Advanced Algorithms

Lectures by Ely Porat
Summary by Ari Feiglin (ari.feiglin@gmail.com)

Contents

1 Linear Programming

1.1 Linear Programs

Suppose we have a pizzeria which sells different types of pizzas, each with an assortment of toppings. This pizzeria has one-thousand units of dough, 500 units of sauce, and 800 units of cheese. The pizzeria sells the following types of pizzas, which each require the following units of toppings and have the following prices:

	Regular Pizza	**Thin Pizza**	**Extra-Cheese Pizza**
Dough	2	1	2
Sauce	2	3	2
Cheese	1	2	3
	\$25	\$30	\$40

So if we sell x_1 regular pizzas, x_2 thin pizzas, and x_3 extra-cheese pizzas, we must have the following constraints:

$$2x_1 + x_2 + 2x_3 \leq 1000, \qquad 2x_1 + 3x_2 + 2x_3 \leq 500, \qquad x_1 + 2x_2 + 3x_3 \leq 800$$

And we want to maximize our total sales, i.e. maximize $25x_1 + 30x_2 + 40x_3$ where $x_1, x_2, x_3 \geq 0$.
This is an example of *linear programming* (LP).

1.1.1 Definition

A **Linear Program** is the problem of maximizing some objective expression of the form

$$f(x_1, \ldots, x_n \,;\, c_1, \ldots, c_n) = \sum_{i=1}^{n} c_i x_i$$

for some parameters c_i under the constraints

$$\sum_{j=1}^{n} \alpha_{i,j} x_i \leq \beta_i$$

for $1 \leq i \leq m$ where m is the number of constraints.

Notice that if we want to have the constraint $\sum_{j=1}^{n} \alpha_{i,j} x_i = \beta_i$, this is just the same as having two constraints $\sum_{j=1}^{n} \alpha_{i,j} x_i \leq \beta_i$ and $\sum_{j=1}^{n} \alpha_{i,j} x_i \geq \beta_i$. And if we want the constraint $\sum_{j=1}^{n} \alpha_{i,j} x_i \geq \beta_i$, this is equivalent to $\sum_{j=1}^{n} (-\alpha_{i,j} x_i) \leq -\beta_i$.
Now say we want to *minimize* $f(x_1, \ldots, x_n \,;\, c_1, \ldots, c_n)$, this is just the same as maximizing the expression $f(x_1, \ldots, x_n \,;\, -c_1, \ldots, -c_n)$.

1.1.2 Definition

An LP is in **standard form** if it has the constraints $x_i \geq 0$ for every variable x_i.

This seems to weaken the strength of LP, but in fact every general LP can be converted to an equivalent LP in standard form. Suppose we have the LP $f(x_1, \ldots, x_n \,;\, c_1, \ldots, c_n)$ with constraints using $\alpha_{11}, \ldots, \alpha_{n,m}$ and $\beta_1, \ldots, \beta_m$. Then we can define

$$f'(x_1^+, x_1^-, \ldots, x_n^+, x_n^-) = \sum_{i=1}^{n} c_i (x_i^+ - x_i^-)$$

and the constraints

$$\sum_{j=1}^{n} \alpha_{i,j} (x_i^+ - x_i^-) \leq \beta_i \qquad (1 \leq i \leq m)$$

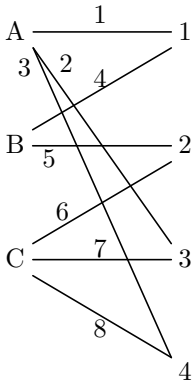
along with $x_i^+, x_i^- \geq 0$. This is equivalent, as we can then just define $x_i = x_i^+ - x_i^-$, we have split x_i into is positive and negative parts.

Thus given an LP in standard form, we can write it as a problem of the form

$$\begin{aligned} \text{Maximize} \quad & \vec{c}^{\top} \cdot \vec{x} \\ \text{Such that} \quad & A\vec{x} \leq \vec{b} \\ & \vec{x} \geq \vec{0} \end{aligned}$$

where $\vec{c}$ is the vector of coefficients $(c_1, \ldots, c_n)^{\top}$, $\vec{x}$ is the vector of variables $(x_1, \ldots, x_n)^{\top}$, and A is the matrix such that $A_{ij} = \alpha_{ij}$. We write $\vec{a} \leq \vec{b}$ to mean $a_i \leq b_i$ for every i.

Example: Suppose we have the following graph (the edges are labelled), and we'd like to find the maximum number of matches we can find.



The LP form of this would be maximizing $\sum_{i=1}^{8} x_i$ (where x_i denotes whether or not the ith edge was chosen), with the constraints ($x_i \geq 0$ is implicit)

$$\begin{aligned} x_1 + x_2 + x_3 &\leq 1 \\ x_4 + x_5 &\leq 1 \\ x_6 + x_7 + x_8 &\leq 1 \\ x_1 + x_4 &\leq 1 \\ x_5 + x_6 &\leq 1 \\ x_2 + x_7 &\leq 1 \\ x_3 + x_8 &\leq 1 \end{aligned}$$

But we also want the constraint that x_i be an integer. This is called *Integer Linear Programming*. This is an NP-hard problem, as we will see in the following example. ◊

Example: Now let us look at another example: the problem of *Vertex Covers* (VC). Given a graph $G = (V, E)$, we want to find the smallest vertex cover, which is a set of vertices S such that every $v \in V$ has a neighbor in S. Now, we can use LP as follows: associate each vertex $v \in V$ with a variable x_v. Then we want to minimize $\sum_{v \in V} x_v$. And we add the constraints that for every $(u, v) \in E$, $x_u + x_v \geq 1$ (i.e. either u or v is in S). We can solve this using integer linear programming, and since this is an NP-complete problem (whether G has a VC of a particular size), so too is integer linear programming.

Using LP won't necessarily give us a valid solution to the problem since we can have that x_v be a non-integer. We only know that the answer LP gives us is at most VC_{opt} (the optimal solution), since the optimal solution satisfies the constraints. In fact, if we define $\text{Sol} = \left\{ v \;\middle|\; x_v \geq \frac{1}{2} \right\}$ then we have that

$$\text{LP} \leq \text{VC}_{opt} \leq |\text{Sol}| \leq 2\text{LP}$$

This is because Sol must be a vertex cover: if there exists a $v \in V$ with no neighbors in Sol, this means that $x_v < \frac{1}{2}$ and $x_u < \frac{1}{2}$ for every $(v, u) \in E$. But then we can just set $x_v = \frac{1}{2}$ and this will satisfy the constraints. And if we multiply LP by 2, then every vector in Sol gets multiplied by 2, and so summing them gives Sol. ◊

1.1.3 Definition

The **slack form** of an LP is one of the form:

$$\begin{aligned} \text{Maximize} \quad & \vec{c}^{\top} \cdot \vec{x} \\ \text{Such that} \quad & A\vec{x} = \vec{b} \\ & \vec{x} \geq \vec{0} \end{aligned}$$

Every LP in general form can be brought to one in slack form by adding variables. For every constraint $\sum_{j=1}^{n} \alpha_{i,j} x_j \leq \beta_i$ we add a basic variable s_i and alter the constraint to be $\sum_{j=1}^{n} \alpha_{i,j} x_j + s_i = \beta_i$. Since we must also add the constraint $s_i \geq 0$, this is equivalent. The coefficients of the basic variables in the expression we optimize is of course zero.

We will also assume that in slack form, $\vec{b} \geq \vec{0}$ as otherwise there may be no solution.

1.2 The Simplex Algorithm

First we begin by defining a few mathematical concepts.

1.2.1 Definition

A **convex set** $S \subseteq \mathbb{R}^n$ is a set such that for every $x, y \in S$, $\lambda x + (1-\lambda)y \in S$ for all $\lambda \in [0,1]$. The **convex hull** of a set of points $\{p_1, \ldots, p_m\}$ is the smallest convex set containing these points. It can be shown that the convex hull is

$$[p_1, \ldots, p_n] = \left\{ \sum_{i=1}^{m} \lambda_i p_i \;\middle|\; \sum_{i=1}^{m} \lambda_i = 1 \right\}$$

An **extreme point** of a convex set is a point $x \in S$ such that there exists no $a, b \in S$ such that x lies on the open segment between a and b. I.e. there exist no $a, b \in S$, $\lambda \in (0,1)$ such that $x = \lambda a + (1-\lambda)b$. Two extreme points x, y are **adjacent** if the line connecting them is not intersected by any other line between two points not on the line between x and y.

Notice that if $[p_1, \ldots, p_n]$ is a convex hull, then every extreme point is one of the p_is.

1.2.2 Definition

A **convex polytope** is the region of some n-dimensional space defined by $A\vec{x} \leq \vec{b}$. The extreme points of a convex polytope are called **vertices** and there are finitely many of them (in fact a convex polytope is the convex hull of its vertices).

Notice that the space of *feasible solutions* to an LP problem under the constraint $A\vec{x} \leq \vec{b}$ and $\vec{x} \geq 0$ forms a convex polytope (it is obviously a polytope, convexity is easy to verify). We will use the following results for constructing an algorithm to find the maximum of some objective $\vec{c}^{\top}\vec{x}$.

1.2.3 Lemma

Let P be a convex polytope with vertices $v_1, \ldots, v_n$ and f be a convex function over P. Then f attains its maximum at one of the vertices.

Proof: suppose x is the maximum of f, then since it is in P, it can be written as $x = \sum_{i=1}^{n} \lambda_i v_i$ for $\sum_{i=1}^{n} \lambda_i = 1$ and $\lambda_i \geq 0$. So by convexity

$$f(x) = f\left(\sum_{i=1}^{n} \lambda_i v_i\right) \leq \sum_{i=1}^{n} \lambda_i f(v_i) \leq \sum_{i=1}^{n} \lambda_i f(x) = f(x)$$

Thus $f(x) = \sum_{i=1}^{n} \lambda_i f(v_i)$, but $f(v_i) \leq f(x)$ for all i so $f(v_i) = f(x)$ for all i with $\lambda_i \neq 0$, meaning v_i must be a maximum. ■

So to find the maximum of the objective, it is sufficient to find its maximum over the vertices of the polyope defined by $A\vec{x} \leq \vec{b}$. But there are many vertices, so how can we effectively find the maximum?

