Analysis of Algorithms

1 Greedy Algorithms

1.1 Coin Change Problem

Given a set of fixed integer coin denominations: $c_1 = 1, c_2, \ldots, c_k$, the problem is to determine the optimal way to pay the given amount A, that is, to minimize $x_1 + \ldots + x_k$ subject to $c_1x_1 + \ldots c_kx_k = A$, where x_i is the number of coins of value c_i used.

The greedy algorithm is as follows:

```
for l = k to 1
        print(g_l = ⌊A/c_l⌋ of c_l)
        A ← A − c_l · ⌊A/c_l⌋
```

For example, consider the denominations $\{1, 5, 10, 20\}$. The question is whether the greedy algorithm obtains the optimal solution, i.e. whether $g_\ell = x_\ell$ for $1 \leq \ell \leq k$.

Proof. Take an optimal solution for A, say $A = x_1 + \ldots + 20x_4$. It follows that

$$0 \leq x_1 \leq 4$$
$$0 \leq x_2, x_3 \leq 1.$$

Hence $x_1 + 5x_2 + 10x_3 \leq 19$. So indeed $g_4 = \lfloor \frac{A}{20} \rfloor = x_4$. Take $R_1 = A \bmod 20$, then since $x_1 + 5x_2 \leq 9$, again we have $g_3 = \lfloor \frac{R_1}{10} \rfloor = x_3$. Similarly, we could show that g_1 and g_2 are optimal. ∎

As before, we can also show that the greedy algorithm is optimal for the set $\{1, 3, 7\}$. Unfortunately, this is not always the case. Consider $\{1, 5, 11\}$ and $A = 20$. The optimal choice is $(x_1, x_2, x_3) = (0, 4, 0)$, however, the greedy algorithm yields $(4, 1, 1)$.

1.1.1 Coin Change: Dynamic Programming.

Unlimited Supply. Define $S[0..A]$ where $S[a]$ is the minimum number of coins to pay amount a. Then

$$S[a] = \begin{cases} 0, & \text{if } a = 0 \\ 1 + \min_{\substack{1 \leq i \leq k \\ c_i \geq a}} S[a - c_i], & \text{otherwise} \end{cases}$$

Proof. If you can pay $a - c_i$ using l coins, then you can pay a using $l + 1$ coins. Hence $S[a] \leq 1 + S[a - c_i]$ for all $1 \leq i \leq k$, $c_i \leq a$. Now suppose some optimal solution for a uses coin c_i, then $S[a] = 1 + S[a - c_i]$, so equality holds for some i and the claim is valid. ∎

We can extract the solution by maintaining a second array, $P[0..a]$ where $P[a] = \arg\min S[a - c_i]$.

Limited Supply. Suppose we have coins $c_1, \ldots, c_k$ and amount A. Find an algorithm that determines whether we can pay A exactly.

Let $T[a, l]$ be true if you can pay amount a using a subset of coins $c_1, \ldots, c_l$, and false otherwise. Observe that if $T[a, l-1]$ or $T[a - c_l, l - 1]$ is true, then $T[a, l]$ is true. Similarly, the converse also holds. Hence

$$T[a, l] = T[a, l-1] \texttt{ OR } T[a - c_l, l - 1].$$

Now suppose, instead, we wish to compute the minimum number of coins to pay A. First, observe that one of the following two inequalities must hold

$$T[a, l] \leq \begin{cases} T[a, l-1] \\ 1 + T[a - c_l, l - 1]. \end{cases}$$

Hence if we define $T[0..a, 0..l]$ such that

$$\begin{cases} T[0, 0] = 0 \\ T[a, 0] = \infty, 1 \leq a \leq A \\ T[a, l] = \min(T[a, l-1], 1 + T[a - c_l, l - 1]), 1 \leq a \leq A, 1 \leq l \leq k \end{cases},$$

then our output is $T[A, k]$ (a return of ∞ indicates that A is not payable using the specified coin values). On the other hand, if we wanted to compute the maximum number of coins to pay A, we may simply initialize to $-\infty$ and switch the min with max.

1.2 Knapsack Problem

Given n items of value $v_1, \ldots, v_n$ with weights $w_1, \ldots, w_n$ and a total weight limit L, determine

$$\max_{\substack{S \subset \{1,\ldots,n\} \\ \sum_{i \in S} w_i \leq L}} \sum_{i \in S} v_i.$$

Let $T[l, i]$ be the maximum value of a subset of the first i items with weight at most l. Then

$$T[l, i] = \max \begin{cases} T[l, i-1] \\ v_i + T[l - w_i, i - 1] \end{cases}$$

Our base case if $T[l, 0] = 0$.

Suppose that we know our values are small relative to our weight. Can we design a more efficient algorithm in this scenario?

Let $S[i, v]$ be the minimum weight of a subset of the first i items with total value at least v and let $S[i, v] = \infty$ if v cannot be achieved with the first i items. Also set $S[i, v] = 0$ if $v < 0$. Then

$$S[i, v] = \min \begin{cases} S[i-1, v] \\ S[w_i + S[i-1, v - v_i]] \end{cases}$$

Our maximum value is thus $\arg\min_{0 \leq k \leq V} \{S[n, k] \leq L\}$, where $V = \sum v_i$.

1.2.1 Knapsack Counting Problem

Given n coins with value $c_1, \ldots, c_n$, how many different ways are there to pay amount A exactly?

Let $S[a, i]$ be the number of ways of paying a with a subset of the first i coins, so

$$\begin{cases} S[a,i] = S[a,i-1] + S[a-c_i, i-1], 1 \leq i \leq n, 0 \leq a \leq A \\ S[a,0] = 0, a \neq 0 \\ S[0,0] = 0 \end{cases}.$$

1.3 Subsequences

Given real numbers $a_1, \ldots, a_n$ find the longest increasing subsequence. Let $T[l]$ be the length of the longest increasing subsequence of the first l numbers that ends with a_l, then

$$T[l] = \max_{1 \leq i < l} \begin{cases} 1 \\ T[i] + 1; a_l > a_i \end{cases} \tag{1}$$

Longest Common Subsequence. Given two sequences $(a_i)_{i=1}^n, (b_i)_{i=1}^m$, let $T[u, w]$ be the length of the longest common subsequence of $(a_i)_{i=1}^u, (b_i)_{i=1}^w$. Then, if $a_u \neq b_w$

$$T[u,w] = \max \begin{cases} T[u-1, w] \\ T[u, w-1], \end{cases} \tag{2}$$

but if $a_u = b_w$, then $T[u, w] = 1 + T[u-1, w-1]$.

Other problems.

(a). Longest common increasing subsequence.

(b). Increasing subsequence with largest sum.

(c). Longest convex subsequence.

1.4 Activity Selection Problem

Given intervals $I_1, \ldots, I_n$, with $I_i = [a_i, b_i]$, $a_i \leq b_i$, together with values, v_i, associated with each interval, we want to select a subset $S \subset \{1, \ldots, n\}$ such that there are no overlaps in the selected intervals, and which maximizes $\sum_{i \in S} v_i$. If $T[k]$ is the maximum total sum of a valid subset of the first k activities then

$$T[k] = \max \begin{cases} T[k-1], \\ v_k, \\ \max_{\substack{1 \leq i \leq k-1 \\ a_k > b_i}} T[i] + v_k \end{cases} \tag{3}$$

1.5 Matrix Multiplication

Given $a_1, \ldots, a_{n+1}$ positive integers corresponding to dimensions of matrices $A_1, \ldots, A_n$ (matrix A_i is an $a_i \times a_{i+1}$ matrix), we want to mind the optimal way to associate the matrices in the product

$A_1 \dots A_n$, so that we perform the fewest total multiplications. Let $T[i,j]$ be the optimal cost of multiplying matrices of dimensions $a_i, \dots, a_i$. Then $T[i, i+1] = 0$ and

$$T[i,j] = \min_{i<k<j} \{T[i,k] + T[k,j] + a_i a_k a_j\}$$

assuming $i \leq j+1$.

1.6 Traveling Salesman

Given an $n \times n$ matrix D, where D_{ij} is the distance from city i to city j, we want to output a path of minimal length, starting at 1 and ending at city n, that visit every city exactly once. Let $T[a, b, S]$ be the minimal cost of a path from city a to city b visiting only cities in S in between (where $S \subset \{1, \dots, n\} \backslash \{a, b\}$). Then $T[a, b, \varnothing] = D_{ab}$ and

$$T[a, b, S] = \min_{s \in S} \{T[s, b, S \backslash \{s\}] + D_{as}\}.$$

Since T has $n^2 \cdot 2^{n-2}$ entries, this algorithm is $O(n^3 \cdot 2^n)$. Note however, that our recursion is invariant with respect to b, so we may simply compute $T[a, S]$, improving our algorithm by a factor of n.

2 Graph Algorithms

Let $s, t \in V$ and $s - t$ *walk* is a sequence of vertices $s = v_1, v_2, \dots, v_n = t$ such that $\{v_i, v_{i+1}\} \in E$ for all $1 \leq i < k$. An $s - t$ *path* is a walk with no repeated vertices.

If there is an $s - t$ walk, then there is an $s - t$ path (formed by removing loops from the walk).

A graph is *connected* if for any $u, v \in V$ there exists an $u - v$ path. A disconnected graph can have at most $\binom{n-1}{2}$ edges. A connected graph must have at least $n - 1$ edges.

Proof. Let $G(V, E)$ be connected. Suppose $v \in V$ has $\deg(v) = 1$. Let u be the vertex adjacent to v. Let $G' = (V \backslash \{v\}, E \backslash \{\{u, v\}\})$. Then it is clear that G' is connected. Any walk which used vertex v had a loop starting at u, so we can remove v from the walk. Now suppose G is connected with $n \geq 1$, $m < n - 1$ and n is as small as possible. G can have no vertex of degree 0, else it wouldn't be connected. It can have no vertex of degree 1, else we could remove the vertex and its only edge, to obtain a smaller graph still satisfying the condition. So every vertex in G has degree at least 2. But then $2m = \sum \deg(v) \geq 2n$, so $m \geq n$ a contradiction. ∎

A *tree* is a connected graph with $n - 1$ edges.

2.1 Connectivity.

Is G undirected a connected graph?

```
for all v ∈ V :
    visited[v] ← false

explore(1)
```

```
check for all v ∈ V whether visited[v] = true
if so, return 'Connected', else return 'Disconnected'
```

```
explore(v):
    visited[v] ← true
    for u ∈ N(v) :
        if not visited[u]
            explore(u)
```

Runtime $O(n+m)$.

Theorem 1. *While* `explore`(v) *is on the 'call stack',* `explore`(u) *will execute for all* u *reachable from* v *by a path of unvisited vertices.*

Proof. Note that any unvisited neighbor of v is sure to be called while `explore`(v) executes. Now suppose u (unvisited) is reachable from v by by a path of unvisited vertices $v = v_1, v_2, \ldots, v_{k-1}, v_k = u$. Let v_i be the first vertex for which `explore`(v_i) is called. Note this happens while `explore`(v) is still executing since `explore`(v_2) must execute before `explore`(v) finishes. It follows that there is an unexplored path from v_i to u of length less than k, so by induction, we're done. ∎

2.1.1 Connectivity in Digraphs

A digraph is *strongly connected* if for all $s, t \in V$ there exists an $s-t$ path. Note that a strongly connected graph requires at least n edges, as every vertex must have an exit. Let $N^O(v)$ be the set of out-neighbors of v and define $N^I(v)$ analogously. The out-degree of v is simply $|N^O(v)|$ and in-degree is defined analogously. Observe that the sum of the out-degrees (in-degrees) of each vertex is m.

Theorem 2. *A digraph* G *is strongly connected if and only if there exists a vertex* $v \in V$ *so that every vertex* $u \in V$ *is reachable from* v *and* v *is reachable from any* u*.*

The above theorem gives us a $O(n+m)$ algorithm to test for connectivity of a digraph, using the algorithm for an undirected graph.

```
for all v ∈ V :
    visited[v] ← false
explore(1)
if some visited[v] = false, then exit
else:
    compute G^R #reverse of G
    reset visited to false
    explore(1^R)
    check for all visited[v^R] = true
```

2.2 Directed Acyclic Graph

A *direct acyclic graph* (DAG) is a digraph with no cycles. A DAG can have at most $n(n-1)/2$ edges. (Form equivalence classes $\{(a,b),(b,a)\}$ where $a, b \in V$, then choose one edge from every equivalence class.)

Theorem 3. *If G is a DAG and $|V| \geq 2$, then G is not strongly connected. In particular, every DAG contains a vertex with in-degree 0.*

2.3 Depth First Search

```
DFS(G): # O(m + n)
    c ← 0
    for all v ∈ V:
        visited[v] ← false
    for all v ∈ V:
        if not visited[v] then explore(v)

explore(v):
    visited[v] ← true
    for all out_neighbors u of v:
        if not visited[u] then explore(u)
    post[v] ← c
    c = c + 1
```

Problem. *Given a DAG G output a topological sorting of its vertices. That is, a list $v_1, \ldots, v_n$ of its vertices so that for all $1 \leq i < j \leq n$ we have $(v_j, v_i) \notin E$.*

Algorithms.

```
while G non-empty: # O(n^2)
    for all v ∈ V:
        if in-deg(v) = 0:
            print v
            remove v from G
            for all out-neighbors u of v:
                in-deg[u]--
```

```
for all v ∈ V: # O(m + n)
    if in-deg[v] = 0:
        add v to L
while G non-empty:
    get v ∈ L:
        print v
        remove v from G
        for all out-neighbors u of v:
            in-deg[u]--
            if in-deg[u] = 0, then add u to L
```

```
DFS(G) # O(m + n)
print vertices in decreasing post value
```

Lemma 1. *Let G be a DAG. Let $(u, v) \in E$. Then `post[u]` > `post[v]`.*

Proof. If `explore`(u) is called before `explore`(v), then `explore`(v) must finish before `explore`(u). Hence `post`$[u] >$ `post`$[v]$.

