7 Randomized Linear Programming

Problem 1 *The 2D* **linear programming** *problem is the following constrained optimization problem:*

$$\begin{array}{rc} \textit{maximize} & c_1x_1 + c_2x_2 \\ \textit{subject to:} & a_{11}x_1 + a_{12}x_2 \leq b_1 \\ & a_{21}x_1 + a_{22}x_2 \leq b_2 \\ & \vdots \\ & a_{n1}x_1 + a_{n2}x_2 \leq b_2 \end{array}$$

Let $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$, $c = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$, and $A = \begin{pmatrix} a_{11} & a_{12} \\ \vdots & \vdots \\ a_{n1} & a_{n2} \end{pmatrix}$, then we have the matrix format of the LP:

$$\begin{array}{rc} \text{maximize} & c^T x \\ \text{subject to:} & Ax \leq b \end{array}$$

7.1 Inner Product and Cross Product

Consider two 2D vectors $v_1 = \begin{pmatrix} r_1 \cos\alpha \\ r_1 \sin\alpha \end{pmatrix}$ and $v_2 = \begin{pmatrix} r_2 \cos\beta \\ r_2 \sin\beta \end{pmatrix}$.

Recall the **inner product** of these two vectors are defined as $r_1r_2 \cos\alpha \cos\beta + r_1r_2 sin\alpha \sin\beta$.

Since $r_1r_2 \cos\alpha \cos\beta + r_1r_2 sin\alpha \sin\beta = r_1r_2 (\cos\alpha \cos\beta + r_1r_2 sin\alpha \sin\beta) = r_1r_2 \cos(\alpha - \beta)$, geometrically, the inner product between two vectors v_1 and v_2 is simply the product of their lengths multiplied by the cosine of the angle between them. Since cosine is an even function, inner product for real vectors are commutative.

Recall the **cross product**, denoted by $v_1 \times v_2$ is the vector $|r_1r_2 \cos\alpha \sin\beta - r_1r_2 \sin\alpha \cos\beta| \vec{n}$, where $\vec{n}$ is the directional vector given by the **righthand rule**, i.e., with the index finger pointing along $\vec{v_1}$ and middle finger pointing along $\vec{v_2}$.

Since $|r_1r_2 \cos\alpha \sin\beta - r_1r_2 \sin\alpha \cos\beta| = |r_1r_2 \sin(\beta - \alpha)|$. Thus, the cross product calculates the area defined by the two vectors.

The cross product is also closely related to the **determinant** of the matrix $\left| \begin{pmatrix} r_1 \cos\alpha & r_2 \cos\beta \\ r_1 \sin\alpha & r_2 \sin\beta \end{pmatrix} \right| = r_1r_2 \sin(\beta - \alpha)$. Note that if $\beta \geq \alpha$ and $\beta - \alpha \leq 180°$, then $\sin(\beta - \alpha) > 0$, and $\vec{v_2}$ is counter clockwise from $\vec{v_1}$.

7.2 Partial Derivative and Gradient

Let $f(x) : \mathcal{R}^n \to \mathcal{R}$ be a function of multi-variables, where $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$.

A **partial derivative** of f is the derivative of f with respect to a particular variable x_j, while holding all other variables constant, and is denoted by $\frac{\partial f}{\partial x_j}$. Intuitively, one could view a partitial derivative as the "rate" of the change of f along a particular axis. Now, assume that we move

from x to $x + \Delta x$, where $\Delta x = \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_n \end{pmatrix}$, what is the impact of this move to the value of f, i.e, $\Delta f = f(x + \Delta x) - f(x)$? Since when x_j increases to $x_j + \Delta x_j$, the impact of this change to f is $\frac{\partial f}{\partial x_j}\Delta x_j$, the total impact is the sum of the changes along each axis and is $\sum_{j=1}^{n} \frac{\partial f}{\partial x_j}\Delta x_j = \begin{pmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \cdots & \frac{\partial f}{\partial x_n} \end{pmatrix} \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_n \end{pmatrix}$.

