Massachusetts Institute of Technology
6.046J/18.410J: Introduction to Algorithms
Professors Piotr Indyk and Bruce Tidor

Handout 21
April 24, 2003

Problem Set 6 Solutions

Problem 6-1. On-line String Matching

In an on-line algorithm, the input is generated as the algorithm is running. The idea is to solve the problem efficiently before seeing all the input. You can't scan forward to look at 'future' input, but you can store all input seen so far, or some computation on it.

Solution: For both parts we are going to use Karp-Rabin (KR) algorithm seen in the class (the KMP algorithm could be used as well). As in the class, we will assume that the alphabet is $\{0, 1\}$ (the algorithm can be easily modified to handle arbitrary alphabet). Recall that KR used $\log^{O(1)} n$ time to find a random prime q, and $O(m)$ time to "hash" the pattern P; we will refer to these two steps as "preprocessing".

(a) In this setting, the text $T[1 \ldots n]$ is being broadcast on the network, one letter at a time, in the order $T[1], T[2], \ldots$. You are interested in checking if the text seen so far contains a pattern P. Every time you see the next letter of the text T, you want to check if the text seen so far contains P.

Design an algorithm that solves this problem efficiently. Your algorithm should use no more than $\Theta(m)$ time on preprocessing P (with m being the length of P). In addition it should do only constant amount of work per letter received. Your algorithm can be randomized, with constant probability of correctness.

Solution: After the preprocessing, the KR algorithm computes the hash values for the m-length substrings of T in an incremental way. Whenever a new symbol $T[i]$ is given, KR computes the hash value of the substring $T[i - m + 1 \ldots i]$ in constant time. Thus, KR works in our setting without any modifications.

(b) Now say that you have the same pattern P, but the text $T[1 \ldots n]$ is being broadcast in reverse. That is, in the order $T[n], T[n-1], \ldots$ Modify your algorithm so that it still detects the occurence of P in the text $T[i \ldots n]$ immediately (i.e., in constant time) after the letter $T[i]$ is seen.

Solution: For an array $A[1 \ldots n]$, let A^R be the "reverse" of A, i.e., $A^R[i] = A[n - i + 1]$. Searching for an occurrence of P in T is equivalent to searching for an occurence of P^R in T^R; thus, we can focus on the latter task. Since the symbols of T are given in the reverse order, it means that the symbols of T^R are given in the proper order (i.e., $T^R[1], T^R[2] \ldots$). Thus, we can find the occurences of P^R in T^R by using the algorithm from the part (a).

Problem 6-2. Average-case analysis

Assume that for a pattern matcher, both the pattern and the text are randomly generated. That is, each symbol in both is selected independently and uniformly at random from $\{0, 1\}$. Show that the simple string matching algorithm given in the lecture, slide 3, runs in expected time $O(n)$.

Can you reproduce the result if the pattern in fixed, and only the text is random ?

Solution

(a) Recall that $T^s = T[s+1 \ldots s+m]$. The main idea is to observe that the comparison of T^s with P can be done very quickly if T and P are random. Specifically, let l_s be the largest integer l such that the first l letters of T^s match the first l letters of P. The algorithm's running time is bounded by $O(l_0 + \ldots l_{m-n})$. Thus, to bound its expected running time, it suffices to bound $E[\sum_s l_s] = \sum_s E[l_s]$.

What is the value of $E[l_s]$? Every time when we compare the next symbols of T^s and P, the probability that they are equal is $p = 1/2$. Thus, $E[l_s] = 1/p = 2$.

Therefore, the expected running time of the algorithm is bounded by $O(n)$.

(b) The above analysis relied exclusively on the following fact:

> For any s, i, the probability that $T^s[i] = P[i]$ given that $T^s[j] = P[j]$ for all $j < i$, is at most $1/2$.

This fact holds even if T is fixed and the pattern is random, or vice-versa.

Problem 6-3. Summations revisited

We will once again revisit the summations problem.

(a) Assume you are given two sets $A, B \subset \{0 \ldots m\}$. Your goal is to compute the set $C = \{x + y : x \in A, y \in B\}$. Note that the set of values in C could be in the range $0 \ldots 2m$. Your solution should run in time $O(m \log m)$ (the sizes of $|A|$ and $|B|$ do not matter).

Example:

$$\begin{aligned} A &= \{1, 4\} \\ B &= \{1, 3\} \\ \\ C &= \{2, 4, 5, 7\} \end{aligned}$$

Solution: The key realization is that when two polynomials multiply, their exponents are added in every possible pairing. So we take our set A, and do the following: Turn the set into a polynomial of maximum degree m. The coefficient of x^0 is the number of times 0 appears in the set. The coefficient of x^1 is the number of times 1 appears in the set, and on. Do the same with B. The example would then become:

Example:

$$\begin{aligned} A &= x^4 + x^1 \\ B &= x^3 + x^1 \\ \\ C &= x^7 + x^5 + x^4 + x^2 \end{aligned}$$

$C = A * B$. If we use the FFT, we can compute C in $O(m \log m)$.

Note that in the above example, all coefficients ended up being 1. However, a coefficient of three would indicate that there are three ways to get to that sum (although not what the three ways were). This will be important for part b.