Conversely, `explore`(u) cannot be called during `explore`(v), as v is not reachable from u (otherwise, G would not be a DAG). Hence `explore`(v) finishes before `explore`(u), so we're done. ∎

2.3.1 Strongly Connected Components

Given a directed graph $G(V, E)$, define the relation $\sim$ on V via $u \sim v$ if and only if there is a $u - v$ path and a $v - u$ path in G. Then $\sim$ is an equivalence relation. We call its equivalence classes the *strongly connected components* of G. Alternatively, we may define this as the maximal strongly connected subsets of V.

Note that given any digraph $G(V, E)$ we can create a new graph $G'(V', E')$ where the vertices of G' are the strongly connected components of G and $(U, V) \in E'$ if and only if there exist $u \in U$ and $v \in V$ such that $(u, v) \in E$. Moreover, G' is a DAG.

Problem. *Given digraph $G(V, E)$ return $v \in V$ where v is in a source of the DAG of SCCs.*

```
DFS(G)
print arg max_{v∈V}{post[v]}
```

Similarly, to calculate a vertex in a sink, we'd run DFS on G^R. The vertex with the largest post value in G^R is in a sink SCC of G.

Algorithm. *(Strongly connected components).*

```
#O(m + n)
DFS(G^R)
clear visited[], c ← 0
for each v ∈ V in decreasing order of post value:
    if not visited[v] then explore(v)
    c++
```

Let $v^1, \ldots, v^n$ be a list in the vertices in G in decreasing order of post value after the call $\texttt{DFS}(G^R)$. Suppose $v_1^1, \ldots, v_{n_1}^1$ are visited by a call to `explore` on v^1. So $v_1^1, \ldots, v_{n_1}^1$ are reachable from v^1 in G. It must be that v^1 is reachable from v_i^1 for each $1 \leq i \leq n_1$, otherwise v^1 would have greater post value than each of the v_i^1. Therefore, the vertices $\{v^1, v_1^1, \ldots, v_{n_1}^1\}$ form a strongly connected component. Continuing, we see that if we call `explore` on v^i and vertices $v_1^i, \ldots, v_{n_i}^i$ are visited for the first time. Then $\{v^i, v_1^i, \ldots, v_{n_i}^i\}$ form a strongly connected component. To form the DAG of SCCs, we just need to add edges between SCCs, but this is easily done in $O(m)$.

2.4 Shortest Path

Problem. *Given a digraph $G(V, E)$ and a function $l : E \to \mathbb{R}$ (such that no cycle of G has a negative weight), compute the distance matrix $D \in \mathbb{R}^{n \times n}$ where $D[u, v]$ is the length of the shortest path from u to v. The length of the a path $v_1, \ldots, v_k$ is $\sum_{i \in \{1, \ldots, k-1\}} l((v_i, v_{i+1}))$.*

Algorithm. (*Floyd-Warshall*).

```
for i,j in 1...n: #O(n^3)
    D[i,j] = (u, v) ∈ E ? l((i, j)) : ∞
for k,i,j in 1...n:
    D[i,j] = min{D[i, k] + D[k, j], D[i, j]}
```

$O(n^3)$ in time, $O(n^2)$ in space. Note: If cycles with negative weights are allowed, then the problem is $\mathcal{NP}$-complete.

Algorithm. Single source, s, shortest path, no negative cycles. $w : E \to \mathbb{R}$ (*Dijkstra*).

```
for all v ∈ V:
    D[v] = ∞, F[v] = false
D[s] = 0
Pick v ∈ V such that F[v] = false and minimizes D[v]:
    for all out-neighbors u of v:
        D[u] = min{D[u] , D[v] + w((v,u))}
    F[v] = true
```

The given algorithm is $O(n^2)$. If instead a priority queue is used to retrieve the vertex with smallest distance, then we improve to $O(m \log n)$. For Fibonacci heaps (amortized constant-time key updates), this improves further to $O(m + n \log n)$. By running Dijkstra on all vertices of a graph, we have a $O(mn + n^2 \log n)$ algorithm to compute all-pairs-shortest-path.

Algorithm. Single source, s, shortest path, no negative cycles. $w : E \to \mathbb{R}$ (*Bellman-Ford*).

```
repeat n − 1 times: #O(mn)
    for all (u,v) ∈ E:
        D[v] = min{ D[v] , D[u] + w((u,v))}
```

See notes for proof of Bellman-Ford and Floyd-Warshall.

(Shortest path in a DAG). Initialize $D \leftarrow \infty$. Set $D[s] = 0$. For u in topological order,

$$D[v] = \min_{v \in \text{out-neigh}(u)} D[u] + w((u, v)).$$

Let $G(V, E)$ be a directed graph with edge weights $w : E \to \mathbb{R}$ (no negative cycles). Let $\Phi : V \to \mathbb{R}$ be a function. Define $\hat{w}((u, v)) = w((u, v)) + \Phi(u) - \Phi(v)$. Shortest paths are invariant under this transformation.

Algorithm. (*Johnson*).

1. Run Bellman-Ford for some vertex $s \in V$.
2. Let $\Phi(v) = -D[s, v]$. Use Φ to transform the edge weights of G. Observe that under this transformation the edge weights of G are non-negative.
3. Run Dijkstra's for all $v \in V$.

This is a $O(mn + n^2 \log n)$ algorithm to compute shortest paths in a graph with no negative cycles.

2.5 Matchings

A *matching* in G is a subset $\mathcal{M} \subset E$, such that $e \cap f = \varnothing$ for any $e, f \in M$.

Let $G(V, E)$ be an undirected graph and $\mathcal{M}$ a matching in G. A path $P = v_1, \ldots, v_k$ is called an *augmenting path* in $\mathcal{M}$ if

1. v_1 and v_k are not matched;
2. every other edge in P is matched (i.e. $\{v_2, v_3\}, \{v_4, v_5\}, \ldots, \{v_{k-2}, v_{k-1}\}$ are in M.)

Theorem 4. *If $\mathcal{M}, \mathcal{M}'$ are two matchings then $\mathcal{M} \cup \mathcal{M}'$, consists of common edges, alternating even cycles, and alternating paths.*

Proof. We cannot have odd length cycles, else either $\mathcal{M}$ or $\mathcal{M}'$ would not be a matching. For the same reason, even cycles and paths must alternate. ∎

Theorem 5. *If $\mathcal{M}$ is not maximal, then there exists an augmenting path.*

Proof. Suppose $\mathcal{M}$ is not maximal. Let $\mathcal{M}^*$ be some maximal matching. Thus $|\mathcal{M}^*| > |\mathcal{M}|$. Note that $\mathcal{M} \cup \mathcal{M}^*$ cannot consist only of alternating even cycles, else $|\mathcal{M}^*| = |\mathcal{M}|$. So there must be some odd length alternating path, starting and ending with a matched edge in $\mathcal{M}^*$. This is an augmenting path in $\mathcal{M}$. ∎

Problem. *Find an augmenting path in $\mathcal{M}$ given an undirected bipartite graph $G(V, E)$, where $V = L \cup R$.*

Create a directed graph $G'(V', E')$ where $V' = V$ and for $\{u, v\} \in E$, with $u \in L$ and $v \in R$ we put $(v, u) \in E'$ if $\{u, v\} \in \mathcal{M}$, otherwise, $(u, v) \in E'$. That is, matched edges with point from right to left, and vice versa for unmatched edges. Then it suffices to find a path in G' from an unmatched vertex in L to a matched vertex in R. We can do this by adding two vertices s, t not in V such that s connects to every vertex in L and t connects to vertex in R. Then run DFS, starting at s. If t is reached, we have an augmenting path.

```
M ← ∅
repeat
    find augmenting path P for M
    M ← M ⊕ P
until no augmenting path
```

Here we define $\mathcal{M} \bigoplus P$ to be the symmetric difference, $(A - B) \cup (B - A)$, of the edges in the two sets. The augmenting path has l edges in $\mathcal{M}$ and $l+1$ edges not in $\mathcal{M}$. So the symmetric difference has cardinality one more than $\mathcal{M}$.

2.5.1 Weighted Matchings

Assume we have an undirected graph G with a edge weight function w. We now wish to find a matching $\mathcal{M}$ that maximizes $w(\mathcal{M}) \equiv \sum_{e \in \mathcal{M}} w(e)$. Given an augmenting path P define

$$\widehat{w}(P) = w(v_1, v_2) - w(v_2, v_2) + \ldots - w(v_{k-2}, v_{k-1}) + w(v_{k-1}, v_k).$$

Then $w(\mathcal{M} \bigoplus P) = w(M) + \widehat{w}(P)$.

```
M ← ∅
repeat
    find augmenting path P maximizing ŵ(P) for M
    M ← M ⊕ P
until no augmenting path
```

Let $M_0, M_1, \ldots, M_k$ be the sequences of matchings obtained as the above algorithm executes.

Claim 1. *Then M_k is the maximum weight matching among matchings of size k.*

Problem. *How to find an augmenting path maximizing $\widehat{w}$ on $\mathcal{M}$, for a bipartite graph $G(V, E)$.*

We create a directed graph G' as in the unweighted case. If $\{u, v\} \in E$, and $\{u, v\} \in \mathcal{M}$, then add (v, u) to E' with weight $w(\{u, v\})$. If $\{u, v\}$ is unmatched, then add (u, v) to E' but with weight $-w(\{u, v\})$. Then the length of an augmenting path P in G' is $-\widehat{w}(P)$ had the weights to unmatched edges not been negated. Therefore, the shortest path in G', corresponds to an augmenting path maximizing $\widehat{w}(P)$. Now we prove the claim.

Proof. Let M be a maximum weight matching of size k and $\widehat{M}$ be a maximum weight matching of size $k+1$. Observe that $M \cup \widehat{M}$ must have balanced cycles and balanced even length paths. (Here balanced means that $\widehat{w}$ on the cycle/path is 0). It they weren't balanced, then we could improve either M or $\widehat{M}$ by inverting the matching for the cycle/path. On the other hand, we have odd length alternating cycles. If the are l such cycles with start and end with an edge in M, there must be $l+1$ which start and end with an edge in $\widehat{M}$, since $\widehat{M}$ has one more edge than M. Now, all odd length path must have that same weight w.r.t. $\widehat{w}$. Again this follows from the fact that $M, \widehat{M}$ have maximum weight. The algorithm chooses one of the $l+1$ augmenting paths starting/end with an edge in $\widehat{M}$. Note: it can also be shown that the graph G' has no negative cycles, so Dijkstra's can be used to find shortest paths in G'. ∎

2.6 Max Flow

A *flow network* $G(V, E)$ is a directed graph together with a *capacity* function $c : E \to \mathbb{R}_{\geq 0}$. We distinguish two vertices $s, t \in V$, called the *source* and *sink*, respectively. A *flow* in G is a function $f : V \times V \to \mathbb{R}$, such that for all $u, v \in V$ we have $0 \leq f(u, v) \leq c(u, v)$ and for and $u \in V$

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v).$$

In other words, the flow from u to v does not exceed the capacity from u to v, and the in-flow at any vertex equals the out-flow at that vertex.

Problem. *Given a flow network with designated source s and sink t, find a flow with maximum value. We say the value of a flow is*

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s).$$

Theorem 6. *If the capacity function c is integral, then there is a polynomial time algorithm to compute a maximum flow f, which is also integral.*

2.6.1 Using Max Flow to Generate Maximal Matchings

Given an undirected bipartite graph $G(V, E)$, we can construct a corresponding flow network $G'(V', E')$ for G such that a maximum flow in G' corresponds to a maximum matching in G. Let the source s and the sink t be vertices not in V. Let $V' = V \cup \{s, t\}$. If the vertex partition of $V = L \cup R$, then

$$E' = \{(s, u) : u \in L\} \cup \{(u, v) : (u, v) \in E\} \cup \{(v, t) : v \in R\}.$$

Finally, we assign each edge a unit capacity.

2.7 Minimum-Weight Spanning Trees

A *tree* on n vertices is either an acyclic, connected graph, *OR*, an acyclic graph with $m \geq n - 1$, *OR* a connected graph with $m \leq n - 1$.

Theorem 7. *Suppose G is connected and $v_1, \ldots, v_k$ is a cycle in G. Then $G - \{v_i, v_{i+1}\}$ is connected for any $1 \leq i < k$.*

Proof. If we remove $\{v_i, v_{i+1}\}$ from a u-v-path, we can form another u-v-path, by taking the "long" way around the cycle. ∎

Theorem 8. *Suppose $T = (V, E)$ is a tree. Let $e = \{v_i, v_j\}$ be an edge not in E. Then $T + e$ contains a cycle. Removing any edge from that cycle gives us a tree.*

A proper subset $\varnothing \neq S \subset V$ is called a *cut*. Edges that have one endpoint in S and the other in $V \backslash S$ *cross* the cut S.

Theorem 9. *Given an undirected graph $G(V, E)$ with edge weights $w : E \to \mathbb{R}$, injective, the minimum weight edge crossing S is contained in some MST.*

Proof. Suppose not. Let e be a min-weight edge crossing S not contained in some MST T. Then $T + e$ contains a cycle. Let f be an edge crossing the cut in T. Then $w(f) > w(e)$, so $T + e - f$ is a spanning tree with less weight, contradiction. ∎

2.7.1 Prim's Algorithm

Assume vertices are labelled $1, \ldots n$ and all edge weights are distinct. Assume that popping the priority queue returns the element with minimal key value.

```
S ← {1}
until S = V do
    let {p, q} be the min-weight edge crossing S
    add {p, q} to T
    S ← S ∪ {q}
repeat
```

The minimum weight edge crossing S is determined by maintaining an array, $D[1...n]$, where $D[i] = \min_{p \in S} w(\{p, i\})$. We can efficiently find the smallest value of D, by storing the vertices in a min-priority queue, with key $D[\cdot]$. Suppose we pop vertex q from the priority queue, if u is a neighbor of q such that $w(\{q, u\}) < D[u]$, then call `DecreaseKey`$(u, w(\{q, u\}))$.

With this, the structure of Prim's algorithm is nearly identical to that of Dijkstra. Hence Prim's algorithm runs in $O(m + n \log n)$. Note: We can relax the assumption about distinct edge weights by "breaking ties" via a lexicographic sorting of the edges in E (index edges $e_1, \ldots, e_m$, where we have $e_i \prec e_j$ if $w(e_i) < w(e_j)$ or $w(e_i) = w(e_j)$ and $i < j$.)