The vector $\begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$ is called the **gradient** of f and is denoted by ∇f. Thus, we have arrived at the first order Taylor expansion:

$$f(x + \Delta x) - f(x) = \nabla f^T \Delta x,$$

or

$$f(x + \Delta x) = f(x) + \nabla f^T \Delta x = f(x) + < \nabla f, \Delta x >,$$

where $< \nabla f, \Delta x >$ is the inner product

As an example, the gradient of the linear function $f(x) = c_1 x_1 + c_2 x_2$ is $\nabla f = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$

7.3 Gradient Search

Recall the Taylor Expansion of $f(x + \Delta x) \approx f(x) + \nabla f^T \Delta x = f(x) + \langle \nabla f, \Delta x \rangle$, where ∇f^T is the transpose of ∇f, and $\langle \nabla f, \Delta x \rangle$ is the inner product of the two vectors ∇f and Δx. Now if one wants to iteratively maximize f, we want move from x to $x + \Delta x$, such that $f(x + \Delta x) \geq f(x)$. Using the Taylor expansion, we have:

$$f(x + \Delta x) = f(x) + \langle \nabla f, \Delta x \rangle \geq f(x)$$

This implies that:

$$\langle \nabla f, \Delta x \rangle \geq 0$$

In other words, if we want to maximize/increase $f(x)$, we need to move along a direction that is within 90° to ∇f.

7.4 Line and Halfspace

A straightline l that goes through a point $p_0 = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$ and perpendicular to a **norm** vector $\vec{n} = \begin{pmatrix} a \\ b \end{pmatrix}$ is the collection of 2D points $p = \begin{pmatrix} x \\ y \end{pmatrix}$, such that the vector $\overrightarrow{p_1 p}$ is perpendicular to $\vec{n}$, i.e., the inner product $< \vec{n}, \overrightarrow{p_0 p} >= 0$. (See Figure 7.4 (a) for illustrations.)

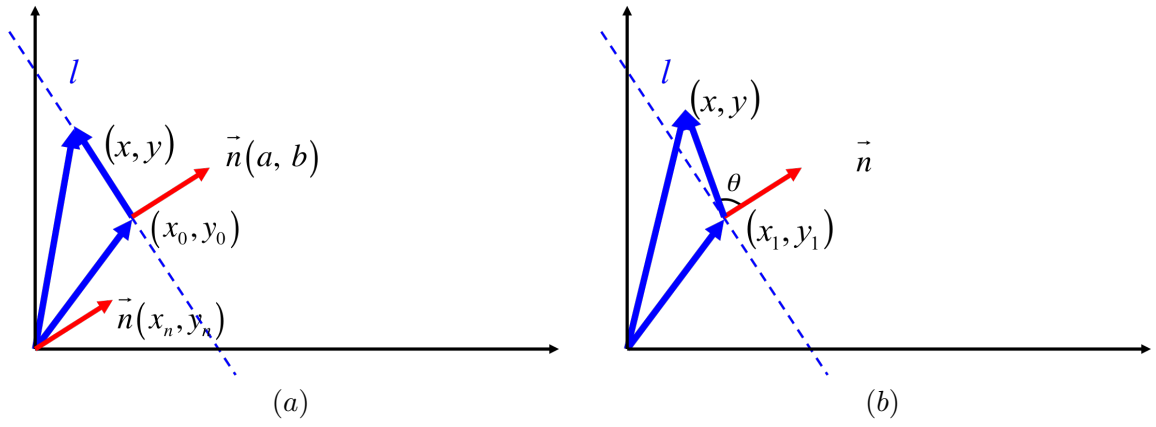

Figure 4:

To put this mathematically, we have $\left\langle \begin{pmatrix} a \\ b \end{pmatrix}, \begin{pmatrix} x - x_0 \\ y - y_0 \end{pmatrix} \right\rangle = 0$. In other words, $a(x-x_0)+b(y-y_0) = 0$, i.e., $ax + by - (ax_0 + by_0)$. Now if we let $c = -(ax_0 + by_0)$, then we have the familiar line function $ax + by + c = 0$.