1.2.4 Lemma

Let P be a convex polytope and a linear function $f(\vec{x}) = \vec{c}^\top \vec{x}$, then if a vertex is larger than all its adjacent vertices, it is a maximum.

We will prove this later. But this tells us that if we start from a vertex and traverse to adjacent vertices such that each one is larger than the next, we will eventually reach a vertex greater than all its neighbors and this must be a maximum.

Notice that when we consider $A\vec{x} = \vec{b}$, we can assume that A has full row rank, since otherwise there is no solution or A has extra equations which we can remove.

1.2.5 Definition

A **basis** of the polytope defined by $A\vec{x} \leq \vec{b}$ is a set of indexes B such that when we take the matrix whose columns are the columns of A whose index are in B, denoted A_B, A_B is invertible.

If we have the boundary of a convex polytope $A\vec{x} = \vec{b}$ where $A \in \mathbb{R}^{m \times n}$, then the vertices are the vectors $\vec{x}$ where $x_i = 0$ for $i \notin B$ where B is some basis. Call such vectors *basic feasible solutions* (bfs), we will show later why these are precisely the vertices of the polytope. For each basis B, such a point is unique, since they must have $A_B\vec{x} = \vec{b}$ and so $\vec{x} = A_B^{-1}\vec{x}$.

These are indeed vertices: we prove this by showing that they are extreme points. Suppose that $\vec{\alpha}, \vec{\beta}$ are two points in the polytope such that $\lambda\vec{\alpha} + (1-\lambda)\vec{\beta} = \vec{x}$ for $\lambda \in (0,1)$. Then on one hand

$$A(\lambda\vec{\alpha} + (1-\lambda)\vec{\beta}) = \lambda A\vec{\alpha} + (1-\lambda)A\vec{\beta} \leq \lambda\vec{b} + (1-\lambda)\vec{b} = \vec{b}$$

but on the other hand this must be an equality since $\lambda\vec{\alpha} + (1-\lambda)\vec{\beta} = \vec{x}$. So $A\vec{\alpha} = A\vec{\beta} = \vec{b}$. For every i for which $x_i = 0$ we must have $\alpha_i = \beta_i = 0$ since $\lambda\alpha_i + (1-\lambda)\beta_i = 0$ and $\alpha_i, \beta_i \geq 0$. But we said that such points are unique, so $\vec{\alpha} = \vec{\beta} = \vec{x}$, so $\vec{x}$ is an extreme point.

We also make the following observation: maximizing $c^\top\vec{x}$ by $A\vec{x} = \vec{b}$ is the same as finding a maximal v such that there exists a $\vec{x}$ where

$$\begin{pmatrix} A & 0 & b \\ -c^\top & 1 & 0 \end{pmatrix} \begin{pmatrix} \vec{x} \\ v \\ -1 \end{pmatrix} = 0$$

The left matrix is called the *tableau* of the LP.

Now suppose B is a basis for A, then if we multiply by $A_B^{-1} \oplus 1$ (which is legal since it is invertible) we get that we can solve for the tableau

$$\begin{pmatrix} I \mid_B A' & 0 & A_B^{-1}b \\ -c^\top & 1 & 0 \end{pmatrix}$$

Where $I \mid_B A'$ means that the columns of matrix I are inserted at indexes in B to the matrix A'. Let us assume that $I \mid_B A' = (I\ A')$, then we have a tableau of the form

$$\begin{pmatrix} I & A' & 0 & \bar{b} \\ -c_1^\top & -c_2^\top & 1 & 0 \end{pmatrix}$$

Performing some row-reduction gives us

$$\begin{pmatrix} I & A' & 0 & \bar{b} \\ 0 & -\bar{c}_2^\top & 1 & z_B \end{pmatrix}$$

Recall that the vertex associated with the basis B has zeroes when $i \notin B$, so multiplying by $(\vec{x}\ v\ -1)^\top$ gives us

$$\begin{pmatrix} I & A' & 0 & \bar{b} \\ 0 & -\bar{c}_2^\top & 1 & z_B \end{pmatrix} \begin{pmatrix} \vec{x} \\ v \\ -1 \end{pmatrix} = \begin{pmatrix} \vec{x}_B - \bar{b} \\ v - z_B \end{pmatrix}$$

So we have that $\vec{x}_B = \bar{b}$ gives us the value $v = z_B$ ($\vec{x}_B$ is the vector $\vec{x}$ with indexes not in B removed). What this tells us that if we evaluate $x_i = \bar{b}_i$ for $i \in B$ and $x_i = 0$ for $i \notin B$ we get a vertex of the polytope, and its value is z_B.

Using the second lemma, that a vertex is a maximum if and only if its value is greater than all its neighbors, we can continue this algorithm. How do we determine if two vertices are adjacent? We will answer this through the following lemmas.

1.2.6 Lemma

Let B_1, B_2 be two bases which differ in a single index, and x, y be the bfs-s corresponding to the bases. Then for every point a in the boundary of the polytope (meaning $Aa = b$), if $a_i = 0$ for every $i \notin B_1 \cup B_2$, a lies on the line between x and y.

Proof: we will show that the line passing through a and x contains y. That is, there exists a λ such that $\lambda a + (1-\lambda)x = y$. Suppose that $B_1 \setminus B_2 = \{j\}$ and $B_2 \setminus B_1 = \{k\}$. Then it is sufficient that $\lambda a_i + (1-\lambda)x_i = 0$ for $i \notin B_2$, since $A(\lambda a + (1-\lambda)x) = b$ and such a point is uniquely y. Therefore it is sufficient that $\lambda a_j + (1-\lambda)x_j = 0$, i.e. $\lambda(a_j - x_j) = -a_j$. This has a solution when $x_j \neq a_j$.

Thus the only issue arises when $x_j = a_j$. Let us denote $A_i = A_{B_i}$, then

$$Aa = b \implies A_1^{-1}Aa = x \implies (I \mid_B A_1^{-1}A_{B_1^c})a = x$$

Since $a_i = 0$ for $i \notin B_1 \cup B_2$, we have that

$$a_{B_1} + a_k A_1^{-1} C_k(A) = a_{B_1} + a_k C_k(A_1^{-1}A) = x$$

Thus $x_j = a_j + a_k[A_1^{-1}A]_{jk}$, so $a_k[A_1^{-1}A]_{jk} = 0$. If $a_k = 0$ then $a = x$ by uniqueness. Otherwise, $[A_1^{-1}A]_{jk} = 0$ why can't this happen? ■

Generalizing this lemma, we get the converse of what we stated earlier: a point is a vertex iff it is a bfs. This is since we can generalize this lemma to say that if $B_1, \ldots, B_k$ differ each by a point and $Aa = b$ such that $a_i = 0$ for $i \notin \bigcup_j B_j$, then a lies on the convex hull defined by $x_1, \ldots, x_k$ where x_i is the bfs defined by by B_i. So the polytope is the convex hull of all the bfs-s, and thus each extreme point must be a bfs.

1.2.7 Lemma

Two bfs-s are adjacent iff their bases differ by a single point.

Proof: suppose that B_1, B_2 are bases which differ by a single index. Let x, y be the corresponding bfs-s, and suppose that α, β are two points in the polytope such that the open segment between them intersects the line segment between x and y. Then there exists $\gamma \in (0,1), \lambda \in [0,1]$ such that

$$\lambda x + (1-\lambda)y = \gamma\alpha + (1-\gamma)\beta$$

this means that for $i \notin B_1 \cup B_2$, $\gamma\alpha_i + (1-\gamma)\beta_i = 0$, since $\alpha, \beta \geq 0$ this means $\alpha_i = \beta_i = 0$. But then by the above lemma, they are both on the line segment of x and y, so x and y are adjacent.

Show converse ■

Proof (of lemm 1.2.4):

So now let us discuss the significance of these lemmas on our algorithm. A current naive algorithm would be to iterate over all the bases B, and multiply the tableau by A_B^{-1}. Notice that this is just equivalent to row-reducing the tableau at the columns indicated by B. Recall that if we have a basis B, our tableau will be in the following form:

$$\begin{pmatrix} I \mid_B A' & 0 & \bar{b} \\ 0 \mid_B -\bar{c}^\top & 1 & z_B \end{pmatrix}$$

and the value at the vertex x_B (the bfs defined by B) is z_B. Choosing a basis B' which differs from B by a single index and then looking at the resulting tableau is the same as choosing a column from A' and using an element from that column as a pivot. The column we choose is what will determine the *entering index* (or entering variable), and the row we choose will determine which column from B will leave (the *leaving index* or variable).

Suppose we choose the index i to enter and the index j to leave, i.e. the column i and the row j. Then to row reduce the final row, we must add $\bar{c}_j / A'_{ji}$, resulting in a change of z_B to $z_{B'} = z_B + \frac{\bar{c}_j}{A'_{ji}}\bar{b}_j$. So we want to find the i, j which maximize $\frac{\bar{c}_j \cdot \bar{b}_j}{A'_{ji}}$. Notice that if all of c is negative, then this means that our current bfs is larger than all its neighbors and so it is maximal by an above lemma.

But we also need our new solution to be feasible, that is we cannot have that any b_k turns negative. The new value of b_k is, when i, j are chosen, $b_k - \frac{b_j}{A_{ji}}A_{ki}$. So we must have that

$$\frac{b_k}{A_{ki}} \geq \frac{b_j}{A_{ji}}$$

So after choosing j, we must choose i to minimize b_j/A_{ji}.

So what we do is we choose the column j such that $\bar{c}_j$ is maximal (the most negative value in the bottom row) (this is to find the optimal neighbor), and then choose the row i such that b_j/A_{ji} is minimal (to ensure feasibility).

This is the simplex algorithm. It is correct since at the worst case we visit every vertex, but that's still a finite number of vertices. Why is it called the *simplex* algorithm? Who knows, in my opinion a more fitting name would be "the convex polytope algorithm", but that's a bit of a mouthful.