2.7.2 Kruskal's

Maintain connected components of MST in union-find.

```
T = ∅ #O(sort(E) + Eα(V) + V)
for v ∈ V: make-set(v)
sort E be weight
for e = {u, v} ∈ E in increasing order:
    if find(u) ≠ find(v):
        T = T ∪ {e}
        Union(u,v)
return T
```

2.8 Sample Problems

Problem. *We are given a digraph $G(V, E)$ with non-negative edge weights $l : E \to \mathbb{R}$, shelter capacity $S : V \to \mathbb{Z}_{\geq 0}$, and the number of workers at a site $w : V \to \mathbb{Z}_{\geq 0}$. Let W be the total number of workers. We want to assign people to shelter so that the shelter capacity is not overloaded but we guarantee everyone gets to a shelter in time at most Q. Minimize Q.*

First compute the shortest path between every pair of vertices, let D be the set of distances achieved. Sort D.

Then create an undirected bipartite graph G with workers on the left and shelters on the right. Set the weight of edge $\{u, v\}$, to be the distance from work vertex u to shelter vertex v. Perform binary search on D, starting with Q as the minimum value of D. Keep only the edges with weight at most Q and check if there exists a matching of size at least W. If so increase Q, else decrease Q, until we converge in D.

Problem. *We are given a $k \times k$ chessboard with some squares covered. We want to place k roots on the board so they don't attack each other.*

Create an undirected bipartite graph with k left and k right vertices (corresponding to row/columns respectively). Put edge $\{i, j\}$ if square (i, j) is available. Then find the maximum matching.

Problem. *We live in a city with one-way streets, corresponding to a directed graph $G(V, E)$, with lengths $l : E \to \mathbb{R}_{\geq 0}$. We have a home at vertex A and work at vertex B. There are k proposals for new one-way roads $\{((s_i, t_i, l_i) : 1 \leq i \leq k\}$. We want to vote for the proposal which will minimize the drive from A to B.*

Create G' a copy of G. Add edge from $A \to A'$, and for each proposal i, add edge (s_i, t'_i) with weight l_i. Use Dijkstra's to compute the shortest path from A to B'. Whichever proposal was used in the minimum length path, is what we vote for (or vote for nothing if we took (A, A')).

Problem. *Same as before but we want to vote for two roads to be built.*

Create two copies G', G'' of G. Add edges (s_i, t_i') and (s_i', t_i'') with weight l_i for each i. Then Dijkstra's.

3 Min-Flow Max-Cut

A *flow network* on a directed graph $G(V, E)$ is given by two distinguished vertices $s \in V$, called the *source*, and $t \in V$, called the *sink*, together with an edge capacity function $c : V \times V \to \mathbb{R}_{\geq 0}$. Assume that if $(u, v) \notin E$, then $c(u, v) = 0$. We also assume G has no self-loops and no cycles of length 2.

Given a flow network on a graph G, a *flow* is a function $f : V \times V \to \mathbb{R}$ satisfying

- (Capacity). $f(u, v) \leq c(u, v)$ for all $u, v \in V$;
- (Conservation). $f(u, V) = \sum_{v \in V} f(u, v) = 0$ for all $u \in V \backslash \{s, t\}$;
- (Skew-symmetry). $f(u, v) = -f(v, u)$ for all $u, v \in V$.

The *value* of a flow f, denoted $|f|$ is

$$|f| := f(s, V) := \sum_{v \in V} f(s, v).$$

Consider the following example,

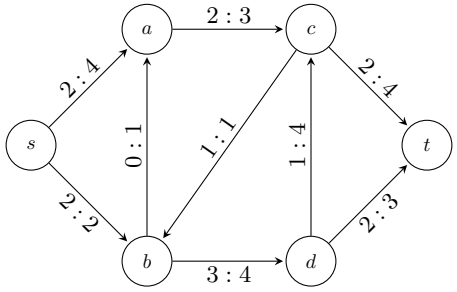


Figure 1: The number before the colon denotes the flow through an edge. The number after denotes the edges capacity.

For example, $c(s, a) = 4$, $c(a, b) = 0$, $c(d, t) = 3$, and $c(s, t) = 0$, whereas $f(s, a) = 2$, $f(c, b) = 1$, $f(b, c) = -1$, and $f(s, t) = 0$. Note $|f| = 4$.

We will use implicit summation notation to simply the proofs to follow. For example, if $X, Y \subseteq V$, then $c(X, Y) = \sum_{x \in X} \sum_{y \in Y} c(x, y)$. Now, let $X, Y, Z \subseteq V$, then the following properties hold.

a). $f(X, X) = 0$.

b). $f(X, Y) = -f(Y, X)$.

c). $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ if $X \cap Y = \varnothing$.

Theorem 10. $|f| = f(V, t)$.

Proof.

$$\begin{aligned}|f| &:= f(s,V) \overset{(c)}{=} f(V,V) - f(V\backslash\{s\},V) \overset{(a)}{=} -f(V\backslash\{s\},V)\\ &\overset{(b)}{=} f(V,V\backslash\{s\}) \overset{(c)}{=} f(V,t) + f(V,V\backslash\{s,t\}) \overset{(b)}{=} f(V,t) - f(V\backslash\{s,t\},V)\\ &= f(V,t), \text{ by conservation.}\end{aligned}$$

■

A *cut* (S,T) in a flow network on $G(V,E)$ is a partition of V into S and T such that $s \in S$ and $t \in T$. The capacity of a cut is $c(S,T)$. For instance, consider the cut $S = \{s,d\}$, $T = \{a,b,c,t\}$ in Figure 1. Then $c(S,T) = 13$ and $f(S,T) = (2+2+0+0) + (0+(-3)+1+2) = 4 = |f|$.

Theorem 11. *For any flow f on a graph $G(V,E)$ and any cut (S,T), we have $|f| = f(S,T)$.*

Proof.

$$f(S,T) = f(S,V) - f(S,S) = f(S,V) = f(s,V) + f(S\backslash\{s\},V) = f(s,V) = |f|.$$

■

Given a flow f on a flow network on $G(V,E)$ the residual capacity, $c_f : V \times V \to \mathbb{R}_{\geq 0}$ is

$$c_f(u,v) := c(u,v) - f(u,v).$$

Note if $(u,v) \notin E$, then $c(u,v) = 0$ and $c_f(u,v) = f(v,u)$. Given a flow network on $G(V,E)$ and a flow f, a *residual network* is a directed graph $G_f(V,E_f)$ where $(u,v) \in V \times V$ is in E_f if and only if $c_f(u,v) > 0$. The capacity of the edge $(u,v) \in E_f$ is $c_f(u,v)$. An *augmenting path* is a path from s to t in G_f. The *residual capacity* of a path P in G_f is

$$c_f(P) := \min_{(u,v)\in P} c_f(u,v).$$

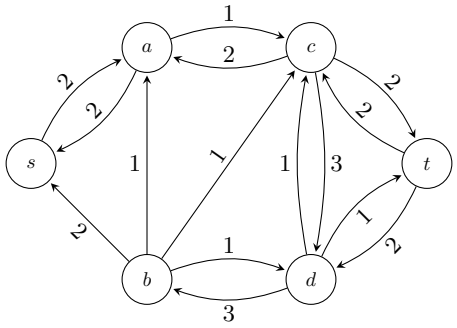


Figure 2: Residual network of the flow network in Figure 1. Edges are labeled only with their flow. The residual capacity of the augmenting path s,a,c,d,t is 1.

Theorem 12 (Min-Flow Max-Cut)**.** *The following are equivalent:*

1. *$|f| = c(S,T)$ for some cut (S,T);*
2. *f is a max flow;*
3. *f admits no augmenting paths.*

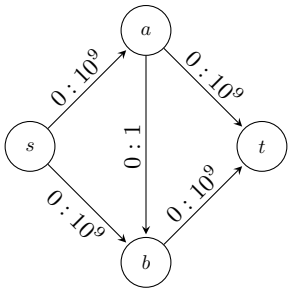


Figure 3: Downside of Floyd-Fulkerson. Depending on how the augmenting path is chosen, the algorithm could halt in as few as 2 iterations or as many as 2 billion.

Proof. $(1 \Rightarrow 2)$. As $|f| \leq c(S,T)$ for any cut (S,T), the claim is immediate from the assumption.

$(2 \Rightarrow 3)$. If f admits some augmenting path, then the flow value can be increased by the residual capacity, contradicting the maximality of the flow.

$(3 \Rightarrow 1)$. Suppose f admits no augmenting paths. Define $S = \{u \in V : u \text{ is reachable from } s \text{ in } G_f\}$ and $T = V \backslash S$. Then (S,T) is a cut. Let $u \in S$ and $v \in T$ such that $(u,v) \in E$. Then $c_f(u,v) = 0$, since there is no edge from u to v in G_f. Thus $f(u,v) = c(u,v)$. Whence $f(S,T) = c(S,T)$. ∎

Algorithm. (Floyd-Fulkerson).

```
for all u, v ∈ V
    f[u,v] ← 0
while an augmenting path P exists in G_f
    augment f by c_f(P)
```

3.0.1 Edmond-Karp

By using breadth-first search to find the shortest s-t path in G_f, Floyd-Fulkerson's algorithm runs in $O(VE^2)$ time. Define $\delta_f(v)$ to be the length of the shortest path from s to v in G_f.

Lemma 2. *Suppose f' is the flow obtained by augmenting f with some augmenting path in G_f. Then for all $v \in V$, $\delta_{f'}(v) \geq \delta_f(v)$.*

Proof. Suppose not, i.e. that there is some $v \in V$ such that $\delta_{f'}(v) < \delta_f(v)$. Let $\hat{v}$ be the vertex with the smallest value of $\delta_{f'}(v)$ satisfying $\delta_{f'}(v) < \delta_f(v)$. Let u be $\hat{v}$'s predecessor on the shortest path in G_f. Then

$$\delta_{f'}(v) = \delta_{f'}(u) + 1 \geq \delta_f(u) + 1.$$

The last step follows since $\delta_{f'}(u) < \delta_{f'}(\hat{v})$, so by choice of $\hat{v}$, $\delta_{f'}(u) \geq \delta_f(u)$.

- Case 1. $(u,\hat{v}) \in G_f$. Then $\delta_f(\hat{v}) \leq \delta_f(u) + 1$, since the shortest path from s to $\hat{v}$ in G_f is no longer than a path from s to $\hat{v}$, in which $\hat{v}$'s predecessor is u.
- Case 2. $(u,\hat{v}) \notin G_f$. Then the transition from f to f' must augment on a path containing (v,u), implying $(v,u) \in G_f$. Thus $\delta_f(u) + 1 = \delta_f(\hat{v}) + 2$.

In any case, we have $\delta_{f'}(\hat{v}) \geq \delta_f(\hat{v})$, a contradiction. ∎

Theorem 13. *The while loop in the Edmond-Karp algorithm iterates $O(VE)$ times.*

Proof. Let P be an augmenting path in G_f. We say an edge (u, v) is *critical* if $c_f(P) = c_f(u, v)$. We'll show that any edge (u, v) can be critical at most $V/2$ times.

Suppose $(u, v) \in E$, then since augmenting paths are shortest paths, when (u, v) is critical for the first time, $\delta_{f'}(u) = \delta_{f'}(v) + 1$. After augmenting (u, v) no longer appears in the residual network. It cannot reappear until (v, u) appears on some augmenting path. If f' is the flow in G when this first occurs, then

$$\delta_{f'}(u) = \delta_{f'}(v) + 1 \geq \delta_f(v) + 1 = \delta_f(u) + 2.$$

Therefore, u's distance from s increases by at least 2 each time (u, v) is critical. Hence (u, v) is critical at most $V/2$ times. ∎

3.1 Applications

3.1.1 Playoff Elimination

We are given a set of n teams. Team i has w_i wins, ℓ_i losses, r_i games remaining, and s_{ij} games remaining against team j, $j \neq i$. We say team i is *eliminated* if some team j is guaranteed to have greater than $w_i + r_i$ wins. Given $i_0 \in [n]$ we want an algorithm to determine whether team i_0 is eliminated.

For each $(i, j) \subset ([n]\backslash\{i_0\}) \times ([n]\backslash\{i_0\})$ with $i < j$ we add edge $(s, v_{i,j})$ with capacity s_{ij} to our flow network. For each $k \in [n]\backslash\{i_0\}$, we add edge (w_k, t) with capacity $w_{i_0} + r_{i_0} - w_k$ (the maximum number of wins team k can get and not necessarily have more wins than team i_0). Finally, add edges (v_{ij}, w_i) and (v_{ij}, w_j) with infinite capacity for all valid i, j pairs.

Now, team i_0 is eliminated if and only if the max flow does not saturated all edges leaving the source. If all edges are saturated, this corresponds to an assignment of wins of the remaining games, not involving team i_0, such that no team exceeds $w_{i_0} + r_{i_0}$ wins.

3.1.2 Vertex Covers

A *cover* of a graph $G(V, E)$ is a subset $S \subseteq V$ such that every edge $e \in E$ has at least one vertex in S. The problem of minimum cover is finding a minimum cardinality cover of G. A bipartite graph $G(V, E)$ has a min-cover of cardinality k if and only if it has a maximum cardinality matching of size k.