Note: Since multiply both sides of an equation with -1, we end up with the same equation, the two equations $ax + by + c =$ and $-ax - by - c = 0$ defines the same line. Geometrically, this implies reversing the direction of the norm vector $\vec{n}$. To remove this ambiquity, we could require the coefficient $a \geq 0$, meaing the norm direction is always within 90° to the positive $X-$direction.

A line in 2D space partitions the plane into two sub-spaces. The **positive subspace** is the side that is in the same direction of the norm vector $\vec{n}$, the other half space is called the **negative halfspace**.

The positive halfspace is the collection of 2D points p, such that the vector $\overrightarrow{p_1 p}$ is within 90° to $\vec{n}$, i.e., the inner product $< \vec{n}, \overrightarrow{p_0 p} >\geq 0$. (See Figure 7.4 (b) for illustrations.) In other words, $\left\langle \begin{pmatrix} a \\ b \end{pmatrix}, \begin{pmatrix} x - x_0 \\ y - y_0 \end{pmatrix} \right\rangle > 0$. Hence $a(x - x_0) + b(y - y_0) > 0$, i.e., $ax + by - (ax_0 + by_0) > 0$. Now if we let $c = -(ax_0 + by_0)$, then the positve halfspace are all the points satisfying the inequality $ax + by + c > 0$.

Similarly, the negative halfspace is defined by the inequality $ax + by + c < 0$.

The inequality $ax + by + c \geq 0$ is obviously the positive halfspace plus the dividing straight line $ax + by + c = 0$.

Note that since we are using vector operations, all of the above can be generalized to high dimensions. More precisely, an $n-$dimensional hyperplane is defined by the function $a_1x_1+a_2x_2+...+a_nx_n = b$. The positive halfspace in $n-$dimension is defined as $a_1x_1 + a_2x_2 + ... + a_nx_n \geq b$

7.5 Geometry of LP

Consider the following example:

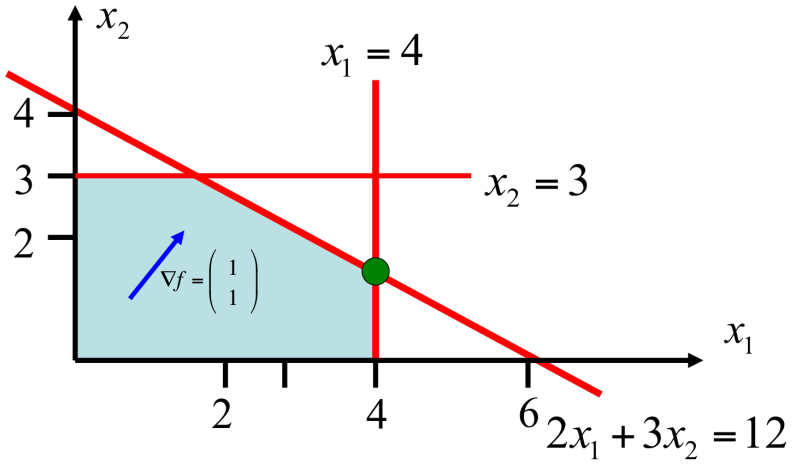

Figure 5: Geometry of 2D Linear Programming.

$$\begin{array}{rl} \max & f(x) = x_1 + x_2 \\ \textit{subject to} & 2x_1 + 3x_2 \leq 12 \\ & x_1 \leq 4 \\ & x_2 \leq 3 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \end{array}$$

We know that each constraint in an LP specifies a halfplane. The **feasible domain** of the LP is in fact the intersection of all these half spaces. Since each half space is convex, the intersection between two convex sets is also convex, the feasible domain of an LP is also convex. In 2D space, if the feasible domain is bounded, then the feasble domain is a convex polygon. (See Figure 5 for illustrations.)

Now suppose we have located a feasible point x. To determine if x is optimal, we ask the question can be find another feasible point $x' = x + \Delta x$, such that $f(x') > f(x)$? If no such point x' exists, then obviously x is optimal. From the section on gradient search, we know that in order for $f(x') > f(x)$, Δx must be an **ascent** and **feasible** direction, where "ascent" implies it is within 90° to the gradient ∇f, and "feasible" implies $x + \Delta x$ still belongs to the feasible domain.