1.3 The Simplex Algorithm; In Short

Suppose we want to solve the following LP in slack form: maximize $c^\top x$ constrained to $Ax = b$ where $A \in \mathbb{R}^{m\times n}$. Then we write out the *initial tableau*:

$$\begin{pmatrix} A & 0 & b \\ -c^\top & 1 & 0 \end{pmatrix}$$

If A contains the $m \times m$ identity matrix, then say that the tableau is in *canonical form*. Changing names of variables (i.e. column swapping), this means that the tableau is of the form

$$\begin{pmatrix} I & A & 0 & b \\ -d^\top & -c^\top & 1 & 0 \end{pmatrix}$$

Row reducing gives us

$$\begin{pmatrix} I & A & 0 & b \\ 0 & -c^\top & 1 & z_B \end{pmatrix}$$

z_B is a feasible solution to the problem when we set the variables in the columns corresponding to the identity matrix to the values of b and all the other variables to zero.

If the tableau is not in canonical form, we can find m independent columns and row reduce those to get the identity.

If c has no positive values, then we finish and z_B is the maximum value. Otherwise, choose the column i for which c_i is maximal. Then choose the row j for which $\frac{b_j}{A_{ji}}$ is minimal, and use the index (j, i) as a pivot in row reducing. This will give (after swapping columns)

$$\begin{pmatrix} I & \bar{A} & 0 & \bar{b} \\ 0 & -\bar{c}^\top & 1 & z_{B'} \end{pmatrix}$$

where $z_{B'} > z_B$, and this is another feasible solution. Continue iteratively choosing a pivot until c has no positive values.

Example: suppose we want to maximize

$$2x + 3y + 4z$$

subject to

$$\begin{aligned} 3x + 2y + z &\leq 10 \\ 2x + 5y + 3z &\leq 15 \\ x, y, z &\geq 0 \end{aligned}$$

Adding slack variables, this is equivalent to

$$\begin{aligned} 3x + 2y + z + s_1 &= 10 \\ 2x + 5y + 3z + s_2 &= 15 \\ x, y, z, s_1, s_2 &\geq 0 \end{aligned}$$

So our initial tableau is

$$\begin{pmatrix} 3 & 2 & 1 & 1 & 0 & 0 & 10 \\ 2 & 5 & 3 & 0 & 1 & 0 & 15 \\ -2 & -3 & -4 & 0 & 0 & 1 & 0 \end{pmatrix}$$

We now perform a pivot operation:

(**1**) we first choose the most negative value in the bottom row: -3, this will be our column.

(**2**) we then compute b_j/A_{ji} and take the row with the minimum value. The values are $10/3$ and $15/2$, so we choose 2 as the pivot.

Row reducing will give

$$\begin{pmatrix} 0 & -5.5 & -3.5 & 1 & -1.5 & 0 & -12.5 \\ 1 & 2.5 & 1.5 & 0 & 0.5 & 0 & 7.5 \\ 0 & 2 & -1 & 0 & 1 & 1 & 15 \end{pmatrix}$$

The next pivot column will be the third column, and computing the quotients gives $-12.5/-3.5 = 25/7$ and $7.5/1.5 = 5$ so we choose the second row as the pivot column. The resulting tableau is

$$\begin{pmatrix} 0 & -5.5 & -3.5 & 1 & -1.5 & 0 & -12.5 \\ 2/3 & 5/3 & 1 & 0 & 1/3 & 0 & 5 \\ 2/3 & 11/3 & 0 & 0 & 4/3 & 1 & 20 \end{pmatrix}$$

The bottom row is all positive, so we can no longer pivot. Thus the maximum value is 20. ◊

1.4 The Simplex Algorithm; Phase I

So far what we discussed relies on us having an initial bfs to our problem, otherwise our initial tableau is not in canonical form (and then we have to find a basis for which the bfs is non-negative). Instead of iterating over each basis, we can instead solve an LP to find an initial bfs.

Suppose we want to maximize $c^\top x$ constrained to $Ax = b$. Now instead let us minimize $1^\top s$ (1 is the vector containing all ones) constrained to $(A \mid I)\binom{x}{s} = b$ as well as $-c^\top x + w = 0$ for some other variable w (this is in order to keep track of the changes to $c^\top x$ as we row reduce). This is equivalent to $Ax + s = b$, i.e. $Ax \leq b$, and the idea is that if we can make $s = 0$ then the resulting bfs will be a vertex in the polytope $Ax = b$. So we minimize $1^\top s$ (equivalently, maximize $-1^\top s$) and if we get $s = 0$ then we have an initial bfs. Otherwise, we can't minimize s to 0 and so we have no solution to $Ax = b$, meaning our polytope is empty.

This procedure is the first step in the simplex algorithm, also called *Phase I*. The second step (row reducing the tableau) is called *Phase II*.

Example: suppose we want to maximize

$$2x + 3y + 4z$$

subject to

$$\begin{aligned} 3x + 2y + z &= 10 \\ 2x + 5y + 3z &= 15 \\ x, y, z &\geq 0 \end{aligned}$$

So our initial tableau is

$$\begin{pmatrix} 3 & 2 & 1 & 0 & 10 \\ 2 & 5 & 3 & 0 & 15 \\ -2 & -3 & -4 & 1 & 0 \end{pmatrix}$$

which is not in canonical form. So now according to phase I, we want to maximize $-u - v$ subject to

$$\begin{aligned} 3x + 2y + z + u &= 10 \\ 2x + 5y + 3z + v &= 15 \\ -2x - 3y - 4z + w &= 0 \\ x, y, z, u, v &\geq 0 \end{aligned}$$

The initial tableau for this is

$$\begin{pmatrix} 3 & 2 & 1 & 0 & 1 & 0 & 0 & 10 \\ 2 & 5 & 3 & 0 & 0 & 1 & 0 & 15 \\ -2 & -3 & -4 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Converting to its canonical form, we get

$$\begin{pmatrix} 3 & 2 & 1 & 0 & 1 & 0 & 0 & 10 \\ 2 & 5 & 3 & 0 & 0 & 1 & 0 & 15 \\ -2 & -3 & -4 & 1 & 0 & 0 & 0 & 0 \\ -5 & -7 & -4 & 0 & 0 & 0 & 1 & -25 \end{pmatrix}$$

We could take the second column as the pivot column, but let's choose column 3 instead. We then perform the simplex algorithm on this to give an initial tableau. ◊

If some of the constraints have slack variables, we can optimize this so that we only add artificial variables to constraints without them. This way we still start with a canonical form, and we have fewer artififical variables to deal with.

Now suppose we have the linear program of maximizing $c^\top x$ subject to $Ax = b$ where b is not non-negative. Let us assume that A is in canonical form, then simply converting $R_i(A)x = b_i$ to $-R_i(A)x = -b_i$ will not work, as it will affect the canonicalness of A. For every constraint where b_i is negative, we substract some artificial variable a_0 from the constraint, so it becomes $\sum_j a_{ij}x_i - a_0 = b_i$, and we want to minimize a_0, i.e. maximize $-a_0$. We then add a_0 to the initial basis, we do this to make $\bar{b}$ nonnegative. This is because if we pivot on the ith row, then this means that b_i/A_{ij} (where j is the row corresponding to a_0) is maximal, but $A_{ij} = -1$ so $-b_i$ is maximal. This will set b_i to $-b_i$ (since remember the coefficient is -1), and for every other b_j which is negative (so a_0's coefficient is -1), this becomes $b_j - b_i \geq 0$.

This makes more sense when our constraint is $a_1x_1 + \cdots + a_nx_n \leq -b$, we add a slack variable to become $a_1x_1 + \cdots + a_nx_n + s = -b$. But then setting $x_i = 0$ gives us an infeasible solution $s = -b < 0$. So we add an artificial variable a, $a_1x_1 + \cdots + a_nx_n + s - a = -b$, then our goal is to now maximize $-a$ so that it becomes zero.

Example: we want to maximize $5x_1 - x_2 - x_3$ subject to

$$\begin{aligned} 3x_1 - x_2 - x_3 &\leq -1 \\ x_1 + 2x_2 - x_3 &\leq -2 \\ 2x_1 + x_2 &\leq 2 \\ x_1 + x_2 &= 1 \\ x_1, x_2, x_3 &\leq 0 \end{aligned}$$

We need two artificial variables: one for the equality constraint, and one for the negative constraints. We also add slack variables. So our problem becomes maximizing $-a_1 - a_2$ subject to

$$\begin{aligned} 3x_1 - x_2 - x_3 + s_1 - a_1 &= -1 \\ x_1 + 2x_2 - x_3 + s_2 - a_1 &= -2 \\ 2x_1 + x_2 + s_3 &= 2 \\ x_1 + x_2 + a_2 &= 1 \\ -5x_1 + x_2 + x_3 + z &= 0 \end{aligned}$$

So the initial tableau is

$$\left(\begin{array}{cccccccccccc} x_1 & x_2 & x_3 & s_1 & s_2 & s_3 & z & a_1 & a_2 & w & b \\ \hline 3 & -1 & -1 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & -1 \\ 1 & 2 & -1 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & -2 \\ 2 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ -5 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{array}\right)$$

To make this canonical, we must adjust the objective row so that a_2 is in the basis:

$$\left(\begin{array}{cccccccccccc} x_1 & x_2 & x_3 & s_1 & s_2 & s_3 & z & a_1 & a_2 & w & b \\ \hline 3 & -1 & -1 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & -1 \\ 1 & 2 & -1 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & -2 \\ 2 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ -5 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline -1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -1 \end{array}\right)$$

Now we pivot with respect to a_0, which enters, and s_2 leaves. We then continue normally until we get

$$\left(\begin{array}{cccccccccccc} x_1 & x_2 & x_3 & s_1 & s_2 & s_3 & z & a_1 & a_2 & w & b \\ \hline 1 & 0 & 0 & 1/5 & -1/5 & 0 & 0 & 3/5 & 0 & 0 & 4/5 \\ 0 & 0 & 1 & -1/5 & -4/5 & 0 & 0 & 7/5 & 1 & 0 & 16/5 \\ 0 & 0 & 0 & -1/5 & 1/5 & 1 & 0 & -8/5 & 0 & 0 & 1/5 \\ 0 & 1 & 0 & -1/5 & 1/5 & 0 & 0 & 2/5 & 0 & 0 & 1/5 \\ 0 & 0 & 0 & 7/5 & -2/5 & 0 & 1 & 6/5 & -1 & 0 & 3/5 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{array}\right)$$

Phase I is thus successful since we have maximized to 0. We can then perform phase II on the canonical tableau

$$\left(\begin{array}{cccccccc} x_1 & x_2 & x_3 & s_1 & s_2 & s_3 & z & b \\ \hline 1 & 0 & 0 & 1/5 & -1/5 & 0 & 0 & 4/5 \\ 0 & 0 & 1 & -1/5 & -4/5 & 0 & 0 & 16/5 \\ 0 & 0 & 0 & -1/5 & 1/5 & 1 & 0 & 1/5 \\ 0 & 1 & 0 & -1/5 & 1/5 & 0 & 0 & 1/5 \\ \hline 0 & 0 & 0 & 7/5 & -2/5 & 0 & 1 & 3/5 \end{array}\right)$$

Now s_2 enters and s_3 leaves since the most negative value in the bottom row is $-2/5$ and the largest quotient is 1. This gives

$$\left(\begin{array}{cccccccc} x_1 & x_2 & x_3 & s_1 & s_2 & s_3 & z & b \\ \hline 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & -1 & 0 & 4 & 0 & 4 \\ 0 & 0 & 0 & -1 & 1 & 5 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 & 2 & 1 & 1 \end{array}\right)$$

So the maximum value is 1 with $x_1 = 1$, $x_2 = 0$ and $x_3 = 4$. ◊

1.5 Duality

1.5.1 Definition

Call an LP problem of the form: maximize $c^\top x$ such that $Ax \leq b, x \geq 0$ **primal**. Its **dual** is: minimize $b^\top y$ such that $A^\top y \geq c, y \geq 0$.