4 Linear Programming

Problem. *We are given n foods of cost c_i (\$/kg) and m types of nutrients, where b_j is the minimum intake of nutrient j in (kg/day). A_{ij} is the kgs. of nutrient j in a kg. of food i. We want to minimize the cost of our diet.*

Let x_i be the amount of food i consumed per day.

minimize $\sum_{i=1}^{n} c_i x_i$ subject to

$$x_i \geq 0 \quad i = 1, \ldots, n$$
$$\sum_{i=1}^{n} A_{ij} x_i \geq b_j \quad j = 1, \ldots, m.$$

Problem. *Transportation.*

- *n producers with productivity $p_i, i = 1, \ldots, n$.*
- *m consumers with demand $d_j, j = 1, \ldots, m$.*
- *Assume $\sum p_i \geq \sum d_j$.*
- *T_{ij} is the cost of transporting 1 unit from producer i to consumer j.*
- *We want to identify the optimal allocation. Let x_{ij} be the number of units producer i gives to consumer j.*

$$\begin{aligned}
\text{minimize} \quad & \sum_{i,j} T_{ij} x_{ij} \\
\text{subject to} \quad & x_i \geq 0, && i = 1, \ldots, n \\
& \sum_{i=1}^{n} x_{ij} = d_j, && j = 1, \ldots, m \\
& \sum_{j=1}^{n} x_{ij} \leq p_i, && i = 1, \ldots, n.
\end{aligned}$$

Problem. *Mixing Liquids.*

- *n chemicals, m mixtures of cost c_i (\$/liter).*
- *Mixture i is an $a_{i1} : \ldots : a_{in}$ ratio of the n chemicals. (The ratio is normalized so $\sum_{j=1}^{n} a_{ij} = 1$).*
- *Goal mixture: $v_1 : \ldots : v_n$ (normalized)*
- *Let x_i be the amount of mixture i consumed.*

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{n} x_i c_i \\
\text{subject to} \quad & x_i \geq 0, && i = 1, \ldots, m \\
& \sum_{i=1}^{m} a_{ij} x_i = v_j, && j = 1, \ldots, m.
\end{aligned}$$

Problem. *Max Flow. Digraph $G(V, E)$, $s, t \in V$, $c : E \to \mathbb{R}_{\geq 0}$. Let x_{uv} be the flow through edge $(u, v) \in E$.*

$$\begin{aligned}
\text{maximize} \quad & \sum_{v:(s,v)\in E} x_{sv} - \sum_{v:(v,s)\in E} x_{vs} \\
\text{subject to} \quad & 0 \leq x_{uv} \leq c_{uv}, && (u, v) \in E \\
& \sum_{v:(w,v)\in E} x_{vw} = \sum_{v:(v,w)\in E} x_{wv}, && w \in V \backslash \{s, t\}.
\end{aligned}$$

4.1 Duality

Given a linear program (LHS), there is a corresponding dual (RHS).

$$\begin{array}{rll} \max c^\mathsf{T} x & & \min y^\mathsf{T} b \\ x & \geq 0 & y^\mathsf{T} A \quad \geq c \\ Ax & \leq b & y \quad \geq 0 \end{array}$$

Theorem 14. *If $Ax \leq b$ where $x \geq 0$ and $y^\mathsf{T} A \geq c$ where $y \geq 0$, then $c^\mathsf{T} \leq y^\mathsf{T} b$. So in particular, the value of the dual program is the same as the original (should there exist a feasible solution to either).*

Proof.

$$c^\mathsf{T} x \leq (y^\mathsf{T} A - c^\mathsf{T})x + c^\mathsf{T} x = y^\mathsf{T} Ax = y^\mathsf{T}(Ax - b) + y^\mathsf{T} b \leq y^\mathsf{T} b.$$

■

4.2 Form Matching - Duality Conversion

To eliminate equalities, replace by two inequalities, one in each direction. To ensure all variable are non-negative, e.g. if x_1 is allow to obtain any value in $\mathbb{R}$, rewrite $x_1 = y_1 - z_1$ where $y_1, z_1 \geq 0$. To eliminate inequalities, e.g. $2x_1 + 11x_2 \leq 3$, use a slack variable, $w_1 \geq 0$ and $2x_1 + 11x_2 + w_1 = 3$.

Unrestricted variables in the primal become equality constrains in the dual. Non-negative variables become inequality constraints. LP with a maximization objective, always have $\leq$ constraints. Minimization LPs have $\geq$ constraints.

4.3 Zero-Sum Games & Other LP Problems

Given $A \in \mathbb{R}^{m \times n}$. The row players strategy, specified by $p_1, \ldots, p_m$ (the probability of picking a row in the matrix) is given by $\max z$ subject to $p_i \geq 0$, $p_1 + \ldots + p_m = 1$, and

$$\begin{cases} a_{11}p_1 + \ldots + a_{m1}p_m \geq z \\ \vdots \\ a_{1n}p_1 + \ldots + a_{mn}p_m \geq z \end{cases}.$$

Note the equations are the expected payoff of choosing column j. Correspondingly, the column player's strategy, column probabilities $q_1, \ldots, q_n$, is given by $\min w$ subject to $q_j \geq 0$, $q_1 + \ldots + q_n = 1$, and

$$\begin{cases} a_{11}q_1 + \ldots + a_{1n}q_n \leq w \\ \vdots \\ a_{m1}q_1 + \ldots + a_{mn}q_n \leq w \end{cases}.$$

These programs are duals of each other (hence two-person zero-sum games always have a value).

Given a convex polygon find the largest, axis-parallel square that can fit inside the polygon. *Note: If we fix an orientation of the polygon, say clockwise, then we can specify a side by a triple (a, b, c), and (x, y) lies to the right of the side if $ax + by + c \geq 0$ and to the left if it is ≤ 0.* So we use variables x_1, y_1 to denote the upper-left corner of the square and x_2, y_2 to denote the lower-right

corner. We want to $\max x_2 - x_1$ subject to $x_2 - x_1 = y_1 - y_2$ and we need to check that (x_i, y_j) $i, j \in \{1, 2\}$ lie in the polygon. So add the constraints

$$\begin{cases} a_1 x_i + b_1 y_j + c_1 \geq 0 \\ \vdots \\ a_n x_i + b_n y_j + c_n \geq 0 \end{cases}$$

for each corner of the square.

We want to find the largest circle which can fit inside a convex polygon. *Note: If we normalize the* (a, b, c) *so that* $a+b+c = 1$*, then* $|ax+by+c|$ *gives the distance from* (x, y) *to the edge corresponding to* (a, b, c). So $\max r$ subject to

$$\begin{cases} a_1 x_1 + b_1 y_1 + c_1 \geq r \\ \vdots \\ a_n x_1 + b_n y_1 + c_n \geq r \end{cases}.$$

Here r is the radius of the circle and (x_1, y_1) is the center.

Problem. *We're given an undirected graph* $G(V, E)$*. For each edge* $e \in E$ *we are given two positive integers* a_e *and* b_e*. We want to determine whether it is possible to assign a number* $\varphi(v) \in \{1, \ldots, 10\}$ *to each vertex in* V *such that for every edge* $e = \{u, v\} \in E$ *we have* $\varphi(u) + \varphi(v)$ *equal to* a_e *or* b_e*. Assume* $|V| = n$*. Find a polynomial-time algorithm for this problem.*

Label the vertices $v_1, \ldots, v_n$. For each vertex v_i in V add variable x_i to the ILP. For each edge $e = \{u, v\} \in E$ add two variables $z_{e,a}$ and $z_{e,b}$ to the ILP.

$$1 \leq x_i \leq 10 \qquad \text{for } i \in [n] \tag{4}$$

$$0 \leq z_{e,a}, z_{e,b} \leq 1 \qquad \text{for } e \in E \tag{5}$$

$$z_{e,a} + z_{e,b} = 1 \qquad \text{for } e \in E \tag{6}$$

$$x_i + x_j = a_e z_{e,a} + b_e z_{e,b} \qquad \text{for } e = \{v_1, v_2\} \in E \tag{7}$$

5 $\mathcal{P}$ vs $\mathcal{NP}$

Problem. ***SAT.***

Input: A CNF ("AND of ORs") formula φ*.*

Output: A satisfying assignment for φ*.*

Problem. ***Integer Linear Programming (ILP).***

Input: A collection of linear inequalities.

Output: An assignment of integers to the variables that satisfies the inequalities.

Problem. ***Independent-Set.***

Input: A graph $G(V, E)$*; integer* $K \geq 0$*.*

Output: $S \subset V$*, with* $|S| = K$ *and for all* $\{u, v\} \in E$*, at most one of* u, v *are in* S*.*

Problem. ***3-Coloring.***

Input: A graph $G(V, E)$*.*

Output: $\phi : V \to \{$*red, green, blue*$\}$ *such that for all* $\{u, v\} \in E$*,* $\phi(u) \neq \phi(v)$*.*

Claim 2. *Either all of these problems have polynomial-time algorithms, or none of them do.*

Assuming we have an oracle for problem A. If we can use the oracle to solve problem B in polynomial-time, then we say B *reduces* to A, which we denote $B \leq^{\mathrm{p}} A$.

Theorem 15. *SAT $\leq^p$ ILP.*

For each variable x_i in φ, y_i is a variable in the ILP. We require $0 \leq y_i \leq 1$. To translate a clause, e.g. $x_1 \vee \overline{x_2} \vee x_3$, we substitute y_i for x_i and $(1 - y_i)$ for $\overline{x_i}$, and require the resulting expression to be at least 1, e.g. $y_1 + (1 - y_2) + y_3 \geq 1$. If the oracle sets $y_i = 1$, in the resulting solution, then x_i is true in the satisfying assignment, if y_i is 0, then x_i is false. If no feasible solution to the ILP exists, φ is not satisfiable.

Theorem 16. *I-S $\leq^p$ ILP.*

For every $v \in V$ add x_v as a variable ($0 \leq x_v \leq 1$). For every edge $\{u, v\} \in E$, add $x_u + x_v \leq 1$. Require that $x_{v_1} + \ldots + x_{v_n} = K$. If $x_v = 1$, then v is added to S, otherwise $v \notin S$. No feasible solution, implies no I-S of size K exists.

Theorem 17. *3-coloring $\leq^p$ ILP.*

For each vertex v and color c, add $0 \leq x_{v,c} \leq 1$ to ILP. Require

$$x_{v,green} + x_{v,red} + x_{v,blue} = 1.$$

If u, v are adjacent vertices, require

$$x_{u,red} + x_{v,red} \leq 1; \quad x_{u,blue} + x_{v,blue} \leq 1; \quad x_{u,green} + x_{v,green} \leq 1$$

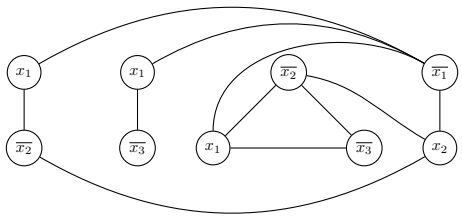
Theorem 18. *3-coloring $\leq^p$ SAT.*

For each vertex v and color c, add $x_{v,c}$ as a literal to SAT. Add clause $x_{v,r} \vee x_{v,b} \vee x_{v,g}$ to formula. Add clause $\overline{x_{v,c}} \vee \overline{x_{v,c'}}$ for each vertex v and distinct pair of colors (c, c'). For each edge $\{u, v\} \in E$ add

$$(\overline{x_{u,r}} \vee \overline{x_{v,r}}) \wedge (\overline{x_{u,b}} \vee \overline{x_{v,b}}) \wedge (\overline{x_{u,g}} \vee \overline{x_{v,g}}).$$

Theorem 19. *SAT $\leq^p$ I-S.*

For each literal in φ add a vertex to the graph. Connect two vertices if their corresponding literals are in the same clause. Connect every occurrence of a literal and its negation. The literals picked in the I-S, are set to true. If some x_i and its negation aren't picked, arbitrarily assign it true. e.g. $(x_1 \vee \overline{x_2}) \wedge (x_1 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3} \vee x_1) \wedge (\overline{x_1} \vee x_2)$, has corresponding graph



Theorem 20. *SAT $\leq^p$ 3-Coloring.*

Connect all literals to R vertex. Connect x_i to $\overline{x_i}$. Let B be true and G be false. (See notes.)

6 Convolution and FFT

Problem. *We are given a string $T = t_0 t_1 \dots t_n$ and a pattern $P = p_0 \dots p_m$ with $t_i, p_j \in \{a, b\}$. We want to output all $k \in \{0, \dots, n-m\}$ such that the pattern P occurs in T at position k, i.e. $t_{k+i} = p_i$ for all $i \in \{0, \dots, m\}$.*

First we encode our string in binary by mapping $a \mapsto 0$ and $b \mapsto 1$.

We'd like to compute $a_k = \sum_{k=0}^{m} (t_{k+i} - p_i)^2$ for $0 \le k \le n-m$, since $a_k = 0$ if and only if P occurs in T at k. (In fact, a_k gives the number of mismatches between the pattern P and T starting at k.) Now

$$a_k = \sum_{i=0}^{m} t_{k+1}^2 + \sum_{i=0}^{m} p_i^2 - 2\sum_{i=0}^{m} t_{k+i} p_i = B_k + A + C_k.$$

(In fact, since everything is zeros and ones we will drop the squares in the first two sums.) Clearly, we can compute A in $O(n)$ time. Furthermore, observing that $B_{k+1} = B_k - t_k + t_{m+k+1}$, once we compute B_0, we may compute $B_1, \dots, B_{n-m}$ in $O(n)$ time. Finally, to compute C_k we first set $r_i = p_{m-i}$ for $0 \le i \le m$. Then

$$C_k = \sum_{i=0}^{m} t_{k+i} r_{m-i} = (r * t)_{m+k},$$

where $r * t$ is the convolution of r and t (computed in $O(n \log n)$). Lastly, we compute the a_k in $O(n)$ and return the k for which $a_k = 0$.

Problem. *Now consider the previous problem, but allow wildcards in the pattern, i.e. $p_j \in \{0, 1, ?\}$*

Then we map $0 \mapsto 1$, $1 \mapsto -1$ and $? \mapsto 0$ and we put our counter $a_k = \sum_{i=0}^{m} p_i(t_{k+i} - p_i)$. In which case, a_k is -2 times the number of disagreements in the string. (We want all mismatches to count the same.)

t_i \ p_i	0	1	-1
1	0	0	-2
-1	0	-2	0

Problem. *Now we allow wildcards in our initial string.*

Then we map $0 \mapsto 1$, $1 \mapsto -1$ and $? \mapsto 0$ and we put our counter $a_k = \sum_{i=0}^{m} p_i t_{k+i}(t_{k+i} - p_i)^2$. In which case, a_k is -4 times the number of disagreements in the string.

t_i \ p_i	0	1	-1
0	0	0	0
1	0	0	-4
-1	0	-4	0

We can still efficiently compute the $a_k = \sum p_i t_{k+i}^3 + \sum p_i^3 t_{k+i} - 2\sum p_i^2 t_{k+i}^2$, since we can ignore the cubes, meaning $A_k = B_k = (r * t)_{m+k}$ and $(r^2 \star t^2)_{m+k} = C_k$, and $a_k = 2A_k - 2C_k$.