Observe that if x is in the **interior** of the feasible domain, then all 360° around x are feasible, for non-zerolinear functions $f(x)$, **feasible ascent** directions always exist. Thus the optimal solution to the LP, if exists, must lie on the boundary of the feasible domain.

Suppose x^* is an optimal solution to the LP on a boundary edge $\overline{p_1 p_2}$, where p_1 and p_2 are two extreme vertices of the feasible domain. Since x^* is optimal, then ∇f must be perpendicular to the edge $\overline{p_1 p_2}$. Thus, all points on this edge all optimal, and therefore, one of the extreme vertices of the feasible domain must be an optimal solution (if exsits) to the LP. This is the basis of the famous **simplex** method, which iterate through all the extreme points of the feasible domain before terminating at the optimal solution.

7.6 Randomized Algorithm for 2D LP

For easy of discussion, let the constaints be denoted by $h_1, h_2, ..., h_n$, where each h_j represents a halfplane defined by the $j-$th constraint. Let $H_j = h_1 \cap h_2 \cap ... \cap h_j\}$, i.e., the intersection of the first j constraints. Observe that $H_1 \supseteq H_2 \supseteq ... \supseteq H_n$. With this notation, our LP can also be denoted as (f, H_n), where f is the objective function. The randomized algorithm for 2D LP is an incremental algorithm. In each iteration, the algorithm finds the optimal solution of the LP (f, H_j), and terminates when with solution to f, H_n.

Clearly, one can determine the solution to the LP (H_1, f) and (H_2, f) within constant time. The key question is suppose we have the solution v_{j-1} to the LP (f, H_{j-1}), how can we find the solution v_j to the LP (f, H_j)?

Observe that there are two possibilities:

1. $v_{j-1} \in H_j$: This implies that $v_{j-1} \in h_j$. Since $H_j \subseteq H_{j-1}$, v_{j-1} is also the solution to the LP (f, H_j), i.e., $v_j = v_{j-1}$. Note that it only takes constant time to check if v_{j-1} is in h_j.

2. $v_{j-1} \notin H_j$: Note that there are two scenarios here:

   (a) $H_j \neq \phi$: This implies that $v_j \neq v_{j-1}$. From the previous analysis, we know that v_j must be one of the two endpoints of the edge induced by the half space h_j. To obtain the edge induced by h_j, we can simply calculat the $j-1$ intersections between h_j and all the previous $j-1$ halfspaces. This takes linear time.

   (b) $H_j = \phi$: This implies the LP has no feasibible domain, and therefore has not solution. Observe that if H_j is empty, then the half space h_j will not induce an edge to the feasible domain. Just like in case (a), this will also takes $O(j)$ to determine.

Clearly, the worst case running time for the above incremental algorithm will take $\sum_{j=1}^{n} j = O\left(n^2\right)$ time. However, what if we randomly perturb the input order of the constraints? Will it allow us to beat this worst case running time?

Let X_j be the R.V. describing the running time of the $j-$iteration. X_{j-1} has two possible values. When $v_{j-1} = v_j$, $X_j = 1$, and when $v_{j-1} \neq v_j$, $X_j = j$. The running time of the whole algorithm is $\sum_{j=1}^{n} X_j$, and the expected running time is: $E\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} E\left[X_j\right]$.

Since $E\left[X_j\right] = j \cdot Pr\left(v_{j-1} \neq v_j\right) + 1 \cdot Pr\left(v_{j-1} = H_j\right)$, in order the figure out this expectation, we need to know the probability of $Pr\left(v_{j-1} \neq v_j\right)$.

Suppose that v_j is the lines of the two lines l_s and l_t, where l_s and l_t are the lines defining the half spaces h_s and h_t from $\{h_1, h_2, ..., h_j\}$. Notice that in order for v_j to be not equal to v_{j-1}, h_j must be either h_s or h_t. Since we randomly perturb the order of the constraints, every halfspace is equally likely to be h_j. Thus, the probability that h_j is either h_s or h_t is $\frac{2}{j}$, i.e., $Pr\left(v_{j-1} \neq v_j\right) = \frac{2}{j}$. Hence, $E\left[X_j\right] = j \cdot \frac{2}{j} + +1 \cdot \left(1 - \frac{2}{j}\right) = 2 + \frac{j-2}{j} \leq 3$. Thus $\sum_{j=1}^{n} E\left[X_j\right] = O(n)$.

