Design and Analysis of Algorithms March 15, 2012
Massachusetts Institute of Technology 6.046J/18.410J
Profs. Dana Moshkovitz and Bruce Tidor Handout 13

Problem Set 4 Solutions

This problem set is due **at 9:00pm** on **Wednesday, March 14, 2012**.

Problem 4-1. Word-Search: Pattern Matching Revisited

In recitation we covered an $O(n \lg n)$ FFT-based algorithm for finding the offset in the text that gives the best match score between a pattern string and a target text. For this problem, we are interested in finding the first exact occurrence of the pattern in the text. Let $t = t_1 t_2 \ldots t_n$ be a target text and $p = p_1 p_2 \ldots p_m$ a pattern, both over alphabet $\Sigma = \{0, 1\}$ with $m \leq n$. Identifying the first exact occurrence of the pattern in the text amounts to finding the smallest $j \in \{1, 2, \ldots, n-m+1\}$ such that for $1 \leq i \leq m$, it holds that $t_{j+i-1} = p_i$.

In this problem we will work with the word RAM model, where we can store each integer in a single word (or "byte") of computer memory. The number of bits that can be stored in a single word is $\lceil \lg n \rceil$. So, arithmetic operations on words or $O(\lg n)$-bit numbers take $O(1)$ time. Finally, with this model comparing two n-bit numbers takes $O(n)$ time.

For this problem you can assume that you are provided with a black-box prime number generator, and that it takes $O(1)$ time to sample $O(\lg K)$-bit primes, for K polynomial in n. Also, you can assume that $\pi(k) \sim \frac{k}{\ln k}$, where $\pi(k)$ be the number of distinct primes less than k.

(a) Define $f_p : \{0,1\}^m \to \mathbb{Z}$ as $f_p(X) := g(X) \mod p$, where p is a prime and $g : \{0,1\}^m \to \mathbb{Z}$ is a function that converts an m-bit binary string to a corresponding base 2 integer. Note that if X is equal to Y then $f_p(X) = f_p(Y)$. However, if X differs from Y then it can still be the case that $f_p(X) = f_p(Y)$. We will refer to cases where the results of evaluating the function on two different string inputs are equal as false positives.

Take the set $P = \{p_1, p_2 \ldots p_t\}$, where p_i are all primes less than some large integer K. Suppose we choose a prime p uniformly at random from the set P and take m-bit strings X and Y such that $X \neq Y$. Prove that we can bound the probability of a false positive as follows:

$$P\left(f_p(X) = f_p(Y)\right) \leq \frac{m}{t}$$

Hint: Consider the prime factorization of $|g(X) - g(Y)|$. Notice that the number of prime factors is at most m.

Solution:

A false positive happens when m-bit strings $X \neq Y$ but $f_p(X) = f_p(Y)$. This implies that the binary integers $g(X)$ and $g(Y)$ are the same modulo p, $g(X) - g(Y) \equiv 0$

$\mod p$. This will happen if and only if p is a prime factor of $g(X) - g(Y)$. Thus, we should count the number of distinct prime factors of $g(X) - g(Y)$. Let $g(X) - g(Y) = q_1 q_2 \dots q_c$ where q_i are its (not necessarily distinct) prime factors. Clearly, $q_i \geq 2$, since 2 is the smallest prime. Thus, $g(X) - g(Y) \geq 2^c$. Since $g(X), g(Y) \leq 2^m$, $c < m$. Hence, $g(X) - g(Y)$ can have at most m prime factors, which is also the maximum number of distinct prime factors it can have. When we sample a random prime from the set $P = p_1, p_2, \dots p_t$, the probability of us finding a prime factor of $g(X) - g(Y)$ and obtaining a false positive is at most:

$$P(f_p(X) = f_p(Y)) < \frac{m}{t}$$

Note that the above bound is very loose, we did not even use the distinctness of the prime factors, we can achieve a slightly better bound if we take that into consideration. Suppose $g(X) - g(Y) = q_1^{s_1} q_2^{s_2} \dots q_c^{s_c}$. Where $q_1, q_2, \dots$ are distinct. We have $g(X) - g(Y) > q_1 q_2 \dots q_c > 2 \times 3 \times 4 \times \dots \times c = c!$. Hence the bound on c is: $c! < 2^m$. Using Sterling's approximation, $\log(c!) = \Theta(c \log c)$, so we have $c \log c < \Theta(m)$, this implies $c = O(m/\log(m))$, because then $c \log c = \frac{m}{\log m}(\log m - \log\log m) = m$. Using this upperbound, the probability of a false positive is:

$$P(f_p(X) = f_p(Y)) < \frac{m}{t \log m}$$

An even tighter bound is possible, however, they do not improve the asymptotic running time of our algorithm in the subsequent parts.

(b) Let $X(j)$ be a length-m substring of the target text that starts at position j, for a given $j \in \{1, 2, \dots, n - m + 1\}$. Design a randomized algorithm that given $f_p(Y)$ and $f_p(X(j))$ determines if there is a match between the pattern and the target text for a given offset $j \in \{1, 2, \dots, n - m + 1\}$.