6.1 Fast Convolution using FFT

6.1.1 Interpolation

Let $r_1, \ldots, r_{n+1}$ be $n+1$ distinct real numbers. Let $p(x) = a_0 + \ldots + a_n x^n$ be the unique degree n polynomial that contains $(r_1, v_1), \ldots (r_{n+1}, v_{n+1})$. For $i \in \{1, \ldots, n+1\}$ define

$$p_i(x) = \frac{\prod_{k \neq i}(x - r_k)}{\prod_{k \neq i}(r_i - r_k)}.$$

Then

$$p(x) = \sum_{j=1}^{n+1} v_j p_j(x).$$

Alternatively, define

$$V(r_1, \ldots, r_n) = \begin{pmatrix} 1 & r_1 & r_1^2 & \cdots & r_1^n \\ 1 & r_2 & r_2^2 & \cdots & r_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & r_{n+1} & r_{n+1}^2 & \cdots & r_{n+1}^n \end{pmatrix}. \tag{8}$$

Then

$$V(r_1, \ldots, r_n) \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} p(r_1) \\ p(r_2) \\ \vdots \\ p(r_n). \end{pmatrix} \tag{9}$$

Hence $V^{-1}(r_1, \ldots, r_n)\mathbf{v} = \mathbf{a}$. These are called the *Vandermonde matrices*.

6.1.2 Efficient Interpolation

Let $N = 2^l$ and consider the set $R_N = \{\exp(k \cdot \frac{2\pi i}{N}) : 0 \leq k \leq N-1\}$. Observe that $R_{\frac{N}{2}} = \{x^2 : x \in R_N\}$. Let $p(x) = a_0 + \ldots a_{N-1}x^{N-1}$. We would like to efficiently evaluate a $p(x)$ at every point in R_N.

Define

$$\begin{aligned} p_e(x) &= a_0 + a_2 x + a_4 x^2 + \ldots + a_{N-2} x^{\frac{N}{2}-1} \\ p_o(x) &= a_1 + a_3 x + a_5 x^2 + \ldots + a_{N-1} x^{\frac{N}{2}-1}. \end{aligned} \tag{10}$$

Observe that $p(x) = p_e(x^2) + x p_o(x^2)$. Then we can recursively evaluate p_o and p_e on $R_{\frac{N}{2}}$ before combining to compute $p(x)$ in $O(n)$ time. Thus we can compute $p(x)$ at all points in R_N in $O(n \log n)$ time.

Now if we're given distinct $(r_1, v_1), \ldots, (r_N, v_N)$, and we want to find $p(x) = a_0 + \ldots + a_{N-1}x^{N-1}$ that goes through each point. It is easy to verify that

$$V^{-1}(\omega_0, \omega_1, \ldots, \omega_{N-1}) = \frac{1}{N} V(\omega_0^{-1}, \ldots, \omega_{N-1}^{-1}). \tag{11}$$

Let $q(x) = v_1 + \ldots + v_N x^{N-1}$. Then $a_j = \frac{1}{N} q(\omega_j^{-1})$. Hence $\mathbf{a}$ can be computed in $O(n \log n)$.

6.1.3 Polynomial Multiplication

Let $p(x) = a_0 + \ldots + a_n x^n$ and $q(x) = b_0 + \ldots + b_m x^m$ and $r(x) = p(x)q(x)$. To recover $r(x)$ we need $N \geq m + n + 1$ and we can always find $N < 2(m + n + 1)$. Then we simply evaluate $p(x)$ and $q(x)$ on R_N and then interpolate $r(x)$ using $r(\omega_i) = p(\omega_i)q(\omega_i)$, $i \in \{0, \ldots, N-1\}$.

7 Priority Queues

7.1 Heaps

- *Min-Heap Property (MHP).* The key of the parent is greater than or equal to that of its two children.

7.1.1 Binary Heap

- Complete binary tree with the min-heap property.
- `insert`(x): inserted as leftmost available leaf, the bubble up to maintain MHP.
- `deleteMin`: deletes and returns root of tree. Moves most recently added leaf to root of tree and bubbles down to maintain MHP.
- `decreaseKey`(x, k): assuming $k < x$`.key`, update x`.key` to k and bubble up.

7.1.2 Binomial Heap

- Forest of binomial trees, each with the MHP.
- For any tree in the heap, its *rank* is the degree of its root. A tree of rank k will have 2^k nodes. For each k there is at most 1 tree of rank k.
- A root of rank k has k children of rank $0, 1, \ldots, k-1$, respectively.
- Roots of the trees in the binomial heap are stored in a doubly linked list.
- `insert`(x): Add singleton tree for x. Recursively merge tree of same rank. To merge two tree we need only compare their roots. For example, after inserting 12, the tree in figure 1 has the form of figure 2, followed by figure 3, and finally figure 4.
- `deleteMin`: Traverse linked list of roots to find minimum root. Delete minimum root, its children are merged into the forest. See figure 5.
- `decreaseKey`(x, k): Sets x's key to k and bubbles up within x's tree.

7.1.3 Fibonacci Heaps

Ideal for cases where the number of calls to `deleteMin` is relatively small compared to the number of calls to `insert` or `decreaseKey`. For example, Dijkstra's or Prim's algorithm.

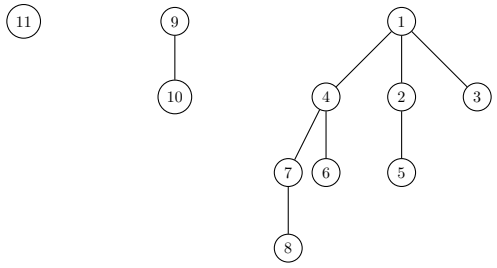


Figure 4: Node with key 1 has rank 3. Node 9 has rank 1. Node 11 has rank 0.

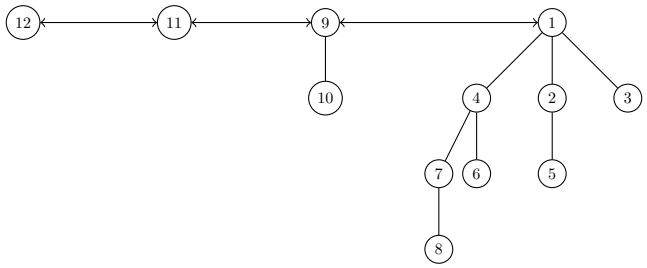


Figure 5: Inserting 12.

- Every node x has a pointer x`.p` to its parent and a pointer x`.c` to one of its children. The children of x are linked together via a circular, doubly linked list. For instance, each child y, has pointers y`.left` and y`.right`, pointing to y's left and right children, respectively. If y is an only child, then y`.left` $= y$`.right` $= y$.
- Every node x has contains an x`.degree` attribute storing the number of children x has and a x`.mark` attribute indicating whether x has lost a child since the last time it was made a child of another node. New nodes are unmarked and x becomes unmarked whenever it is made the child of another node.
- A Fibonacci heap H consists of a pointer H`.min` to the root of a tree containing the minimum key (or any such tree if multiple roots contain the same minimum key). When H is empty, H`.min` is `null`. The roots of the trees in H are linked together via a circular, doubly linked list. H also contains one other attribute, namely H`.n` the number of nodes currently in H.
- Define $D(n)$ to be the maximum degree of a node in a Fibonacci heap on n nodes.

- `insert(`x`)`: Add unmarked node x to root list of H, updating H`.n` and if necessary, H`.min`. Note x's child and parent are initialized to `null`.
- `deleteMin`: Let $\alpha = H$`.min`. For each child x of α, add x to the root list, set x's parent to `null`. Delete α from root list. If deleting α makes H empty, set H`.min` to `null`. Otherwise, set H`.min` to α`.right` and call `consolidate`. Update H`.n` and return α.

```
consolidate:
    A = new Array(D(H.n)) #initialized to null
    for each w in the root list of H
        x = w
        d = w.degree
        while A[d] ≠ null
            y = A[d]
            if x.key > y.key
```

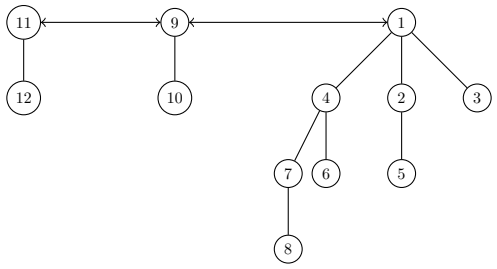


Figure 6: Merging 11 and 12.

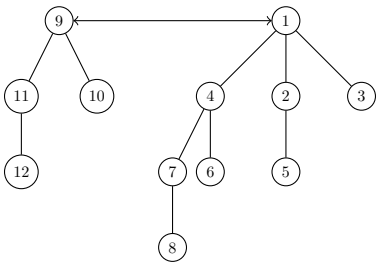


Figure 7: Merging 9 and 11.

```
                switch x and y
            fibLink(x, y)
            A[d] = null
            d = d + 1
        A[d] = x
    H.min = null
    for i = 0 to D(H.n)
        if A[i] ≠ null
            if H.min = null
                create a root list of H containing only A[i]
                H.min = A[i]
            else
                insert A[i] into H's root list
                if A[i].key < H.min.key
                    H.min = A[i]

fibLink(x, y):
    remove y from root list of H
    make y a child of x, increment x.degree
    y.mark = false
```

```
decreaseKey(x, k):
    x.key = k
    y = x.p
    if y ≠ null and x.key < y.key
        cut(x, y)
        cascascadeCut(y)
    if x.key < H.min
        H.min = x

cut(x, y):
    remove x from y's child list, decrement y.degree
```

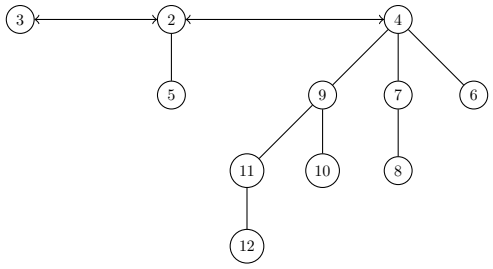


Figure 8: Call to `deleteMin` returns 1.

```
    add x to H's root list
    x.p = null
    x.mark = false

cascadeCut(y):
    z = y.p
    if z ≠ null
        if y.mark is false
            y.mark = true
        else
            cut(y, z)
            cascadeCut(z)
```

	Binary (worse case)	Binomial (worse case)	Fibonacci (amortized)
`insert`	$O(\lg n)$	$O(1)^*$	$O(1)$
`decreaseKey`	$O(\lg n)$	$O(\lg n)$	$O(1)$
`deleteMin`	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$

Table 1: *amortized

Lemma 3. *Let x be any node in a Fibonacci heap on n nodes. Suppose $k = x.\texttt{degree}$ and $y_1, \ldots, y_k$ are the children of x in the order in which they were linked to x from earliest to latest. Then $y_1.\texttt{degree} \geq 0$ and for $i \geq 2$, we have $y_i.\texttt{degree} \geq i - 2$.*

Proof. That $y_1.\texttt{degree} \geq 0$ is clear. For $i \geq 2$, when y_i is linked to x so must have $y_1, \ldots, y_{i-1}$. Hence $x.\texttt{degree} \geq i-1$, when y_i was linked to x. However, for y_i to be linked to x (in `consolidate`), it must have been that $x.\texttt{degree} = y_i.\texttt{degree}$, and hence $y_i.\texttt{degree} \geq i - 1$ when it was linked to x. Since y_i is still a child of x, it must be that y_i has lost at most one child since if it had lost 2, then y_i would've been cut from x by `cascadeCut`. Hence $y_i.\texttt{degree} \geq i - 2$. ∎

Let F_k denote the k-th Fibonacci number. Assume $F_0 = F_1 = 1$. Recall that $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$ for all integers $k \geq 0$ and moreover, $F_{k+2} \geq \phi^k$, where $\phi = \frac{1+\sqrt{5}}{2}$.

Theorem 21. *In a Fibonacci heap, a node x with degree k has size at least F_{k+2}.*

Proof. Let s_k denote the minimum possible size of any node of degree k in any Fibonacci heap. Clearly, $s_0 = 1$ and $s_1 = 2$ and $size(x) \geq s_k$. Consider some node z with $z.\texttt{degree} = k$ and $size(z) = s_k$. Suppose $y_1, \ldots, y_k$ are the children of z in the order they were added, earliest to

latest. Then we find the following lower bound on s_k

$$\begin{aligned} size(x) &\geq s_k \\ &\geq 2 + \sum_{i=1}^{k} s_{y_i\texttt{.degree}} \\ &\geq 2 + \sum_{i=1}^{k} s_{i-2} \end{aligned} \tag{12}$$

since we count one for z and one for y_1 ($size(y_1) \geq 1$). The claim follows by showing $s_k \geq F_{k+2}$ for $k \geq 0$. ∎

Theorem 22. $D(n) = O(\lg n)$.

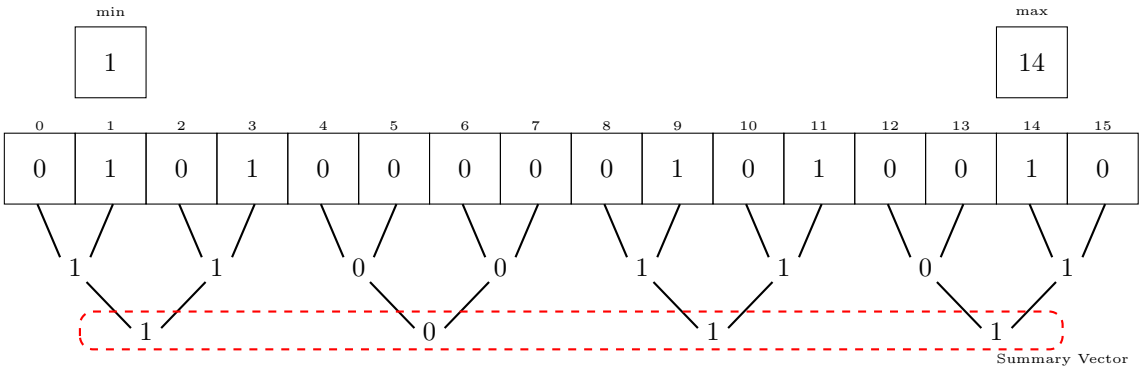
Proof. Let x be any node in a Fibonacci heap on n vertices. Let $k = x\texttt{.degree}$. Then $n \geq size(x) \geq F_{k+2} \geq \phi^k$. The claim follows by taking logarithms. ∎

8 Binary Search Trees

8.1 van Embde Boa Trees

A van Embde Boa tree (vEB-tree) is a data structure designed to implement sets where the maximum set size is known beforehand.