1.5.2 Lemma (The Weak Duality Theorem)

If x is a feasible solution to the primal LP and y to the dual LP, then $c^\top x \leq b^\top y$.

Proof: this is just a chain of inequalities:

$$c^\top x = x^\top c \leq x^\top (A^\top y) = (x^\top A^\top) y = (Ax)^\top y \leq b^\top y$$ ■

1.5.3 Theorem (The Strong Duality Theorem)

If x is a maximal solution to the primal LP, and y a minimal solution to the dual LP, then $c^\top x = b^\top y$.

Proof: let us show that if x is a maximal solution to the primal problem, then there exists a solution y to the dual problem such that $c^\top x = b^\top y$. By the weak duality theorem, this is sufficient.

Suppose first that our problem is in slack form: $Ax = b$. Let x be a maximal solution, and its corresponding basis is B. Suppose that $A = A_B \,|_B\, A_N$ and $c^\top = c_B^\top \,|_B\, c_N^\top$, then our initial tableau, multiplied by $A_B^{-1} \oplus (1)$ is

$$\begin{pmatrix} A_B^{-1} & \\ & 1 \end{pmatrix} \cdot \begin{pmatrix} A & 0 & b \\ -c^\top & 1 & 0 \end{pmatrix} = \begin{pmatrix} I & |_B & A_B^{-1}A_N & 0 & A_B^{-1}b \\ -c_B^\top & |_B & -c_N^\top & 1 & 0 \end{pmatrix}$$

Row reducing gives us

$$\begin{pmatrix} I & |_B & A_B^{-1}A_N & 0 & A_B^{-1}b \\ 0^\top & |_B & -c_N^\top + c_B^\top A_B^{-1} A_N & 1 & z_B \end{pmatrix}$$

Since x is optimal, the bottom row must be positive, so

$$c_N^\top - c_B^\top A_B^{-1} A_N \leq 0$$

but trivially, we also know (since the following is equal to 0)

$$c_B^\top - c_B^\top A_B^{-1} A_B \leq 0$$

Thus by concatenating them with their order determined by B,

$$c^\top - c_B^\top A_B^{-1} A \leq 0$$

Now, our problem was $Ax \leq b$, which is equivalent to $(A \mid I)\binom{x}{s} = b$ by introducing slack variables. And the objective function is $(c^\top \mid 0)\binom{x}{s}$, so by our above computation (by splitting according to which indexes are slack variables), we get

$$\begin{aligned} \text{For non-slack variables:} \quad & c^\top - c_B^\top \bar{A}_B^{-1} A \leq 0 \\ \text{For slack variables:} \quad & 0^\top - c_B^\top \bar{A}_B^{-1} I \leq 0 \end{aligned}$$

where $\bar{A} = A \mid I$. So let us define $y^\top = c_B^\top \bar{A}_B^{-1}$, then we have that

$$c^\top - c_B^\top \bar{A}_B^{-1} A = c^\top - y^\top A \leq 0 \implies y^\top A \geq c^\top$$

and $y^\top \geq 0$. So y is a solution to the dual LP problem. Furthermore

$$b^\top y = y^\top b = c_B^\top \bar{A}_B^{-1} b = c_B^\top x_B = c^\top x$$

as required. ■

2 Approximation Algorithms

2.1 Optimization Complexity Classes

2.1.1 Definition

An **optimization problem** is a tuple $\mathcal{O} = (I, \mathsf{Sol}, m, \mathsf{type})$, where

(**1**) I is a set called the **instance set**, whose elements are instances of the problem.

(**2**) Sol is a function such that for every instance $i \in I$, $\mathsf{Sol}(i)$ is the set of all feasible solutions to the instance i.

(**3**) m is a function such that for every instance $i \in I$ and solution $s \in \mathsf{Sol}(i)$, $m(i, s)$ is the **measure** of the solution.

(**4**) $\mathsf{type} \in \{\max, \min\}$ which tells us if we want to maximize or minimize the measure of a solution.

Then the **optimal measure** of an instance $i \in I$ is

$$\mathrm{opt}(i) = \underset{s \in \mathsf{Sol}(i)}{\mathsf{type}}\ m(i, s)$$

For example, for the problem of finding the shortest path between two vertices s and t can be described as such:

- $I = \{(G = (V, E), s, t) \mid s, t \in V\}$
- $\mathsf{Sol}(G, s, t) = \{P \mid P \text{ is a path from } s \text{ to } t \text{ in } G\}$
- $m((G, s, t), P) = |P|$
- $\mathsf{type} = \min$

Given an optimization problem $\mathcal{O}$, its corresponding decision problem $\mathcal{D}_\mathcal{O}$ is the problem of determining if given a $k \in \mathbb{R}$ and instance $i \in I$, whether there exists a solution $s \in \mathsf{Sol}(i)$ such that $m(i, s) \geq k$ for $\mathsf{type} = \max$ and $m(i, s) \leq k$ for $\mathsf{type} = \min$.

2.1.2 Definition

The complexity class **NPO** (**NP** optimization) is the class of all optimization problems $\mathcal{O}$ such that its corresponding decision problem $\mathcal{D}_\mathcal{O}$ is in **NP**.

Call an optimization problem $\mathcal{O}$ **NPO**-complete if $\mathcal{D}_\mathcal{O}$ is **NP**-complete.

2.1.3 Definition

An **NPO** problem $\mathcal{O}$ is in **PO** (polynomial optimization) if there is a deterministic polynomial-time algorithm to compute $\mathrm{opt}(i)$.

Since NPO-complete problems are NPO-complete, we currently (and maybe always) do not have an efficient method of solving them. So the next best thing is approximating solutions.

2.2 Vertex Cover

2.2.1 Problem

Given a graph $G = (V, E)$, a set $S \subseteq V$ is a **vertex cover** if for every edge $\{u, v\}$, either $u \in S$ or $v \in S$. The optimization problem of vertex covers is to find the size of the minimum vertex cover of an input

graph.

As you should know, the decision problem is NP-complete. So the optimization problem is NPO-complete.

We can come up with a naive algorithm to solve this:

```
1. function VERTEX-COVER(G = (V, E))
2.     A ← ∅
3.     while (there exists an edge {u, v} st u, v ∉ A)
4.         A ← A ∪ {u, v}
5.     end while
6.     return A
7. end function
```

2.2.2 Theorem

The above algorithm is a 2-approximation, that is A is a vertex cover and $|A| \leq 2 \cdot |\text{Opt}|$.

Proof: let M be the set of edges chosen in the algorithm. Then Opt must include a vertex from each edge in M as it is a vertex cover, so $|M| \leq |\text{Opt}|$. Furthermore, $|A| = 2|M|$, so

$$|A| = 2|M| \leq 2|\text{Opt}|$$

as required. ■

This analysis is tight: there exists a graph such that $|A| = 2|\text{Opt}|$.

2.2.3 Definition

APX (approximation) is the family of all approximation problems $\mathcal{O} \in$ **NPO** such that for some constant $c \geq 0$, there is a deterministic polynomial-time algorithm such that for every $i \in I$, it finds a solution $s \in \mathsf{Sol}$ such that

(**1**) If type = min then $c \geq 1$ and $m(i, s) \leq c\,\text{opt}(i)$.

(**2**) If type = max then $c \leq 1$ and $m(i, s) \geq c\,\text{opt}(i)$.

So Vertex-Cover is in **APX**.

2.3 Max Cut

2.3.1 Problem

Given a graph $G = (V, E)$ the problem of **max cut** is the problem of finding a cut $S \subseteq V$ that maximizes the number of edges in the cut

$$E(S) = E(S, V \setminus S) = \{\{u, v\} \in E \mid u \in S, v \notin S\}$$

The decision problem of max cut is NP-complete, so max cut is NPO-complete. We describe an algorithm to approximate max cut:

```
1. function MAX-CUT(G = (V, E))
2.     S ← ∅
3.     while (true)
4.         if (there exists a v ∉ S such that |E(S ∪ {v})| > |E(S)|)
5.             S ← S ∪ {v}
6.         else if (there exists a v ∈ S such that |E(S \ {v})| > |E(S)|)
7.             S ← S \ {v}
8.         else
9.             return S
```

10. **end if**
11. **end while**
12. **end function**

There are at most $\binom{n}{2}$ edges in the graph, and thus the maximum possible size of $E(S)$ is $\binom{n}{2}$. Notice that at each iteration, $|E(S)|$ increases by at least 1, and so there can be at most $\binom{n}{2}$ iterations, so this algorithm is polynomial-time.