8 Uniform Hashing

The problem we are interested here is how to store n numbers from the universe $\{0, 1, 2, ..., M-1\}$ for $M >> n$ to a table of size n, while achieving searching, insertion, and deletion in **constant expected time**. The data structure that can achieves this is called a **hash table**. The key to design a hash table is the construction of a **hash function** $h : \{0, 1, 2, ..., M-1\} \to \{0, 1, 2, ..., n-1\}$,

Example 3 *Most of the hash functions are based on modulo operators, for example,* $h(x) = x \mod n$.

Since h is mapping a larger finite set to a smaller finite set, with the **pigeon hole principle**, multiple numbers from $\{0, 1, 2, ..., M-1\}$ will be mapped to the same item in $\{0, 1, 2, ..., n-1\}$. In other words, for $i, j \in \{0, 1, 2, ..., M-1\}$ and $i \neq j$, $h(i) = h(j) = k \in \{0, 1, 2, ..., n-1\}$.

Note that if i and j are in the collection of items that we want to store in the hash table, they will be assigned to the same entry in the table. This situation is called a **collision**. The usual way of handling collision is called **separate chaining**, where a linked list is attached to each entry of the hash table to store the "collided" items.

Searching a linked list is well known to take linear time in terms of the length of the list. Now imagine if the hash function is not well chosen, and $O(n)$ number of items are mapped to the same entry, then searching the hash table will take $O(n)$ time. Thus the performance of a hash table is determined by the number of collisions in expectation and in worst-case scenarios.

Definition 21 *Let H be a finite family of hash functions, each mapping from a universe $U = \{0, 1, 2, ..., M-1\}$ to a smaller range $\{0, 1, 2, ..., n-1\}$. We say that H is universal if for distinct keys $x, y \in U$, with a hash function h chosen uniformly at random from H, the odds of collision is $\leq \frac{1}{n}$.*

Observation 5 *Consider a pair of distinct keys $x \neq y$. Imagine that we partition H into two catorogies $H_1 = \{h | h \in H, h(x) = h)y)\}$ and $H_2 = \{h | h \in H, h(x) \neq h(y)\}$. Since every hash function $h \in H$ is equally-likely to be chosen, the odds of collision when we pick a random hash function to use is $\frac{|H_1|}{|H_1|+|H_2|} = \frac{|H_1|}{|H|}$. From the definition, if H is universal, then $\frac{|H_1|}{|H|} \leq \frac{1}{n}$.*

Observation 6 *Imagine that we have a hash function that can "randomly" hash keys to any entry in the hash table with equally-likely probability. Now image we have two keys $x \neq y$, what is the odds of collision? In other words, assuming x is somewhere in the hash table, we are hashing y into the table with equal probability to any entry. Clearly, the odds of collision is $\frac{1}{n}$. Thus, the concept of universal hashing is to match the capability of a hash function that can randomly hash keys into the table.*

We now look at an example of a universal family of hash functions.

Example 4 *Let n be the size of the hash table. Let M be the size of the universe. Let p be a prime number, such that $p > M$. Then the family of hash functions*

$$H = \{h(a, b) = ((ax + b]) \mod p) \mod n | a \in \{1, 2, ..., p-1\}, b \in \{0, 1, 2, ..., p-1\}\}$$

is a universal family of hash functions.

Proof:

Lemma 1 *If* $b \equiv c \mod n$, *then* $m|(b-c)$.

Lemma 2 *If* $m|(b-c)$, *then* $b \equiv c \mod n$.

Lemma 3 *Given integers* a, b, p, *if* $GCD(a,p) = 1$ *and* $GCD(b,p) = 1$, *then* $GCD(ab,p) = 1$.

Observe that we have $p(p-1)$ hash function, i.e., $|H| = p(p-1)$. We have to show that for arbitrary pair of distinct keys $x \neq y$, the number of hash functions h from H can gives a collision (i.e., h(x) = h(y)) is $\leq \frac{|H|}{n} = \frac{p(p-1)}{n}$.