A size $u = 2^{2k}$ vEB-tree consists of a table `V.cluster`, where `V.cluster[`i`]` is a size $\sqrt{u}$ vEB-tree for $0 \leq i < \sqrt{u}$, together with a size $\sqrt{u}$ vEB-tree, called `V.summary`. Each vEB-tree also keeps track is its minimum and maximum element. The operations `insert`, `delete`, and `successor` all run in $O(\lg \lg u)$ time. In the case where $u = n^{\lg^{O(1)} n}$, our operations run in $O(\lg \lg n)$ time, where n is the number of elements in the data structure.



Given x, we can write $x = i\sqrt{u} + j$ where $0 \leq j < \sqrt{u}$. Then `high(`x`)` $= i$ and `low(`x`)` $= j$. Note `high` and `low` return the upper half and lower half of the binary representation of x, respectively. Furthermore, `index(`i, j`)` $= i\sqrt{u} + j$. Storing the minimum and maximum elements allows us to avoid unnecessary searches. If x is greater than or equal to the maximum element of its cluster, then clearly x's successor is not in its cluster, so we search the summary vector.

```
successor(V, x):
    if x < V.min
```

```
        return V.min #never recursively insert the minimum
    i = high(x)
    if low(x) < V.cluster[i].max #successor of x lies in its high cluster
        j = successor(V.cluster[i],low(x)) #search x's high cluster
    else
        i = successor(V.summary, i) #search summary
        j = V.cluster[i].min
    return index(i, j)
```

```
insert(V, x):
    if V.min = null
        V.min = V.max = x
        return #no recursive update when inserting to min
    if x < V.min: swap x with V.min #now insert old min
    if x > V.max: V.max = x
    if V.cluster[high(x)].min = null:
        insert(V.summary, high(x)) #will run in O(1) time
    insert(V.cluster[high(x)],low(x)) #always expensive
```

```
delete(V, x):
    if x = V.min: #deleting the minimum
        i = V.summary.min
        if i = null: V.min = V.max = null #tree emptied?
        x = V.min = index(i, V.cluster[i].min) #find next smallest
    delete(V.cluster[high(x)], low(x)) #one of delete calls has to be cheap
    if V.cluster[high(x)].min = null: #when deletion empties a cluster
        delete(V.summary, high(x)) #always expensive
    if x = V.max:
        if V.summary.max = null: V.max = V.min #only two elements
        else:
            i = V.summary.max
            V.max = index(i, V.cluster[i].max) #update to next largest max
```

Observe that if `delete(V.summary, high(`x`))` is called, then the deletion in the previous line must have emptied x's high cluster and thus ran in constant time. If x's cluster wasn't emptied, then there is no reason to update `V.summary`. Also note that if the minimum is deleted and the tree is still not empty, then the second smallest element must be stored in `V.min` and recursively deleted from the tree, since the minimum element is *never* recursively stored. The runtime $O(\lg \lg u)$ is obtained because all operations satisfy the recurrence $T(u) = T(\sqrt{u}) + O(1)$.

8.2 B-trees

- $B \leq$ # of children per node $< 2B$
- $B - 1 \leq$ # of keys per node $< 2B - 1$
- The root is the only exception to the above two rules.
- All leaves have same depth.
- Keys within a node are in sorted order.

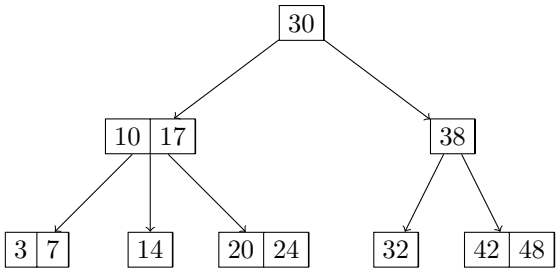


Figure 9: A 2-3 tree, i.e. a B-tree where $B = 2$.

Insertion works in a similar fashion to BST insert; however, if a node becomes overfull we must call `split`. The `split` operation removes the middle key of the new overly full node and moves it up to the parent, inserting the two halves of the split node accordingly. The operation may recurse if the parent is node overly full. The tree may end up with a new root.

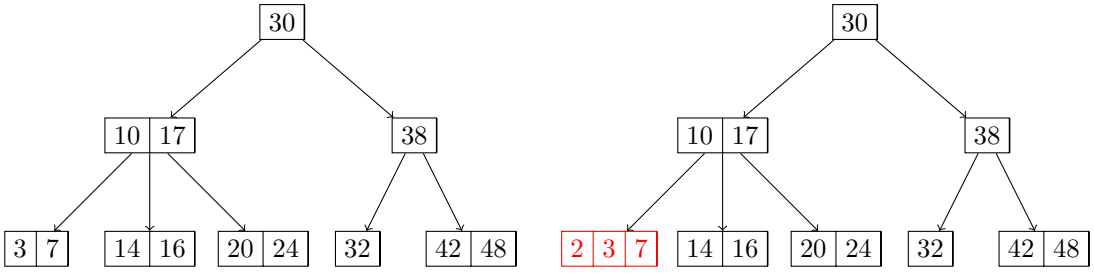


Figure 10: An insertion of 16, followed by 2. An overfull node is highlighted in red.

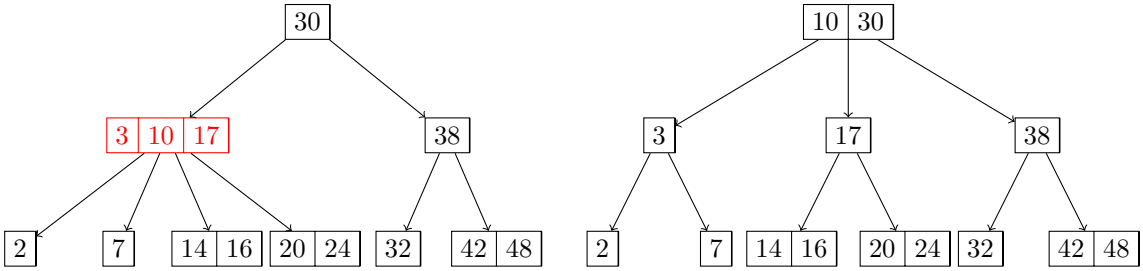


Figure 11: Two recursive `split` operations fix the B-tree properties.

Deletion works by first filling in the deleted node, either using the leftmost key in the right child, or the rightmost key in the left child, until the deletion is "pushed" down to a leaf. If a leaf has been made "underfull", then we either borrow from a sibling via a rotation, or merge two siblings together with a key from the parent in between. If some sibling has at least B keys, then we can borrow, otherwise a node and its sibling have $(B-2)+(B-1)$ keys in total, so we can safely merge to form a node with $2B-2$ keys.

8.3 Red Black Trees

Red-black trees are a type of self-balancing binary search tree with the following properties:

- Every node is red or black.

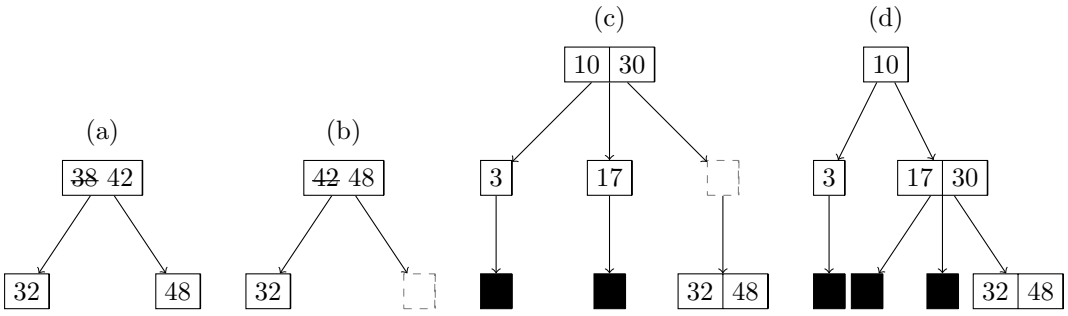


Figure 12: Delete 38 and then 42. (a). First we replace 38 with 42, to push the deletion down. Our 2-3 tree properties are still preserved at this point. (b). Once, we delete 42 and replace with 48, 48's right child is underfull. (c). To fix this, 32, 48, and 48's empty right child are merged. However, this leaves the 32-48 node's parent, underfull. (d). So we merge again with 17-30 and the empty parent. Underfull nodes are highlighted in gray. Only necessary parts of the tree are shown, some parts have been replaced with black boxes for compactness.

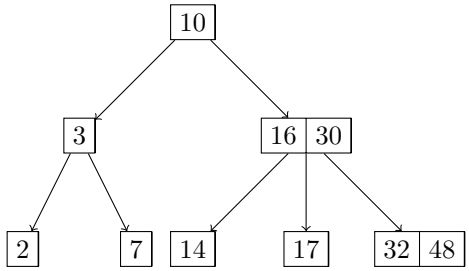


Figure 13: Delete 20 and 24. This leaves a child of 17-30 underfull. However, a sibling is able to donate 16 via a rotation.

- The root is always black.
- All paths from the root to a leaf always have the same number of black nodes.
- No consecutive nodes are red, i.e. if a node is red, both of its children must be black.
- Null nodes are black.

A red-black tree node contains the following attributes: `p` - the parent, `left`/`right` - the left and right child, respectively, `color` - either red or black, and `key`.

The *black height* of a node x in a red-black tree, denoted $bh(x)$, is the number of black nodes on a path from x to a leaf of the tree rooted at x. The *black height* of a red-black tree is the black height of its root.

Lemma 4. *A red-black tree rooted at a node x has at least $2^{bh(x)} - 1$ nodes.*

Proof. We proceed by induction on the height of the tree rooted at x. If $h(x) = 0$, then x is null and $0 \geq 2^{bh(x)} - 1 = 2^0 - 1 = 0$. Now assume x has height $h > 0$. The children of x either have black-height $bh(x)$ or $bh(x) - 1$, depending on their color. Hence by induction, the subtrees

rooted at the children of x have at least $2^{bh(x)-1} - 1$ nodes. Hence the tree rooted at x has at least $2(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nodes. ∎

Theorem 23. *A red-black tree with n nodes has height at most $2\lg(n+1)$.*

Proof. Let x be the root of a red-black tree with n nodes. Since every red node has two black children, $bh(x) \geq h(x)/2$. Hence $n \geq 2^{h(x)/2} - 1$. Taking logarithms gives the desired result. ∎

```
RB-insert(T,z):
    BST-insert(T,z) #insert according to BST property
    z.left = T.null
    z.right = T.null
    z.color = red
    while z.p.color = red:
        if z.p = z.p.p.left: #in left subtree of grandparent
            y = z.p.p.right
            if y.color = red: #z has red aunt
                Case 1 - Color flip
                z = z.p.p
            else: #z has black aunt
                if z = z.p.right: # Case 2 - left rotation
                    z = z.p
                    leftRotate(z)
                z.p.color = black # Case 3 - right rotation
                z.p.p.color = red
                rightRotate(z.p.p)
        else:
            #same as if but reverse roles of left and right
        T.root.color = black
```

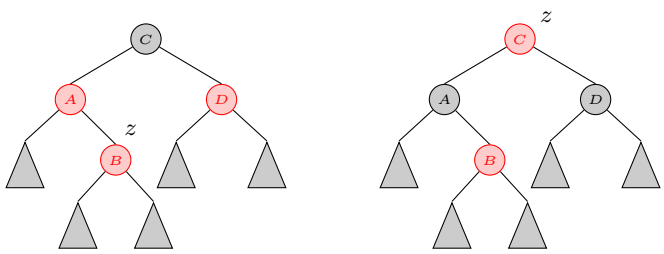


Figure 14: Case 1 - Color flip.

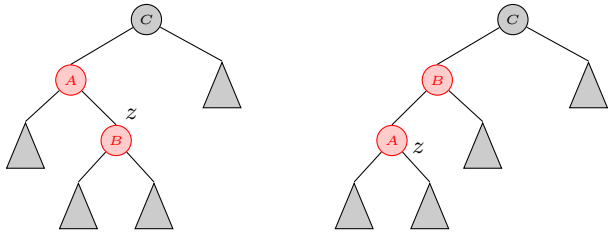


Figure 15: Case 2 - Left rotation about z.p.

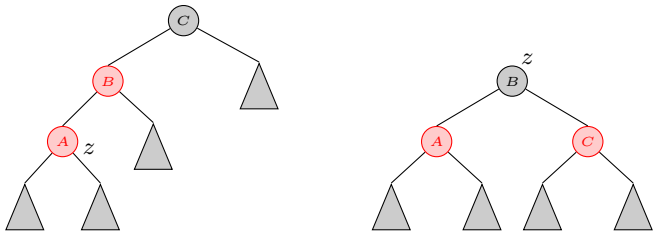


Figure 16: Case 3 - Right rotation about z.`p.p`.

```
RB-delete(T,x):
```

8.4 AVL Trees

AVL property. The heights of the left and right child of any node differ by at most 1. In addition to a key, every node stores its own height, $1 + \max\{h(\texttt{lchild}), h(\texttt{rchild})\}$. We define null nodes to have height -1. Insertion is standard BST insertion. Rotations are used to restore the AVL property. Deletion is also standard BST delete followed by rotations.

8.5 Splay Trees

The `splay` operation incrementally moves a node up the tree towards the root via double rotations (and possibly one additional single rotation).