Notice that if the algorithm returns S, then for $v \in S$, $|E(\{v\}, V \setminus S)| \geq \frac{\deg v}{2}$ since otherwise more than half of v's neighbors are in S and then the algorithm would've placed v in $V \setminus S$. Similarly for $v \notin S$, $|E(\{v\}, S)| \geq \frac{\deg v}{2}$. Thus

$$|E(S, V \setminus S)| \geq \frac{1}{2} \sum_{v \in V} \frac{\deg v}{2} = \frac{|E|}{2} \geq \frac{\text{opt}}{2}$$

This is because when we sum over $v \in V$, we count how many edges are in the cut, but since we count both $v \in S$ and $v \notin S$, we double count. So we must divide by 2.

Thus this algorithm is a $\frac{1}{2}$-approximation, meaning max cut is in APX.

2.4 Minimum Makespan Scheduling

2.4.1 Definition

The problem of minimum makespan scheduling is as follows: given n jobs and the time it takes to complete each one: $j_1, \ldots, j_n$, as well as the amount m of identical machines, find an allocation of the jobs to the machines such to minimize the total time spent (also called the makespan).

This is an **NP**-hard problem, but we can come up with a greedy algorithm:

(**1**) Order the jobs arbitrarily.

(**2**) For each job, allocate it to the machine which has the least amount of work currently.

Now consider an optimal solution opt, notice that each machine must work for at most opt time, and so $\sum_{i=1}^{n} j_i \leq m \,\text{opt}$, or equivalently $\frac{\sum_{i=1}^{n} j_i}{m} \leq \text{opt}$. Furthermore, trivially, $\max_i j_i \leq \text{opt}$.

2.4.2 Theorem

The greedy algorithm is a $2 - \frac{1}{m}$-approximation to the minimum makespan scheduling problem.

Proof: suppose machine q takes the maximum amount of time in our algorithm, suppose it takes $t_{\max}$ time. Let j_s be the last job assigned to q at time t_s, so at time t_s the rest of the machines were computing and so

$$t_s \cdot m + j_s \leq \sum_i j_i \leq m \,\text{opt}$$

and thus $j_s \leq \text{opt}$ and $t_s + \frac{j_s}{m} \leq \text{opt}$, so

$$\mathsf{alg} = t_s + j_s = t_s + \frac{j_s}{m} + \left(1 - \frac{1}{m}\right) j_s \leq \text{opt} + \left(1 - \frac{1}{m}\right) \text{opt} = \left(2 - \frac{1}{m}\right) \text{opt}$$ ■

2.4.3 Theorem

If we alter the greedy algorithm slightly so that it orders the jobs in decreasing order first, then it gives a $\frac{3}{2}$-approximation for the minimum makespan scheduling problem.

Proof: let $j_1 \geq \cdots \geq j_n$ be the jobs in decreasing order, and let $J_L = (j_1, \ldots, j_k)$ be the jobs longer than $\frac{\text{opt}}{2}$. Call these jobs *large*. Notice that $k \leq m$, as otherwise some machine would get two large jobs, and this would take greater than opt time. So suppose machine q has the maximum completion time in the algorithm, and let j_s be the last job allocated to q at time t_s. If $s \leq k$ then j_s is the only job of q so $\mathsf{alg} = t_s \leq j_s \leq \text{opt}$. So we can assume $s > k$. Since at t_s all other machines are occupied,

$$t_s \cdot m \leq t_s \cdot m + j_s \leq \sum_i j_i \leq m \,\text{opt}$$

thus

$$\mathsf{alg} = t_s + j_s \leq \text{opt} + \frac{\text{opt}}{2} = \frac{3}{2} \,\text{opt}$$ ■

2.4.4 Definition

(**1**) We say that a minimization problem admits a **polynomial-time approximation scheme** (PTAS) if for every $\varepsilon > 0$, there is a $1+\varepsilon$ approximation in polynomial time.

(**2**) A problem which admits a PTAS in $poly(n) \cdot f(1/\varepsilon)$ time (for f arbitrary) is said to admit a **efficient polynomial-time approximation scheme** (EPTAS).

(**3**) A problem which admits a PTAS in $poly(n + 1/\varepsilon)$ time is said to admit a **fast polynomial-time approximation scheme** (FPTAS).

2.4.5 Theorem

The minimum makespan scheduling problem admits a PTAS.

2.5 Bin Packing Problem

2.5.1 Definition

The problem of **bin packing** can be stated as follows: given a finite set $U = \{\mu_1, \ldots, \mu_n\}$ of objects with rational size $\mathsf{s}(\mu_i) \in [0, 1]$, return a partition of U into subsets $U_1, \ldots, U_k$ such that for every $1 \leq j \leq k$:

$$\sum_{\mu_i \in U_j} \mathsf{s}(\mu_i) \leq 1$$

The goal is to minimize k.

2.5.2 Definition

The problem of **partition** is as follows: given a multiset of numbers $S = \{x_1, \ldots, x_n\}$, determine if one can partition S into $S_1 \uplus S_2$ such that $\sum_{x \in S_1} x = \sum_{x \in S_2} x$.

Parition is **NP**-hard.

2.5.3 Theorem

Unless $\mathbf{P} = \mathbf{NP}$, for every $\varepsilon > 0$ there is no $\frac{3}{2} - \varepsilon$-approximation for bin packing.

Proof: suppose for some $\varepsilon > 0$ there is a $\frac{3}{2} - \varepsilon$-approximation for bin packing, we will use it to solve partition. Let $S = \{x_1, \ldots, x_n\}$ be an instance of partition, let $\mu = \sum_{x \in S} x$, then let $S' = \left\{\frac{2}{\mu}x_1, \ldots, \frac{2}{\mu}x_n\right\}$ be an input to the bin packing problem. Using our $\frac{3}{2} - \varepsilon$-approximation, find a solution $U_1, \ldots, U_k$ to the bin packing problem. Return `true` if $k = 2$ and otherwise `false`.
Suppose that the true answer is `true`, then $S = S_1 \uplus S_2$ and $\sum_{x \in S_1} x = \sum_{x \in S_2} x = \frac{\mu}{2}$. Then $U_i = \left\{\frac{2}{\mu}x \;\middle|\; x \in S_i\right\}$ is a solution to bin packing since

$$\sum_{x \in S_i} \frac{2}{\mu}x = \frac{2}{\mu} \cdot \frac{\mu}{2} = 1$$

The approximation algorithm of bin packing must return a solution of at most size $(3/2 - \varepsilon) \cdot 2 < 3$, so 2. Thus if the answer is `true`, then our algorithm will return `true`.
Now suppose the algorithm returned `true`, then since the total weight of S' is 2, there must be two bins of size 1, the corresponding partition of S is a valid solution to partition. ■

We can develop an algorithm for bin packing, called first fit, as follows: for every object μ_i place it in the first bin in which it fits. If no bin is available, open a new one. Denote the number of bins used by the algorithm on an instance I by $\mathrm{FF}(I)$.

2.5.4 Lemma

$\mathrm{FF}(I) \leq 2|\mathrm{opt}| + 1$

Proof: first notice that if opt uses bins $B_1, \ldots, B_n$ then

$$\sum_i \mathsf{s}(\mu_i) = \sum_i \mathsf{s}(B_i) \leq \sum_i 1 = |\mathrm{opt}|$$

Now suppose $\mathrm{FF}(I)$ uses bins $B_1, \ldots, B_m$, then there is a single bin of weight $\leq \frac{1}{2}$ as otherwise the algorithm would've merged the two bins. Thus

$$|\mathrm{opt}| \geq \sum_i \mathsf{s}(B_i) \geq \frac{1}{2}(m-1)$$

so $m \leq 2|\mathrm{opt}| + 1$, as required. ■

2.5.5 Theorem

There is a PTAS for bin packing. In other words, for every $\varepsilon > 0$, there is a poly-time algorithm which solves the bin-packing problem using at most $(1+\varepsilon)|\mathrm{opt}| + 1$ bins.

To prove this, we will utilize the following lemmas:

2.5.6 Lemma

Suppose that all the elements are of size at most δ, then $\mathrm{FF}(I) \leq (1+2\delta)|\mathrm{opt}| + 1$.

Proof: if $\delta \geq \frac{1}{2}$ then since $\mathrm{FF}(I) \leq 2|\mathrm{opt}| + 1 \leq (1+2\delta)|\mathrm{opt}| + 1$ as required. So let us assume $\delta < \frac{1}{2}$, then all the bins besides the last have at least $1-\delta$ weight. This is since if the weight of a bin is $\leq 1-\delta$ then it can take another object. Thus

$$(\mathrm{FF}(I)-1)(1-\delta) < \sum_i \mathsf{s}(\mu_i) \leq |\mathrm{opt}| \implies \mathrm{FF}(I) \leq \frac{|\mathrm{opt}|}{1-\delta} + 1 \leq (1+2\delta)|\mathrm{opt}| + 1$$ ■

2.5.7 Lemma

Suppose that there are only k distinct weights, then there is an algorithm to find the optimal solution in $O(n^{2k+1})$ time.

Proof: let A be the set of the different weights. Then let S be the set of all multi-sets with n values from A, then by definition

$$|S| = \left(\!\binom{k}{n}\!\right) = \binom{n+k-1}{n} = \frac{(n+k-1)!}{n!(k-1)!} = \frac{(n+1)\cdots(n+k-1)}{1\cdots(k-1)}$$

This is a polynomial of degree $k-1$, so it is less than (asymptotically) n^k. Let $S_1 \subseteq S$ be all multisets which can be packed into a single bin, then S_1 can be computed in $O(n^k)$ time. Let $S_{i-1} \subseteq S$ be all the multisets which be packed into $i-1$ bins. Then

$$S_i \subseteq \{B \cup C \mid B \in S_{i-1}, C \in S_1\}$$

So we can simply iterate over all pairs in $S_{i-1} \times S_1$ and check if their union is in S_i. There are $O(n^{2k})$ such pairs, so computing S_i takes $O(n^{2k})$ time. Once $\{\mathsf{s}(\mu_j) \mid \mu_j \in U\}$ is in S_i, then we know we can pack all the objects into at least i bins. So we stop and return i. And since all the objects can be trivially packed into n bins, we stop at n iterations at most. So the total runtime is $O(n^{2k+1})$. ■

2.5.8 Lemma

Suppose that all the items are of size $\geq \delta$. Then there is an algorithm producing a solution with $(1+\delta)|\text{opt}|+1$ bins in runtime $O(n^{2/\delta^2}+1)$.