Consider an arbitrary hash function $h(a,b) \in H$. Let $s = ax + b \mod p$ and $t = ay + b \mod p$. Notice that if $s = t$, then $(ax+b) \equiv (ay+b) \mod p$. In other words, $p|\left((ax+b) - (ay+b)\right)$, i.e., $p|a(x-y)$. However, since $x \neq y$ and $x - y \in \{-(M-1), -M-2, ..., -1, 1, ..., M-2, M-1\}$, hence $GCD(p, x-y) = 1$. On the other hand, $a \in \{1, 2, ..., p-1\}$, $GCD(a,p) = 1$. From one of the lemmas, we know that $GCD(a(x-y), p) = 1$. A contradiction, thus $s \neq t$.

Thus distinct (x,y) are mapped to distinct (s,t) by the function $ax + b \mod p$. If $h(x) = h(y)$, then $s \mod n = t \mod n$.

Now consider two functions $h_1(a_1,b_1), h_2(a_2,b_2) \in H$, with $h_1 \neq h_2$, i.e., $(a_1,b_1) \neq (a_2,b_2)$. Let $s_1 = a_1x + b_1$ and $t_1 = a_1y + b_1$. Let $s_2 = a_2x + b_2$ and $t_2 = a_2y + b_2$. We will show that $(s_1,t_1) \neq (s_2,t_2)$. This implies that for fixed x, y, there is a one-to-one corresponence between the two sets $\{(s,t)|s \neq t, s, t \in \{0, 1, ..., p-1\}\}$ to H. Thus, the number of hash functions from H that will cause a collision is the number of (s,t) pairs such that $s \equiv t \mod n$.

Since $(a_1,b_1) \neq (a_2,b_2)$, there are three possibilities:

(1) $a_1 = a_2$, $b_1 \neq b_2$:

$s_1 - s_2 = (a_1x + b_1) - (a_2x + b_2) \mod p = (b_1 - b_2) \mod p \neq 0$. Hence $s_1 \neq s_2$, and $(s_1,t_1) \neq (s_2,t_2)$.

(2) $a_1 \neq a_2$, $b_1 = b_2$:

$s_1 - s_2 = (a_1x + b_1) - (a_2x + b_2) \mod p = (a_1 - a_2)x \mod p$.

$t_1 - t_2 = (a_1y + b_1) - (a_2y + b_2) \mod p = (a_1 - a_2)y \mod p$.

Since $x \neq y$, either at least one of x and y is nonzero. Thus, at least one of $(a_1 - a_2)x \mod p$ and $(a_1 - a_2)y \mod p$ is non-zero. Hence either $s_1 \neq s_2$ or $t_1 \neq t_2$, and $(s_1,t_1) \neq (s_2,t_2)$.

(3) $a_1 \neq a_2$, $b_1 \neq b_2$:

We will use proof by contradiction here, and assume that $s_1 = s_2$ and $t_1 = t_2$. Thus we have:

$$\begin{cases} a_1x + b_1 \equiv a_2x + b_2 \mod p \\ a_1y + b_1 \equiv a_2y + b_2 \mod p \end{cases} \quad (1)$$

This gives:

$$\begin{cases} p|(a_1 - a_2)x + (b_1 - b_2) \\ p|(a_1 - a_2)y + (b_1 - b_2) \end{cases} \quad \text{Equation (2)}$$

This gives:

$$p|(a_1 - a_2)(x - y) \quad \text{Equation (3)}$$

Since $a_1 - a_2 \neq 0$ and $x - y \neq 0$, $p \not| (a_1 - a_2)(x - y)$, and therefore the assumption is wrong and $(s_1, t_1) \neq (s_2, t_2)$.

Since the number of hash functions from H that will cause a collision is the number of (s, t) pairs such that $s \equiv t \mod n$, to figure out the number of hash functions causing collisions, all we need to do is to count the number of (s, t) pairs with $s \neq t$ such that $s \equiv t \mod n$.