To search in a splay tree, perform a standard BST search. Once x is found, called `splay(`x`)`. To insert a node into a splay tree, insert as in standard a BST, then splay. To delete x from a splay tree, splay and then delete x, find its predecessor y and splay it, before connecting to x's right subtree.

Theorem 24 (Access Theorem)**.** *Let $\{w_i\}$ be a set of arbitrary positive weights on the nodes of the BST. Define the size $s(x) :=$ total number of descendents of x and the rank, $r(x) := \lg s(x)$. The potential function is $\Phi := \sum_i r(i)$. Then the amortized time to splay x to the root t is $\leq 3(r(t) - r(x)) + 1 = O\left(\lg \frac{s(t)}{s(x)}\right)$.*

Proof. It is sufficient to show the result holds for one rotation and then theorem follows by telescoping. We will only show the analysis for a zig-zig rotation, as the analysis for zig-zag is similar. Suppose z's left child is y and y's right child is x. After a zig-zig rotation, x now has y as a right child and y has z as a right child. Moreover, the change in rank is $\Delta r(x) + \Delta r(y) + \Delta r(z)$. Since $r'(x) = r(z)$, it follows that the amortized cost is

$$\begin{aligned} O(1) + \Delta r(y) + r'(z) - r(x) &\leq O(1) + \Delta r(x) + \Delta r(z) \\ &= O(1) + (r'(z) - r(x)) + \Delta r(x). \end{aligned} \tag{13}$$

Hence it suffices to show $O(1) + r'(z) - r(x) \leq 2\Delta r(x)$. Equivalently,

$$\lg \frac{s'(z)}{s'(x)} + \lg \frac{s(x)}{s'(x)} \leq -O(1).$$

Observe that the descendents of x in the original tree are disjoint from the descendents of z in the rotated tree. Hence $s'(z)$ and $s(x)$ are essentially complementary. Hence if $a = \frac{s'(z)}{s'(x)}$, then $1 - a \geq \frac{s(x)}{s'(x)}$. Since $\lg x + \lg(1-x)$ is maximized at $x = \frac{1}{2}$ on $(0,1)$, the claim is immediate by selecting an appropriate scaling factor on the potential function. ∎

8.5.1 Splay Tree Properties

- (Static Optimality). Suppose we have i items with access frequencies, $0 \leq p_i \leq 1$. The optimal static BST has amortized access cost $O(-\sum p_i \log p_i)$, the Shannon entropy of the access distribution. If we set $w_i = p_i$, the access theorem gives an amortized cost for an access of item i of $O(\log \frac{\sum p_i}{p_i}) = O(-\log p_i)$.
- (Static Finger). Let f be any fixed item. The amortized cost of an access of item i is $O(\lg(|f-i|+1)+1)$. (Take $w_i = \frac{1}{(1+|f-i|)^2}$.)
- (Dynamic Finger). Dynamic finger starts the next search at the previous access. For a sequence of accesses, $x_1, \ldots, x_m$, the amortized cost of the i-th access in a splay tree is $O(\lg(|x_i - x_{i-1}|+1)+1)$.
- (Working Set). If t items have been accessed since the last access to x, then a splay tree finds x in $O(\lg t)$.
- (Unified Access).
- *Dynamic Optimality Conjecture.* It is conjectured that for any sequence of accesses σ, splay trees achieve $O(1)\mathrm{OPT}(\sigma)$, where OPT is the optimal cost of servicing access sequence σ in the BST model (following pointers and performing rotations). Tango trees achieve $O(\lg\lg n)\mathrm{OPT}$.

9 Simplex Algorithm

$$\begin{array}{lrcl} \text{maximize} & 3x_1 + x_2 + x_3 & & \\ \text{subject to} & x_1 + x_2 + x_3 & \leq & 30 \\ & 2x_1 + 2x_2 + 5x_3 & \leq & 24 \\ & 4x_1 + x_2 + 2x_3 & \leq & 36 \\ & x_1, x_2, x_3 & \geq & 0 \end{array}$$

Convert to equivalent slack form by introducing new basic variables x_4, x_5, x_6:

$$\begin{array}{rcl} z & = & 3x_1 + x_2 + x_3 \\ x_4 & = & 30 - x_1 - x_2 - x_3 \\ x_5 & = & 24 - 2x_1 - 2x_2 - 5x_3 \\ x_6 & = & 36 - 4x_1 - x_2 - 2x_3 \\ x_i & \geq & 0 \end{array}$$

Basic solution: Set all non-basic variables to 0, compute values of all basic variables. In this example, $(x_1, x_2, x_3, x_4, x_5, x_6) = (0, 0, 0, 30, 24, 36)$ and $z = 0$.

Pivoting: Select some non-basic variable, x_i whose coefficient in the objective function is positive. Increase the value of x_i as much as possible without violating any constraints. Then x_i becomes a basic variable and some other variable becomes non-basic.

For example, suppose we take x_1. The x_6 constraint is the tightest, so we set $x_1 = 9 - \frac{1}{4}x_4 - \frac{1}{2}x_3 - \frac{1}{4}x_6$. We rewrite the remaining constraints substituting for x_1 and moving x_6 to the RHS.

$$\begin{aligned} z &= 27 + \tfrac{1}{4}x_2 + \tfrac{1}{2}x_3 - \tfrac{3}{4}x_6 \\ x_4 &= 21 - \tfrac{3}{4}x_2 - \tfrac{5}{2}x_3 + \tfrac{1}{4}x_6 \\ x_5 &= 6 - \tfrac{3}{2}x_2 - 4x_3 + \tfrac{1}{2}x_6 \\ x_i &\geq 0 \end{aligned}$$

We simply repeat until the maximum value of the objective is obvious (all other variables have non-positive coefficients).

10 Numerical Algorithms

10.1 Newton's Method & Square Roots

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i).} \tag{14}$$

In the particular case where $f(x) = \sqrt{x}$, we have $x_{i+1} = \frac{x_i + \frac{a}{x_i}}{2}$. To estimate the the rate of convergence, we can write $x_n = \sqrt{a}(1 + \epsilon_n)$. Then, substituting into the recursion, we obtain $\epsilon_{n+1} = \frac{\epsilon_n^2}{2(1+\epsilon_n)}$. Thus, in order to get d digits of precision, we need $\lg d$ iterations.

10.2 Efficient Integer Multiplication

Work in radix r (typically 2 or 10). Let x, y be two n-digit integers, $0 \leq x, y \leq r^n$. Write $x = x_1 r^{n/2} + x_0$ and $y = y_1 r^{n/2} + y_0$. Define

$$\begin{aligned} z_0 &= x_0 y_0 \\ z_1 &= x_0 y_1 + x_1 y_0 \\ z_2 &= x_1 y_1. \end{aligned}$$

Then $z = xy = z_2 r^n + z_2 r^{n/2} + z_0$. The standard multiplication algorithm, therefore satisfies the recurrence $T(n) = 4T(\frac{n}{2}) + \Theta(n)$, and hence runs in $\Theta(n^2)$ time. Karatsuba's algorithm improves upon this by noting that

$$z_1 = (x_0 + x_1)(y_0 + y_1) - z_0 - z_2,$$

and hence satisfies the recurrence, $T(n) = 3T(\frac{n}{2}) + \Theta(n)$, i.e. $T(n) = \Theta(n^{\lg 3}) = O(n^{1.58497})$.

- Toom-Cook: Generalizes Karatsuba to split x and y into $k \geq 1$ parts. For instance, Toom(3) satisfies the recurrence $T(n) = 5T(\frac{n}{3}) + \Theta(n)$, i.e. $T(n) = \Theta(n^{\log_3 5)} = O(n^{1.466})$. However, due to constant factors, Karasuba is more common in practice.
- Schönhage-Strassen: $\Theta(n \lg n \lg \lg n)$ time using FFT.
- Fürur (2007): $\Theta(n \lg n 2^{\log^* n})$, where $\log^*$ denotes the iterated logarithm.

10.3 Efficient Division

To compute $\frac{a}{b}$, we will first compute $\frac{1}{b}$ and then run an efficient multiplication algorithm. In order to compute $\frac{1}{b}$, we must first choose an integer R, sufficiently large, and such that dividing by R is "easy" (typically $R = 2^k$).

Now define $f(x) = \frac{1}{x} - \frac{b}{R}$. Note that f has a root at $\frac{R}{b}$. By Newton's method, we can approximate this root via

$$x_{i+1} = 2x_i - \frac{b}{R}x_i^2.$$

Using similar error analysis as before, Newton's method has quadratic convergence.

11 Hash Functions

Hash functions deterministically map arbitrary stings of data to fixed length outputs. Hash functions are typically public and use no secret keys. Define $h : \{0,1\}^* \to \{0,1\}^d$. Ideally, we have h to be computable in polynomial time in the length of the input string. There are a number of ideal properties hash function should have.

OW *One-wayness*: It is infeasible, given $y \in \{0,1\}^d$, to find an x such that $h(x) = y$.

CR *Collision Resistance*: It is infeasible to find x and x' such that $x \neq x'$ and $h(x) = h(x')$.

TCR *Target Collision Resistance*: It is infeasible, given x, to find $x' \neq x$ such that $h(x) = h(x')$.

PRF *Pseudorandom Function*: Indistinguishable from randomness.

NM *Non-malleability*: It is infeasible given $h(x)$ to produce $h(x')$, where x and x' are related in some way (related inputs should not imply related outputs).

Observe that any function which is CR is TCR, but the converse does not hold. Furthermore, OW does not imply TCR and TCR does not imply OW.

11.0.1 Applications

1. Password Storage. Given a password P, we store $h(P)$ instead of P. To verify a login with typed password P_T, we simply need to check whether $h(P_T) = h(P)$. Clearly, we should require that h be OW (so exposing $h(P)$ does not expose P) and should be somewhat CR (so entering the wrong password does not accidentally allow access).

2. File Modification Detection. Given a file F store $h(F)$. To check if F has been modified, we recompute its hash and compare against $h(F)$. We require TCR, so that we cannot modify a file and obtain the same hash.

3. Digital Signatures. Suppose Alice has a public key PK_A and private key SK_A. Alice signature on a message M is $\sigma = \texttt{sign}(SK_A, M)$. To verify a signature, $\texttt{verify}(M, \sigma, PK_A)$ returns True/False. Requires TCR (so Bob cannot claim Alice signed M' because $h(M) = h(M')$).

4. Commitments. For example consider a silent auction. Suppose Alice has value x for the given item. Alice bids $h(x)$, publically. When the bidding is over, Alice reveals her bid. We require OW (so bid remain hidden), CR (so Alice can't open her bid in multiple ways), and NM (so given $h(x)$, it should not be possible to produce $h(x+1)$).

11.1 Symmetric Key Encryption

Given a secret key k (128-bits), an encryption function e_k, maps a message m to cipher text c. A corresponding decrytion function, d_k maps c to m. The most common symmetric key encryption protocol is AES.

11.1.1 Key Exchange

Diffie-Hellman Key Exchange. We are publically given $g, p \in \mathbb{F}_p$. Alice randomly selects $a \in \mathbb{F}_p$ and Bob randomly selects $b \in \mathbb{F}_p$. Alice computes g^a and sends it to Bob and Bob computes g^b and sends it to Alice. Then both can compute g^{ab} as their shared secret key. The strength of Diffie-Hellman is the inherent difficulty of the discrete logarithm problem: given g^a, compute a, or in this case, given g^a and g^b, compute g^{ab}.

The fundamental flaw to Diffie-Hellman is the *man-in-the-middle attack*. Mal can intercept g^a and instead send g^c to Bob. Similarly, Mal sends g^c to Alice instead of g^b. Then Bob will send messages to Alice using the secret key g^{bc} and Alice sends messages to Bob using secret key g^{ac} (both of which are known to Mal). Hence Mal can listen in to Bob and Alice's communications, without either of them knowing. To avoid the man-in-the-middle attack, Alice and Bob may use public key encryption to exchange their keys.

11.2 Public Key Encryption (RSA)

Alice secretly picks two (large) primes p and q. Alice then computes $N = pq$ and $\phi = (p-1)(q-1)$ and then finds e such that $(e, \phi) = 1$. Alice's public key is (N, e). The decryption exponent, d, satisfies

$$ed \equiv 1 \mod \phi$$

is obtained via the extended Euclidean algorithm. Alice's private key is (d, p, q). Then $c = m^e \mod N$ and $m = c^d \mod N$.

Since $ed \equiv 1 \mod \phi$, we can write $ed = 1 + k\phi$ for some integer k. If $(m, p) = 1$, then by FLT, $m = m^{ed} = (m^{p-1})^{k(q-1)} = m \mod p$ and similarly, if $(m, q) = 1$. If $(m, p) = p$, then trivially $m^{ed} \equiv m \mod p$. So in either case, $m^{ed} \equiv m \mod p$ and mod q. Hence $m^{ed} \equiv m \mod N$. The inherent strength of RSA lies in the difficulty of factoring N and hence computing d.

12 Miscellaneous Sorting

12.1 Counting & Radix Sort: Linear Time Sorts

We assume keys are integers in $\{0, \ldots k-1\}$.

```
Count_Sort(x_1, ..., x_n): # O(n + k)
  L = array of k empty lists
  for j in 1..n:
      L[x_j.key].append(x_j)
  output = []
  for i in 0..(k - 1):
```

```
        output.append(L[i])
```

Radix sort in base b. All keys in interpreted in base b and have $d = \log_b k$ digits. Radix sort, used counting sort to sort by the least significant bit, followed by the next least significant bit, and so on, until we sort by the most significant bit. Thus radix sort is $O(d(n+b))$. By taking $b = n$, radix sort is $O(n)$ for $k = n^{O(1)}$.

12.2 Lower bounds of Comparison Sorting

Theorem 25. *Sorting n items in the comparison based model requires $\Omega(n \log n)$ comparisons.*

Proof. The decision tree is binary and has $n!$ leaves. By Stirling's approximation, the height of the decision tree (and thus the number of required comparison) is at least $\log n! = \Omega(n \log n)$. ∎

Similar, analysis gives a $\Omega(\lg n)$ bound for searching in the comparison model.

