equiv 1$ p is prime)

Fact If $\exists$ any a relatively prime to p s.t. $a^{p-1} \not\equiv \mod p$ then random a has probability $\geq 1/2$ of failing the test

Charmical number is n s.t. $a^{n-1} \equiv 1 \mod n \; \forall \; a$ and n relatively prime

- $\exists$ infinite Charmical numbers, but they are very rare
- these are the numbers for which in our test above they fail the "not prime" condition, but still are not prime numbers, but we can probably ignore them because they are so rare, this means that if $a^{p-1} \mod p \not\equiv 1 \mod p$ then p is not prime and otherwise we say that it is prime