Suppose that $p = qn + r$, where $0 \leq r < n$ is the remainder. We can partition the numbers $\{0, 1, ..., p - 1\}$ into n categories based on their remainders when divided by n, i.e.,

mod $n =$	0	1	...	r	$r+1$	...	$n-1$
	0	1	...	r	$r+1$	...	$n-1$
	n	$n+1$	...	$n+r$	$n+(r+1)$	...	$2n-1$
	$2n$	$2n+1$	...	$2n+r$	$2n+(r+1)$	...	$3n-1$
	$3n$	$3n+1$	...	$3n+r$	$3n+(r+1)$	...	$4n-1$
	...	...	...	...	...	...	...
	qn	$qn+1$	...	$qn+r$			

The number of (s, t) pairs with $s \neq t$ causing a collision is when they happen to be in the same category, thus the number of collision is:

$$\begin{aligned} & r(q+1)q + (n-r)q(q-1) \\ &= q((nq+r) - n + r) \\ &\leq q(p-1) \end{aligned}$$

Thus the odds of collision is bounded by $\frac{q(p-1)}{p(p-1)} \approx \frac{1}{n}$.

□

9 String Matching and Rolling Hash

Problem 2 *Given a string $s[1..n]$ and a pattern $p[1..m]$, $m < n$ over some alphabet Σ, determine if the pattern p occurs in the string s.*

For example, let $s =$ "$ACGTCGTA''$ and $p =$ "TCG'', then the answer will be yes.

9.1 The Shifting Naive Algorithm

1. for $j = 1$ to n do:
2. for $k = 1$ to m do:
3. if $s[j + k - 1] \neq p[k]$
4. break ;
5. else
6. return Yes and j

Obviously, the running time of the naive shifting string search algorithm is $O(mn)$. If m and n are both large, then this quadratic and is very expensive. The inner loop of the algorithm takes $O(m)$ time, is it possible to reduce the inner loop to constant time?

9.2 The Rabin-Karp Algorithm

1. $hp = hash(p[1..m])$
2. for $j = 1$ to n do:
3. $hs = hash(s[j..j + m - 1])$
4. if $hs = hp$
5. return Yes and j

In order to hash a string of length m, we need to convert the string to a number. The easiest way to do it is to convert the number to base$-|Simga|$ number. More specifically, let $b = |\Sigma|$, then we can convert a string of length m such as $p[1..m]$ to the interger $p[1]b^{m-1} + p[2]b^{m-2} + ... + p[m-1]b + p[m]$.

Since hashing a string of size m will take at least $O(m)$ time, the Rabin Karp algroithm doesn't appear to be any improvement. However, if one considers two successive iterations, i.e., hashing $s[j..j+m-1]$ and $s[j+1..j+m]$, the two strings only differs by one character. Is it possible to update the hash result instead of re-compute the hash from stractch? This is the idea of the rolling hashing. The goal of rolling hash is to calculate $hashs[j + 1..j + m]$ from $hashs[j..j + m - 1]$ in constant time.

9.3 The Rabin-Karp Rolling Hash Algorithm

Let $b = |\Sigma|$. Let $x = s[j]b^{m-1} + s[j+1]b^{m-2} + ... + s[j+m-2]b + s[j+m-1]$, $y = s[j+1]b^{m-1} + s[j+2]b^{m-2} + ... + s[j+m-1]b + s[j+m]$. Let $h(x) = x \mod q$ for some integer q be the hash function.

Since $y = (x - s[j]b^{m-1})b + s[j+1] = xb - s[j]b^m + s[j+1]$,

$$h(y) = y \mod q$$

$$= (xb - s[j]b^m + s[j+1]) \mod q$$

$$= ((x \mod q)(b \mod q) - (s[j] \mod q)(b^m \mod q) + s[j+1] \mod q) \mod q$$

$$= (h(x)(b \mod q) - (s[j] \mod q)(b^m \mod q) + s[j+1] \mod q) \mod q$$

Notice that $h(x)$ is already available, $b^m \mod q$, $b \mod q$ can be pre-computed, we can obtain $h(y)$ in constant time. This idea is called the rollering hash algorithm.