Proof: fix some k, which will be determined later. Then sort U in nonincreasing order (largest elements first), and partition U into $\lceil \frac{n}{k} \rceil$ subsets of size k (the final subset may have fewer than k elements). Let these subsets be $G_1, \ldots, G_{\lceil n/k \rceil}$. Create a new instance U' where we first discard G_1, then for $2 \leq i \leq \lceil \frac{n}{k} \rceil$ we change all the values in the set G_i to the maximal value in G_i. Let these new sets be $G'_2, \ldots, G'_{\lceil n/k \rceil}$. Notice that we can map the elements of G'_{i+1} to G_i so that w' is mapped to a unique $w \geq w'$, let this mapping be f. This means that $|\text{opt}(U')| \leq |\text{opt}(U)|$ since if we put U into bins $B_1, \ldots, B_m$ then $f^{-1}B_1, \ldots, f^{-1}B_m$ (excuse the abuse of notation) are valid bins for U' (since $f^{-1}w \leq w$).

Now, there are $\lceil \frac{n}{k} \rceil - 1 \leq \frac{n}{k}$ distinct values in U', so by the previous lemma, we can compute $|\text{opt}(U')|$ in $O(n^{2\frac{n}{k}+1})$ time.

So our algorithm will work as follows: put every element in G_1 into its own bin. Then find an optimal solution for U'. Since U' is greater than $G_2, \ldots, G_{\lceil n/k \rceil}$ use this optimal solution to pack $G_2, \ldots, G_{\lceil n/k \rceil}$. This uses

$$|\text{alg}(U)| = |\text{opt}(U')| + k \leq |\text{opt}(U)| + k$$

Now let

$$k = \left\lceil \delta \cdot \sum_{\mu_i \in U} \mathsf{s}(\mu_i) \right\rceil$$

Since $|\text{opt}| \geq \sum_\mu \mathsf{s}(\mu)$, we have that $k \leq \lceil \delta |\text{opt}| \rceil \leq \delta|\text{opt}| + 1$. So

$$|\text{alg}(U)| = |\text{opt}(U')| + k \leq |\text{opt}(U)| + k \leq (1+\delta)|\text{opt}(U)| + 1$$

and the runtime is

$$O(n^{2n/k+1}) = O\left(n^{2 \cdot \frac{n}{\delta \sum_\mu \mathsf{s}(\mu)}+1}\right) \leq O\left(n^{2 \cdot \frac{n}{\delta \sum_\mu \delta}+1}\right) = O\left(n^{2/\delta^2+1}\right)$$

as required. ■

2.5.9 Theorem

For every $\varepsilon > 0$ there is a polynomial-time algorithm returning a solution to the bin-packing probkem using at most $(1+\varepsilon)|\text{opt}|+1$ bins in $O(n^{8/\varepsilon^2+1})$ time.

Proof: let U_L be the set of all objects of weight $\geq \varepsilon/2$, and let $U_S = U \setminus U_L$. We run the algorithm of lemma 2.5.8 on U_L, and then use first-fit on U_S to pack it into the remaining space of the U_L bins. If there is no more space, use first-fit on new bins. If no new bins were opened using first-fit then

$$|\text{alg}| = |\text{alg}_L(U_L)| \leq \left(1+\frac{\varepsilon}{2}\right)\text{opt}(U_L) + 1 \leq \left(1+\frac{\varepsilon}{2}\right)\text{opt}(U) + 1$$

Otherwise, notice that if we ended up with m bins, then all but 1 of them must end up with a weight of at least $1-\varepsilon/2$ (since we can't pack any small items into it). Thus $(m-1)(1-\varepsilon/2) < \sum_\mu \mathsf{s}(\mu) \leq \text{opt}$. Thus

$$m \leq \frac{1}{1-\varepsilon/2}\,\text{opt} + 1 \leq (1+\varepsilon)\,\text{opt} + 1$$

as required. ■

3 Pattern Matching

We will focus on the following problem: given a text $T[1,\dots,n]$, we would like to find the first occurrence of a pattern $P[1,\dots,m]$. That is, we want to find the first index i such that

$$T[i,\dots,i+m-1] = P[1,\dots,m]$$

A common way of doing this is constructing an automaton. This is done as follows: we define a *suffix function* $\sigma\colon \Sigma^* \longrightarrow \{0,1,\dots,m\}$ for which given a string w, $\sigma(w)$ is the (index of the end of the) longest prefix of P which is also a suffix of w. Formally

$$\sigma(w) = \max\{k \mid P[1,\dots,k] \sqsupseteq w\}$$

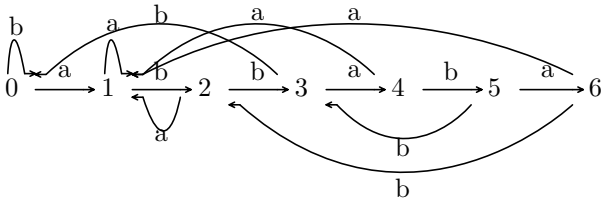
where $w_1 \sqsubseteq w_2$ means that w_2 is a suffix of w_1. Now we define the automaton as follows:

(**1**) The state space is $Q = \{0,1,\dots,m\}$. The initial state is $q_0 = 0$ and the single accepting state is m.

(**2**) The transition function is $\delta(q,a) = \sigma(P[1,\dots,q]a)$.

The idea is as follows: if we are on state q, then we have matched $P[1,\dots,q]$. Now if the next character is a, then is $a = P[q+1]$, great: $\delta(q,a) = q+1$. Otherwise, the current text we have matched looks like: $T[1,\dots,N] = \dots P[1,\dots,q]a$, and we know that we have not found P yet. So we want to find the maximum index k such that $P[1,\dots,k]$ is a suffix of $P[1,\dots,q]a$, since then our text looks like $T[1,\dots,N] = \dots P[1,\dots,k]$ and we can continue the automaton from state k.

So for example, given $\Sigma = \{a,b\}$ and $P =$ abbaba, our atomaton looks like:



Now suppose that we add another character to our alphabet, then we need to add new transitions for each node. Each node requires $|\Sigma|$ transitions, so the size of our automaton is $\Theta(m|\Sigma|)$ and thus the build time is $\Omega(m|\Sigma|)$. This is not ideal for large alphabets.

3.1 Knuth-Morris-Pratt (KMP) Algorithm

A goal of the KMP algorithm is to prevent computing δ in its entirety altogether. Suppose that the text is $T = \dots P[1,\dots,q]$ and then the next character is different from $P[q+1]$. Then we want to find the maximum k such that $P[1,\dots,q] = \dots P[1,\dots,k]$, as then we can check if the next character is $P[k+1]$. So let us define

$$\pi(q) = \max\{k < q \mid P[1,\dots,k] \sqsupseteq P[1,\dots,q]\}$$

So our algorithm will function as follows: if we miss a match at state q, go to state $\pi(q)$ and try again:

```
1. function KMP-Matcher(T, P, π)
2.     q = 0
3.     for (i = 0 to n)
4.         while (q > 0 and P[q+1] ≠ T[i])  q = π(q)    ▷ If miss match
5.         if (P[q+1] = T[i])  q = q+1
6.         if (q = m)  return i − m
7.     end for
8. end function
```

Now the auxiliary space is $\Theta(m)$ as opposed to $\Theta(m|\Sigma|)$ and fortunately the matching time by KMP is still $\Theta(n)$.

3.2 Karp-Rabin Algorithm

Suppose we have a pattern $p_0p_1\dots p_{m-1}$, then we can choose a random r and convert this into $\hat{P} = p_0r^{m-1} + p_1r^{m-2} + \dots + p_{m-1} \bmod q$ for some prime q (since $\mathbb{F}_q$ is a field). Given a text $t_0t_1\dots t_{n-1}$, we can compute $S_i = t_ir^{m-1} + t_{i+1}r^{m-2} + \dots + t_{i+m-1} \bmod q$. Then if $\hat{P} \neq S_i$, then $T[i : i+m-1] \neq P$. If $\hat{P} = S_i$ then we can report that there is a match; this is a randomized/nondeterministic algorithm and so it is allowed false positives.

But computing S_i is costly, how can we optimize this. Well, notice that

$$S_{i+1} = rS_i + t_{i+m} - t_i r^m \bmod q$$

So once we compute S_0, computing each subsequent S_i takes $O(1)$ time.

Now, what is the probability of a false positive? Let us define the following two polynomials:

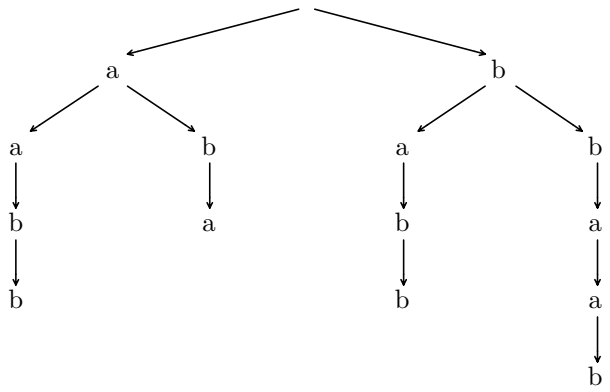
$$p_P(x) = p_0 x^{m-1} + p_1 x^{m-2} + \cdots + p_{m-1} \bmod q, \qquad p_{T,i}(x) = t_i x^{m-1} + t_{i+1} x^{m-2} + \cdots + t_{i+m-1} \bmod q$$

we have that $\hat{P} = S_i$ when $p_P(r) = p_{T,i}(r)$, i.e. r is a root of $H(x) = p_P(x) - p_{T,i}(x)$. This is a root $m-1$ polynomial, and thus has at most $m-1$ roots. Choosing r uniformly from $\mathbb{F}_q$ gives us that the probability of choosing an r which gives a false positive at an index i is $\frac{m-1}{q}$.

3.3 Dictionary Matching

The problem of dictionary matching, is that given a *dictionary* (list) of patterns, we want to find all instances of any pattern in the dictionary in an input text. One solution is to just look for each pattern in the dictionary, but we can do better.