The running time of this algorithm is the time to convert A and B into polynomials. If A is a set (not a multiset) $\in \{0..m\}$ then this can be done in $O(m)$. Same is true for B. To multiply them takes $O(m \log m)$. Finally, to take the resulting polynomial and output the set C takes $O(2m) = O(m)$. The entire algorithm takes $O(m)$.

For a simple argument of correctness, we observe that every coefficient in the target polynomial ($c_i x^i$) represents the number of ways we can add a number from A and from B to get to i. First, we note that all coefficients in A and B are either zero or one (no multisets were indicated). Clearly if there is no $a \in A, b \in B$ that sum to i, then for all $j \in \{0..i\}$, at least one of a_j, b_{i-j} is zero. Similarly, if $c_i = z$, then there are z pairs a_j, b_{i-j} since each such pair will multiple out to x^i, and then the sum of all of those pairs will yield $z * x^i$.

As an aside, we also note that if we allowed multisets (which we need for the next problem), everything still works. If there are two ones in one set, and two threes in the other, there are indeed four different ways to get a target value of 3.

(b) Let us now return to an old problem from PS3. Assume you are given an array of n integers A which can take on values from $\{1 \dots m\}$, $n \leq m$. Additionally, you are given a target sum x. The goal is to check if there are three different indices i, j, k such that $A[i] + A[j] + A[k] = x$. Show how to do it in time $O(m \log m)$.

Solution: We want to solve the much harder problem of finding if it is possible to find unique indices $i, j, k \in A$ such that $A[i] + A[j] + A[k] = x$. Note that we merely need to answer 'yes' or 'no'. Not what the triplet is.

To do this, we use the idea in part (a). The core idea is to find $A * A * A$. Then look at the x^{th} coefficient. If it is a zero, we know for a fact you can't get to this value using any three values in A. Getting a value in there isn't enough though, because it would be non-zero if $x = 27$ and there was a single element in A which is three. So clearly we need to get rid of all cubes that were counted. We also need to get rid of all items generated as a square times another value (since those were also counted). Given a target value q. If we can take some a_i^3 we need to subtract an x^q from our resulting polynomoial. If $q = a_i^2 a_j$ we need to subtract $3x^q$ (as there are really THREE ways to take two i's and one j: iij, iji, jii, and each one of those was counted in $A * A * A$).

We can compute all possible cubes in time $O(m)$ by linearly scanning through all the values in A and cubing them. We can compute all squares in $O(m)$. We can compute all squares times any value in A in $O(m \log m)$ using the FFT. But this will actually also count up all cubes, so we need to fix our counts.

Let A be the original set turned into a polynomial. Let S be the set of all values in A square turned into a polynomial. Let C be the set of all cubes as a polynomial. The final polynomial we want to look at is therefore:

$$(A * A * A) - (3S * A) + (2C)$$

If this polynomial has a non-zero x^{th} coefficient then we can now say 'yes'.

Running Time: As analyzed during the construction, our algorithm takes $O(m \log m)$.

This algorithm does what we want since each coefficient represents the number of ways to combine 3 elements from A to get the respective sum (as explained in part (a)). Since we then remove the number of ways to combing the same element three times, and the number of ways to combine an element with itself plus some other element, we make every coefficient c_i represent the number of ways to get three unique elements of A that sum up to i.

Problem 6-4. String matching reduction.

Prof. Tidor designed a very efficient algorithm for the string matching problem with "don't care" symbols. Given a pattern P of size m and text T of size n, Tidor's algorithm runs in time $T(n, m)$. However, it only accepts strings (pattern and text) over the alphabet $\{A, G, T, C\}$ (plus the "don't care" symbols).

Show how to use Tidor's algorithm as a black-box to obtain an algorithm that handles *any* alphabet Σ. Your algorithm should have running time $O((n + T(n, m)) \log |\Sigma|)$. You can assume that the elements of Σ are represented by integers from $\{0 \ldots |\Sigma| - 1\}$.

Solution:

We need to take the input text and input pattern, convert it into a pattern that the black box understands, and then parse the output of the black box.

To convert any input text into { A, G, C, T }, we simply assign $A = 0, G = 1, C = 2, T = 3$. Take Σ and write out the letter in order using base-4. Then convert every letter in the big alphabet into a set of letters from the short alphabet. Each original letter will need $\alpha = \log_4 |\Sigma|$ letters. Do the same for the input pattern, but convert any "don't care" into α "don't cares".

Create α texts. The first text is just the first 'bit' of each letter. The second text is the second bit, etc. So we now have α texts, each of length n. Similarly, we create α patterns of length m.

Run the first pattern against the first text. Second pattern against the second text, etc. A match is possible in many places for each text. However, a good match is only one where ALL texts have a match in the same position. Each text generates at most n matches. There are α texts, which we can scan through incrementally to find all places where there are α matches, one-per-text. This takes us $O(\alpha n)$ time.

Runtime: Let $\alpha = \log_4 |\Sigma|$. It takes $O(\alpha n)$ to convert the text into α texts of length n. It takes $O(\alpha m)$ to do the same for the pattern. It takes $O(\alpha T(n, m))$ to do all the pattern matches with don't cares. Finally, it takes an additional $O(\alpha n)$ to go through the output and spit out where all texts have a match in the same place.

Justification: Clearly if there is a match of the original pattern in the original text, the encoding in base-4 doesn't change it. There will still be a match in all α texts. If there is no match in the original text with the original pattern, at least one of the α texts will not match. This will therefore catch all matches in the correct running time.