13 Divide & Conquer

13.1 Convex Hulls

Given $S = \{(x_i, y_i) : 1 \leq i \leq n\}$ we want $\texttt{CH}(S)$, the smallest convex polygon containing S. $\texttt{CH}(S)$ returns a sequence of points which form the boundary of the convex hull, in clockwise order. We assume no two points in S have the same x-coordinate or the same y-coordinate. Furthermore, assume not 3 points of S are collinear. The naive algorithm includes the segment formed by $p, p' \in S$ if and only if all points of S lie on one side of the line determined by p and p'. This is $\Theta(n^3)$.

A divide-and-conquer approach works by sorting the points by their x-coordinates. Then dividing S into two halves, A and B, by x-coordinate. We compute $\texttt{CH}(A)$ and $\texttt{CH}(B)$ (using the naive algorithm once the sizes of A and B are sufficiently small) and merge the resulting convex hulls together.

The merge process works as follows. Let $a_1, \ldots, a_k$ denote the $\texttt{CH}(A)$ and $b_1, \ldots, b_\ell$ denote $\texttt{CH}(B)$. We assume a_1 is the right-most point of $\texttt{CH}(A)$ and b_1 is the left-most point of $\texttt{CH}(B)$. Let $l = \frac{a_1 + b_1}{2}$. Define $y(i, j)$ denote the point where the line determined by a_i and b_j intersects $x = l$, namely

$$y(i, j) = \frac{y_j - y_i}{x_j - x_i}(l - x_i) + y_i.$$

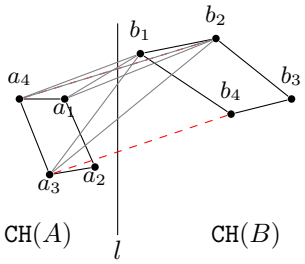
The `Two-Finger` algorithm, returns the upper and lower tangents connecting $\texttt{CH}(A)$ and $\texttt{CH}(B)$.

```
Two-Finger: # O(n)
   i, j = 1
   #reverse inequality for lower tangent
   while y(i, j + 1) > y(i, j) or y(i + 1, j) > y(i, j):
       if y(i, j + 1) > y(i, j):
           j = j + 1 mod ℓ #move right finger
       else:
```

```
            i = i + 1 mod k  #move left finger
    return (a_i, b_j)
```

If the upper tangent is given by (a_i, b_j) and the lower tangent is (a_s, b_t), then $\texttt{CH}(S)$ is

$$a_i, b_j, b_{j+1} \ldots, b_t, a_s, a_{s+1} \ldots, a_i.$$



i	j	$y(i,j)$
1	1	2
1	2	4
2	1	1
2	2	3

Figure 17: Note the upper tangent is not the line connecting the highest point of $\texttt{CH}(A)$ and $\texttt{CH}(B)$ (shown in red). The upper tangent is (a_4, b_1). The lower tangent is (a_2, b_3). The merged convex hull is $a_4, b_1, b_2, b_3, a_2, a_3, a_4$. Some of the line analyzed by `Two-Finger` are shown in gray. An ordinal ranking of $y(i,j)$ is given by the table, so $y(2,1)$ is the largest, for example.

13.2 Median Finding

Given a set of n distinct numbers, let rank(x) be the number of elements less than x. We want to find the element of a particular rank.

```
select(S, k):
    pick x ∈ S
    compute i = rank(x)
        B = {y ∈ S : y < x}
        C = S\({x} ∪ B)
    if i = k: return x
    else if i > k: select(B,k)
    else: select(C, k − i)
```

The crux of the k-`select` algorithm is the choice of x. If we can guarantee that x's rank is some fixed fraction of the size of S, then we ought to achieve $\Theta(n)$ runtime.

To pick x we split S into $\lceil \frac{n}{5} \rceil$ columns with 5 elements each and sort each of these columns. We then select the middle element of each column and recursively compute the median of this set. Observe that by this choice of x, at least $3\left(\lceil \frac{n}{10} \rceil - 2\right)$ elements are greater than x and $3\left(\lceil \frac{n}{10} \rceil - 1\right)$ elements are less than x. So for sufficiently large n, our algorithm satisfies the recurrence

$$T(n) = T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + \Theta(n),$$

so $T(n)$ is $\Theta(n)$.

Probabilistic Selection?

14 Miscellaneous Problems

- Given $A, B\{1, \ldots, n\}$, compute $A + B = \{a + b : a \in A, b \in B\}$ in $O(n \log n)$ time.

  Solution 1. *Let $P_A(x) = \sum_{x \in A} x^i$ and define P_B analogously. Compute $P_A \cdot P_B$, using FFT. Then $A + B\{i \in \{2, \ldots, 2n\} : [x^i] \neq 0\}$.*

- Suppose we are given $0 \leq a_1 < \ldots < a_n$ and we want to minimize

$$f(x) = \sum_{a_i < x} 20(x - a_i) + \sum_{a_i > x} 30(a_i - x).$$

  (Intuitively, we can view this problem as finding the optimal drop point for a set of packages which must be delivered to houses at a_i, respectively, on a sloped hill. If x is the drop point, it takes $20(x - a_i)$ minutes to deliver a package to a house downhill from x and return to the drop zone, whereas it takes $30(a_i - x)$ minutes to deliver uphill).

  Solution 2. *Let $LE(x)$ be the number of a_i less than x, $GE(x)$ be the number of a_i greater than x and $EQ(x)$ the number of houses at x. Let $\epsilon > 0$ be sufficiently small. Then*

$$f(x + \epsilon) - f(x) = 20\epsilon(LE(x) + EQ(x)) - 30\epsilon GE(x)$$

$$f(x - \epsilon) - f(x) = 30\epsilon(GE(x) + EQ(x)) - 20\epsilon LE(x)$$

  At an optimal point, x_0, both of the above quantities would be negative. Using the fact that $GE(x) + LE(x) + EQ(x) = n$, this implies $LE(x_0) < \frac{3}{5}n$ and $GE(x) < \frac{2}{5}n$, hence the drop location $a_{\lceil \frac{3n}{5} \rceil}$ works.

- Let $a_1, \ldots, a_n$ be real numbers. We want to find $i, j \in \{1, \ldots, n\}$ such that $i < j$ and $a_j - a_i$ is maximized.

  Solution 3. *We'll apply a divide and conquer approach.*

```
big_Diff(A: sequence):
    A_1 : A_2 = A #split A in half
    (O_1, m_1, M_1) = big_Diff(A_1)
    (O_2, m_2, M_2) = big_Diff(A_2)

    return (max{O_1, O_2, M_2 − m_1}, min{m_1, m_2}, max{M_1, M_2})
```

- We are given a sequence $x_1, \ldots, x_n \in \{0, \ldots, p-1\}$. We want to know if the operations $+$ and $\times$ (using their usual precedence) can be inserted in between so that the resulting expression is congruent to 0 mod p.

  Solution 4. *Let $T[a, b, i]$ be true if we can insert $+, \times$ so that the first i numbers so that the part before the last $\times$ has value a and afterwards has value b. If $T[a, b, i-1]$ is true, set $T[a, bx_i \bmod p, i]$ and $T[a, b + x_i \bmod p, i]$ to true. If there exists $a \in \{0, \ldots, p-1\}$ such that $T[a, 0, n]$ is true, we output true, otherwise output false. $O(np^2)$.*

- Let $G = (V, E)$ be a directed graph and $u, v \in V$ such that there is no u-v path. We want to find an edge $e \in E$ such that if we flip e, then there will be a u-v path.

  Solution 5. *Run explore from u and mark all reached vertices 'red'. Run explore from v in G^R, marked all reached vertices 'blue'. Find an edge from a blue to a red vertex.*

- A sequence $b_1, \ldots, b_k$ *convex* if $2b_i \leq b_{i-1} + b_{i+1}$ for all $i \in \{2, \ldots, k-1\}$. Given a sequence $a_1, \ldots, a_n$ we want to find a convex subsequence of $a_1, \ldots, a_n$ with the largest sum.

Solution 6. *Let $L[i,j]$ be the length of the largest convex subsequence of $a_i, \ldots, a_n$ whose first two elements are a_i and a_j. (We let $S[i,j]$ be the index of the element which comes after a_i, a_j in the optimal subsequence corresponding to $L[i,j]$).*

$$L[i,j] = a_i + \max\{L[j,k] : j < k \leq n \text{ and } a_i + a_k \geq 2a_j\}$$

$$S[i,j] = \arg \max_{\substack{j<k\leq n \\ a_i+a_k \geq 2a_j}} \{L[i,j]\}$$

- Activity selection with two people. We are given n activities. For $i \in [n]$ the i-th activity is given by an interval $[a_i, b_i]$ and a value v_i. Assume $a_i \leq b_i$. We want the two people to choose activities so that
  - they attend disjoint sets of activities,
  - the activities attended by each person are non-overlapping, and
  - the total value of activities attended is maximized.

  More formally, we want $A, B \subseteq [n]$ such that
  - $A \cap B = \varnothing$;
  - for $i, j \in A$ with $i \neq j$ we have $b_i < a_j$ or $b_j < a_i$;
  - for $i, j \in B$ with $i \neq j$ we have $b_i < a_j$ or $b_j < a_i$;
  - $\sum_{i \in A} v_i + \sum_{i \in B} v_i$ is maximized.

  Assume the activities are sorted by end time, i.e. $b_1 \leq b_2 \leq \ldots \leq b_n$. For $i, j \in [n]$, let $T[i,j]$ be the maximum value of a valid selection where the last activity of the first person is i and the last activity of the second person is j, i.e. $\max A = i$ and $\max B = j$. If $j = k$ let $T[i,j] = -\infty$.

  Solution 7. *If $i < j$, then*

$$T[i,j] = v_i + \max_{\substack{1\leq k\leq i-1 \\ b_k < a_i}} \{T[k,j]\}.$$

  If $j < i$,

$$T[i,j] = v_j + \max_{\substack{1\leq k\leq j-1 \\ b_k < a_j}} \{T[i,k]\}.$$

- Let $G = (V, E)$ be a directed graph with edge weights $w : E \to \mathbb{E}$. *We allow negative edge weights.* Assume all cycles have non-negative weight. Fix $s, t \in V$. Let $A \subseteq V$. Give a $O(mn)$ algorithm to find the shortest path from s to t including at least one vertex of A.

  Solution 8. *Run Bellman-Ford at s in G. Let D_s be the distances from s to other vertices of G. Run Bellman-Ford in G^R at t. Let D_t be the distances from t to other vertices of G^R. Compute $\max_{a \in A}\{D_s[a] + D_t[a]\}$.*

  Solution 9. *(Not $O(mn)$ but still good). Create a duplicate graph G'. Let H be the disjoint union of G and G'. For $a \in A$, add edge $a \to a'$ with weight 0. Run Johnson's algorithm.*

- Let $G(V, E)$ be an undirected graph with positive edge weights $w : E \to \mathbb{R}_{\geq 0}$. Fix $s, t \in V$. We want to find the shortest path from s to t with an even number of edges. Construct a new graph H as follows:
  - Let G' be a copy of G with t (and any edges adjacent to t) removed. The weight of edges in G' are the same as the corresponding edges in G. For $v \in V \backslash \{t\}$ the corresponding vertex in G is denoted v'.

- Let G'' be a copy of G with s (and any edges adjacent to s) removed. The weight of edges in G'' are the same as the corresponding edges in G. For $v \in V \backslash \{s\}$ the corresponding vertex in G is denoted v''.
- Let H be the disjoint union of G' and G''. Add edges $\{v', v''\}$ of weight 0 to H for all $v \in V \backslash \{s, t\}$.

Solution 10. *Let $\mathcal{M}$ be the set of edges between G' and G''. This constitutes a matching in H. Find the minimum length augmenting path on $\mathcal{M}$. To recover the original path, remove the edges between G' and G'' from the augmenting path.*

- Given a string S over the alphabet $\Sigma = \{a, b\}$, we want to find a cyclic permutation of S that disagrees with S in the maximum number of places.

Solution 11. *Let $S = s_1 \dots s_n$. Define $(a_i)_{1 \le i \le 2n}$ and $(b_i)_{1 \le i \le n}$ where $a_i = a_{i+n} = b_i = s_i$ for $1 \le i \le n$ mapping $a \mapsto 0$ and $b \mapsto 1$. Let (r_i) be the reverse of (b_i), i.e. $r_i = b_{n-i}$ for $1 \le i \le n$. Define*

$$c_k = \sum_{i=1}^{n} (a_{i+k} - b_i)^2 = \underbrace{\sum_{i=1}^{n} a_{i+k}}_{A_k} + \underbrace{\sum_{i=1}^{n} b_i}_{B} - 2 \underbrace{\sum_{i=1}^{n} a_{i+k} b_i}_{D_k}.$$

for $0 \le k \le n-1$. Note $A_k = A_{k-1} + a_{k+n} - a_k$, so once we compute A_0 we can find $A_1, \dots, A_{n-1}$ in $O(n)$ time. Furthermore,

$$D_k = \sum_{i=1}^{n} a_{i+k} r_{n-i} = ((a_i) * (r_i))_{n+k}.$$

Hence we can compute c_k in $O(n \log n)$ time. Observe that c_k is the Hamming distance of S with its k-th cyclic permutation. Hence we simply want to find k that maximizes c_k.

- (ℓ_∞-minimization). Given n pairs of real numbers $(x_1, y_1), \dots, (x_n, y_n)$, we want to find real numbers a and b such that $\max\{ax_i + y_i - b\}$ is minimized.

Solution 12. *Minimize t subject to*

$$\begin{aligned} -t &\le ax_i + y_i - b \le t \quad \forall i = 1, \dots, n && (15) \\ 0 &\le t && (16) \end{aligned}$$

- (ℓ_1-minimization). Given n pairs of real numbers $(x_1, y_1), \dots, (x_n, y_n)$, we want to find real numbers a and b such that $\sum_i (ax_i + y_i - b)$ is minimized.

Solution 13. *Minimize $\sum_i t_i$ subject to*

$$\begin{aligned} -t_i &\le ax_i + y_i - b \le t_i \quad \forall i = 1, \dots, n \\ 0 &\le t_i \quad \forall i = 1, \dots, n \end{aligned}$$

References

[1] Leiserson, et. al. *Introduction to Algorithms.*