Let us familiarize ourself with a datastructure called a *trie*. Given a dictionary $D = \{d^1, \ldots, d^n\}$ we create a tree T whose edges are labeled with characters. Then suppose that we have a node q, and suppose to get to the node we must pass through edges $\omega = \sigma_1 \cdots \sigma_k$. Then for every $\sigma \in \Sigma$ where $\omega\sigma$ is a prefix of a word in the dictionary, we add a new node and a connecting edge labelled σ. For example if $D = \{\text{aba}, \text{aabb}, \text{babb}, \text{bbaab}\}$ then the trie would be (edges's labels are below them):



Now if we want to check if a word $\sigma_1 \cdots \sigma_k$ is in the dictionary, we simply follow the edges.

If we store only the relevant characters in each edge, this will take $O(|D|)$ storage ($|D|$ is the size of D, i.e. $\sum_i |d_i|$). If $|\Sigma|$ is constant or we use a hash table to locate the edges, search time is $O(|query|)$. Otherwise we need to perform a binary search to locate each pointer, which takes $O(|query| \log|\Sigma|)$ time.

3.4 Indexing

Suppose we have a text T and a pattern P, and we want to find all the *indices* where P is in T. What we can do is construct a dictionary $D = \{\omega\$ \mid \omega \text{ is a prefix of } T\}$ and construct a trie from D. At each childless node, we put the index of the suffix which formed it (so if $\sigma_i \cdots \sigma_{n-1}$ is a prefix, it ends on a node q which will be given the number i). Then, when we want to find if a pattern P is in T, we simply query T and this may give us a node q. We follow its children until we get to indices, and we return these. This tree is called the *suffix tree*.

For example $T = \text{ababb}$. If we query $P = \text{ab}$, following it down the trie gives us a node which has two children, following these children give us the indices $0, 2$.

The issue is the query time here can be very long, and the size is $O(n^2)$. So we can compress the trie as follows: for every path in the trie (i.e. each node has a single child), we compress it to a single node where we store the substring in the path as well as the indices which start and end the substring. This still takes $O(n^2)$ space (as we still must store the substrings).

3.5 k-Mismatches

3.5.1 LCP

Suppose we have a pattern P and a text T, and we want to find all the indices which match with T up to k places (i.e. all indices i such that $\#\{j \mid P[j] \neq T[i+j]\} \leq k$).

To do so we will use a suffix tree with LCA (least common ancestor), i.e. a datastructure where for any two nodes we can find their closest common ancestor. Given a piece of text T, and two leaves i, j (which correspond to the indices i, j respectively in T), their least common ancestor is the node which corresponds to where they split in the text. That is if their LCA has a path $\sigma_0 \cdots \sigma_{k-1}$ from the root, then $T[i : i+k-1] = T[j : j+k-1]$ and $T[i+k] \neq T[j+k]$.

So let us construct the suffix tree with LCA of $T\#P\$$, then if we take a character $T[i]$ and $P[0]$, then their LCA corresponds to the largest j such that $T[i : i+j-1] = P[0 : j-1]$ Iterating k times, we can find j_2 such that $T[i+j : i+j_2-1] = P[j : j_2-1]$, and so on. If after k iterations (or fewer) we have covered P then we know that we have matched T with P up to k mismatches. Since at every index we must iterate k times, the total time is $O(nk)$.

3.5.2 Amir-Lewenstein-Porat

Now suppose we have two strings $T[0 : n-1]$ and $P[0 : m-1]$, for every $\sigma \in \Sigma$ we define T_σ to be T where $T_\sigma[i]$ is 0 when $T[i] = \sigma$ and otherwise 1, and we define P_σ to be $P_\sigma[i] = 1$ when $P[i] = \sigma$ and otherwise zero. Now, if we convolve P_σ and T_σ, that is for every $0 \leq i \leq n-m-1$, we have

$$(T_\sigma \star P_\sigma)[i] = T_\sigma[i : i+m-1] \cdot P_\sigma$$

where the product is the standard dot product. If we now define

$$O = \sum_{\sigma \in \Sigma} T_\sigma \star P_\sigma$$

we get that $O[i]$ is how many mismatches there are at index i between T and P.

For example take $T = \text{abcbacbbacabbc}$ and $P = \text{abcbca}$, then

$$\begin{aligned} T &= \text{abcbacbbacabbc}, & P &= \text{abcbca} \\ T_a &= 01110111010111, & P_a &= 100001 \\ T_b &= 10101100111001, & P_b &= 010100 \\ T_c &= 11011011101110, & P_c &= 001010 \end{aligned}$$

Using FFT, convolving T_σ and P_σ takes $O(n \log m)$ time, so all in all computing O takes $O(n|\Sigma| \log m)$ time. Notice that this method supports the use of wildcards: a character ϕ which can be matched with any other character. We simply define $T_\sigma[i] = 0$ when $T[i] = \sigma$ or ϕ. We can then go over O and return the indices where $O[i] \leq k$.

But this takes $O(n|\Sigma| \log m)$ time, which can be quite long for large alphabets. Instead suppose we had a filter F such that if P matches with T at an index, so must F. Thus if F doesn't match with T then P won't either, and we can excuse ourselves from checking that specific index for a match.

We create a filter F which is equal to P except at certain indices where we place a wildcard. We first assume that P has $2k$ distinct symbols, that is $|\Sigma| = 2k$ (since if a symbol occurs in T but not P, we know to never count it as a match), and then place wildcards so that each character in F shows up only once. Now let us create an output array of the same size of T and initialize it to zero: $O[0 : n-1] = 0$. We iterate over T and at $T[i]$ we find the index j such that $T[i] = P[j]$, and then we increment $O[i-j]$ by 1 (since there is a match between $T[i-j : i-j+m-1]$ and P). After we iterate over T, $O[i]$ will have the number of matches between $T[i : i+m-1]$ and P (i.e. m minus the Hamming distance).

If at $T[i]$ there are at most k mismatches, then $O[i] \geq k$ (since there are $2k$ characters in P). So next we go over O and find for which indices $O[i] \geq k$ and then check if there is actually k mismatches there (by comparing with P).

Notice though that we incremented O at most n times, so $\sum O[i] \leq n$. Thus if we have N indices where $O[i] \geq k$, we have that $Nk \leq \sum O[i] \leq n$ and so $N \leq \frac{n}{k}$. So we need to check only $\frac{n}{k}$ places, and each takes $O(k)$ time, so in total this algorithm takes $O(n)$ time.

Now recall that we required there to be $2k$ distinct characters in P. Now instead we want to make this work for any pattern P. So instead suppose we can create a filter F which is a wildcard except for at $2k$ places, such that every character occurs in the filter at most $\sqrt{k}$ times. Not every pattern can satisfy this.

Now at each $T[i]$ we must increment O at every index where $P[j] = T[i]$, which is at most $O(\sqrt{k})$ places. In total this takes $O(n\sqrt{k})$ time. Now, we have that $\sum O[i] \leq n\sqrt{k}$ and so $Nk \leq \sum O[i] \leq n\sqrt{k}$ so $N \leq \frac{n}{\sqrt{k}}$. Thus the time to check is $O(n\sqrt{k})$ as well (checking $\frac{n}{\sqrt{k}}$ indices each takes $O(k)$ time). So assuming we can construct such a filter, this algorithm takes $O(n\sqrt{k})$ time.

Otherwise we cannot create such a filter, this means that there exists at most $2\sqrt{k}$ characters in P which occur in P at least $\sqrt{k}$ times (and other characters as well). As otherwise there exists at least $2\sqrt{k}$ characters in P which occur at least $\sqrt{k}$ times, and so we can just take these characters and create a filter. Suppose then $\sigma \in \Sigma$ is common (occurs at least $\sqrt{k}$ times), then define T_σ as before but where $T_\sigma[i] = 1$ when $T[i] = \sigma$ and P_σ as before. Then $T_\sigma \star P_\sigma$ gives the number of times where T matches with P only counting σ. So we then sum over all the common characters to get O_{heavy}, and this takes $O(n2\sqrt{k}\log m) = O(n\sqrt{k}\log m)$ time.

For uncommon characters we just do the previous algorithm where at $T[i]$ (which is uncommon) we find all $P[j] = T[i]$ and increment an array O_{light} by 1. We then sum $O_{heavy} + O_{light}$. At each i there are at most $\sqrt{k}$ matches and so this takes $O(n\sqrt{k})$ time. So in total we have taken $O(n\sqrt{k}\log m)$ time.

So in the first case it takes $O(n\sqrt{k})$ time, and in the second case $O(n\sqrt{k}\log m)$. So our algorithm takes $O(n\sqrt{k}\log m)$.

We can improve this to $O(n\sqrt{k\log m})$ where

(**1**) In the first case (where a filter can be built), require that a filter have $2k$ non-wildcard spots where each character occurs at most $\sqrt{k\log m}$ times.

(**2**) In the second case, there then are at most $2\sqrt{k}/\sqrt{\log m}$ characters which are common: occur at least $\sqrt{k\log m}$ times). Then perform the algorithm as discussed on this new definition of commonness.

3.6 Search with Wildcards

Suppose our pattern has wildcards as well. Our previous algorithm which ran in $O(n|\Sigma|\log m)$ time works here of course as well. But if Σ is large then this is not efficient. So instead what we can do is encode Σ as binary: each character will be converted into $\log|\Sigma|$ bits, then we run our algorithm on the encoded T and P. The new length of T is $n\log|\Sigma|$, and the new length of P is $m\log|\Sigma|$, so all in all the time is

$$O\big(n\log|\Sigma|\log(m\log|\Sigma|)\big) = O\big(n\log|\Sigma|\log m\big)$$

3.6.1 L_2^2 Matching

Suppose we have a text $T[0:n-1]$ and a pattern $P[0:m-1]$, and we want to compute the L_2-norm squared distance between T and P. I.e. we want to create an array $L[i] = \|T[i:i+m-1] - P\|_{L_2}^2$. Note that this is just

$$L[i] = \sum_{j=0}^{m-1}(T[i+j] - P[j])^2 = \sum_{j=0}^{m-1} T[i+j]^2 - 2\sum_{j=0}^{m-1} T[i+j]P[j] + \sum_{j=0}^{m-1} P[j]^2$$

The first sum can be computed for every index in $O(n)$ time (compute it for $L[0]$, then $L[i+1] = L[i] + T[m]^2 - T[i]^2$). The second sum is just a convolution, so it can be computed in $O(n\log m)$ time. And the third and final sum can be computed once in $O(m)$ time. So L can be computed in $O(n\log m)$ time.

Now suppose we want to perform L_2^2 matching with wildcards: T and P can have a wildcard character ϕ which is matched (equal) to any other character. That is, $x - \phi = 0$ for any x. We can do this as follows:

(**1**) First we replace the wildcards in the text and pattern with 0 to create T' and P' and compute the L_2^2 matching to get L'.

(**2**) Define $T''[i] = 1$ when $T[i] = \phi$ and zero otherwise, and define $P'' = (P')^2$. Convolve T'' with P'' to get C''.

(**3**) Define $T''' = (T')^2$ and $P'''[i] = 1$ when $P[i] = \phi$ and zero otherwise. Convolve T''' with P''' to get C'''.

(**4**) Return $L = L' - C'' - C'''$.

Notice then that

$$L'[i] = \sum_{T[i+j],P[j]\neq\phi}(T[i+j] - P[j])^2 + \sum_{T[i+j]=\phi} P'[j]^2 + \sum_{P[j]=\phi} T'[i+j]^2$$

Where we use P' and T' because if $T[i+j] = P[j] = \phi$ then they match and don't add to the sum, so we count the wildcard places as zero. And

$$C''[i] = \sum_{T[i+j]=\phi} P'[j]^2, \qquad C'''[i] = \sum_{P[j]=\phi} T'[i+j]^2$$

Thus we have that $L = L' - C'' - C'''$ as required. This takes $O(n \log m)$ time, as each step does.

3.6.2 One Error

Suppose we want to look for k-mismatches where $k = 1$ and we want to support wildcards. We just showed a way to compute

$$L[i] = \sum_{j=0}^{m-1} \big(T[i+j] - P[j]\big)^2$$

with wildcards. Now let us define

$$S[i] = \sum_{j=0}^{m-1} j\big(T[i+j] - P[j]\big)^2$$

Now suppose $T[i : i+m-1]$ matches with P except for at index ℓ, then we get that $L[i] = (T[i+\ell] - P[\ell])^2$ and $S[i] = \ell(T[i+\ell] - P[\ell])^2$. So we get that $\ell = S[i]/L[i]$. Notice also that if there are no mismatches, then $S[i] = 0$ and so we can easily check if there are no mismatches.

But what if there are two or more mismatches? It is possible that it gives an invalid output (a fraction), but it can also give a valid index. Let $\ell = S[i]/L[i]$, then if ℓ gives a valid index then there is at least one mismatch. If $T[i+\ell] = P[\ell]$, then there must be more than one mismatch. But if they are distinct, we still don't know if there is a single mismatch. Notice though that $L[i] = \sum_{T[i+j] \neq P[j]} (T[i+j] - P[j])^2$, and so if ℓ gives a valid index, we can look at $L[i] - (T[i+\ell] - P[\ell])^2$. If ℓ is the only mismatch, $L[i] = (T[i+\ell] - P[\ell])^2$ and so this will be zero. Otherwise it will be more than zero, and so we know there exists more than a single error.

Now, computing S works as follows:

$$S[i] = \sum_{j=0}^{m-1} jT[i+j]^2 - 2\sum_{j=0}^{m-1} T[i+j](jP[j]) + \sum_{j=0}^{m-1} jP[j]^2$$

The first sum is a convolution of $(0, 1, \ldots, m-1)$ with $T' = T^2$. The second sum is a convolution of T with $P'[j] = jP[j]$. The third sum can be computed once in $O(m)$ time. The convolutions take $O(n \log m)$ time.

3.6.3 A Probabilistic Algorithm for k Mismatches with Wildcards

We will now use One Error to form a probabilistic to return the indices of all the mismatches in the text. Let us define r (to be determined later) patterns $P_{S_1}, \ldots, P_{S_r}$ where

$$P_{S_t}[i] = \begin{cases} P[i] & \text{with probability } 1/2k \\ \phi & \text{with probability } 1 - 1/2k \end{cases}$$

Let S_t be the indices in P_{S_t} which are not wildcards. If P_{S_t} has only a single mismatch from $T[i : i+m-1]$ then One Error will return the index of this mismatch. Let $E = \{\ell_1, \ldots, \ell_{k'}\}$ be the set of mismatches in $T[i : i+m-1]$ (let us assume $k' \leq k$), then the probability of this occurring is

$$\begin{aligned} \mathbb{P}(E \cap S_j = \{\ell_i\}) &= \mathbb{P}(\ell_i \in S_j, \forall t \neq i\colon \ell_t \notin S_j) = \mathbb{P}(\ell_i \in S_j) \cdot \mathbb{P}(\forall t \neq i\colon \ell_t \notin S_j) \\ &= \frac{1}{2k} \cdot \big(1 - \mathbb{P}(\exists t \neq i\colon \ell_t \in S_j)\big) \geq \frac{1}{2k} \cdot \left(1 - \frac{k'-1}{2k}\right) \\ &\geq \frac{1}{4k} \end{aligned}$$

Then the probability that one of the patterns from the r witnesses the error ℓ_i is greater than $1 = \big(1 - \frac{1}{4k}\big)^r$, so let $r = 4k \log(n^c) = O(k \log n)$. Then the probability that one of the patterns witnesses the error is at least $1 - \big(1 - \frac{1}{4k}\big)^{4k \log(n^c)} \approx 1 - \frac{1}{n^c}$. Now the text has at most nk mismatches, and so the probability that one is not witnessed is, by the union bound, at most $\frac{nk}{n^c} \leq \frac{1}{n^{c-2}}$, which is also small.

So suppose that using the pattern P_{S_t} on index i we get the mismatch ℓ_i^t. Then we get the matrix of mismatches

$$\begin{matrix} P_{S_1} & \ell_0^1 & \ell_1^1 & \ell_2^1 & \cdots \\ P_{S_2} & \ell_0^2 & \ell_1^2 & \ell_2^2 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \cdots \\ P_{S_r} & \ell_0^r & \ell_1^r & \ell_2^r & \cdots \end{matrix}$$

Then with high probability the ith column gives all the mismatches at index i.

Note that the time complexity of this algorithm is $O(nr \log m) = O(nk \log n \log m)$ since we must perform One Error on each pattern.

3.7 Suffix Arrays

Suppose we have text T, as we saw before we can create a suffix tree. The issue is storing pointers is not ideal, so instead we would like to store a suffix *array*. The suffix array is defined as follows: it is an array SA where $SA[i]$ is the ith suffix in the lexicographic ordering. That is, we impose an ordering on Σ (the special character \$ is defined to be smaller than all other characters), use this to order all the suffixes of T, and place these in order in an array. We also define an array LCP where $LCP[i]$ is the length of $LCP\big(T[SA[i-1]], T[SA[i]]\big)$ (so if $T[SA[i-1]] = \omega\omega'$ and $T[SA[i]] = \omega\omega''$, then $LCP = |\omega|$).

For example if $T =$ ababb then its prefixes are:

Index	Suffix	Sorted Suffixes	Index
0	ababb\$	\$	5
1	babb\$	ababb\$	0
2	abb\$	abb\$	2
3	bb\$	b\$	3
4	b\$	babb\$	1
5	\$	bb\$	4

So $SA = [5, 0, 2, 3, 1, 4]$ and $LCP = [-, 0, 2, 0, 1, 1]$.

The suffix array and LCP array can be constructed from a suffix tree in $O(n)$ time. And a suffix tree can be constructed from a suffix array and LCP array in $O(n)$ time as well.

3.8 Pattern Matching in the Streaming Model

In the *communication model*, we are not concerned with time or space complexity, rather with the amount of information being exchanged between players. We want to reduce the amount of data needed to transport information.

For example, suppose Alice lives on the moon and Bob on Earth. Alice has an array of n numbers $a_0, \ldots, a_{n-1}$ and Bob does too: $b_0, \ldots, b_{n-1}$, all the numbers are between 0 and n^c. Alice and Bob want to compute what the joint median of these two arrays are: i.e. the median of $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}$. The naive way of doing this is for both Alice and Bob to exchange their arrays and both compute the joint median. Since each element of an array requires $O(\log n)$ bits, this protocol requires an exchange of $O(n \log n)$ bits.

Now let us consider the following alternative protocol. Let Alice and Bob both sort their arrays so they have $a_0, \ldots, a_{n/2}, \ldots, a_{n-1}$ and $b_0, \ldots, b_{n/2}, \ldots, b_{n-1}$. Alice then sends $a_{n/2}$ to Bob. Notice that if $a_{n/2} = b_{n/2}$ then the joint median is just $a_{n/2}$ so Bob can send back $=$ in $O(1)$ bits and Alice and Bob will both know the joint median. Otherwise, suppose $a_{n/2} > b_{n/2}$, then we know that the median must be between $b_{n/2}$ and $a_{n/2}$. So anything greater than $a_{n/2}$ and less than $b_{n/2}$ cannot be the median. So Bob sends back $<$ and Alice knows to disregard $a_{n/2+1}, \ldots, a_{n-1}$ and Bob knows to disregard $b_0, \ldots, b_{n/2-1}$. The joint median is still the same in this new joint array (of $a_0, \ldots, a_{n/2}, b_{n/2}, \ldots, b_{n-1}$), whose size is half that of the original. So now we have the same problem but on an array half the size. Continuing recursively, we see that we require $O(\log n)$ recursive calls. Each call transmits $O(\log n)$ bits of information, so in total this protocol uses $O(\log(n)^2)$ bits of information.

In the *streaming model*, we can only store something like *poly*$(\log n)$ space and we must read the input sequentially. So for example we can create streaming models for the following problems with relative ease:

(**1**) *Counter*: we can store a counter of the number of bits read in $O(\log n)$ bits.

(**2**) *Min/Max*: we can store the minimum/maximum of a stream of numbers in $O(\log U)$ bits where U is the size of the universe.

(**3**) *Average*: we can store the average of a stream of numbers in $O(\log U + \log n)$ bits (given n numbers the maximum value is nU, and we need a counter to store the number of numbers we have read, all in all $O(\log(nU) + \log n) = O(\log U + \log n)$.

Now suppose Alice and Bob both have a set of n numbers, and they want to figure out if their sets are disjoint. It can be shown that this requires $\Omega(n)$ bits to be transmitted.