Solution:

Recall that a Monte Carlo algorithm is guaranteed to terminate quickly on any input, however it has a probability of returning the wrong result. Whereas a Las Vegas algorithm is guaranteed to be correct, but only terminates in some expected time, ie. in some cases it can take a long time. For this problem, we will present both types of algorithms.

Monte Carlo version:

If: $f_p(Y) = f_p(X(j))$, return "A match"

Else: return "Not a match"

Analysis: Notice that when $f_p(Y) \neq f_p(X(j))$, $X \neq Y$, so when our algorithm yields a negative result, it is always correct. If our algorithm yields a positive result, as we have concluded in part (a), the probability that it is a false positive is at most m/t.

Thus, the overall error probability of our algorithm is at most m/t. The algorithm only compares two numbers $f_p(Y)$ and $f_p(X(j))$, which are both at most $p-1$. If we can bound p to be poly(n,m), then it only takes $O(\log n)$ bits to represent both $f_p(Y)$ and $f_p(X(j))$, and comparing them would take $O(1)$ time in our model. Hence we have a Monte Carlo algorithm that runs in $O(1)$ time with an error probability of at most m/t.

Las Vegas version:

If: $f_p(Y) = f_p(X(j))$, check the strings $g(X(j))$ and $g(Y)$, if they are equal, return "A match"

Else: return "Not a match"

Analysis: The difference between this algorithm and the previous one is that when the finger prints $f_p(Y)$, $f_p(X(j))$ are the same, we must make sure the strings $g(Y)$ and $g(X(j))$ are actually the same. There is no other way to guarantee this other than to check them directly. Since $g(Y)$ and $g(X(j))$ are m-bit numbers, this takes $O(m)$ time.

If $g(Y)$ and $g(X(j))$ are actually the same, any algorithm must take $O(m)$ to verify this with 100% certainty (since we need to at least look at all the bits). So in this sense, our randomized algorithm isn't doing any worse.

If $g(Y)$ and $g(X(j))$ are not the same, from part (a), we know that the probability of a false positive is m/t, which means that our algorithm will compare two m-bit strings in $O(m)$ time but eventually conclude that they are not the same. Thus, the total expected running time when $g(Y) \neq g(X(j))$ is $O(1+m^2/t)$. This will be $O(1)$ if we let K be, for example, $\Theta(m^3)$, in which case, by the prime number theorem, $t = \Theta(m^3/\log(m))$. Notice that $f_p(Y)$ and $f_p(X(j))$ are still $O(\log n)$ in this case, so comparing them still takes $O(1)$ time.

Thus, this is an algorithm that runs in expected $O(1)$ time, and always outputs the correct answer. It turns out that the Las Vegas version is better suited for the subsequent parts.

(c) Design a formula that given $g(X(j))$ computes $g(X(j+1))$, where $X(j)$ is a length-m substring of the target text that starts at position j, for a given $j \in \{1, 2, \ldots, n-m+1\}$. Use it to compute $f_p(X(j+1))$ from $f_p(X(j))$.

Note that the formula should depend on X, j, and m.

Solution: We regard $g(X(j))$ as an m-bit binary string with the left most bit being the most significant, ie. $g(X(j)) = 2^{m-1}X_j + 2^{m-2}X_{j+1} + \ldots 2^0 X_{j+m-1}$. $g(X(j+1))$ is basically $g(X(j))$ shifted right by 1 digit, discarding the original most significant bit X_j in the process, while adding a new least significant bit X_{j+m}. Hence we have the following relation between $g(X(j))$ and $g(X(j+1))$:

$$g(X(j+1)) = 2(g(X(j)) - 2^{m-1}X_j) + X_{j+m}$$

so,

$$f_p(X(j+1)) = 2(f_p(X(j)) - 2^{m-1}X_j) + X_{j+m} \mod p$$

We only need to compute $2^{m-1} \mod p$ once and save it for all future uses, subtracting this constant takes $O(1)$ time in our model as long as p is poly(n,m). Multiplying by 2 is equivalent to adding two $\log p$ bit numbers, which can be done in $O(1)$ time. Finally, adding X_{j+m} clearly takes constant time. Thus it takes $O(1)$ time to compute $f_p(X(j+1))$ from $f_p(X(j))$.

(d) Suppose that $X(j)$ and Y differ at every string position. Give the best upper bound you can on the expected number of positions j such that $f_p(X(j)) = f_p(Y)$, where $j \in \{1, 2, \ldots, n-m+1\}$.

Solution:

Let C_k be the indicator variable that $f_p(X(k)) = f_p(Y)$. Then, the total number of false positives FP for all the positions is:

$$\mathbb{E}[FP] = \mathbb{E}[\sum_{i=j}^{n-m+1} C_i]$$

By linearity of expectation, this is equal to:

$$\mathbb{E}[FP] = \sum_{i=j}^{n-m+1} \mathbb{E}[C_i]$$

Since we are told that $g(X(j)) \neq g(Y)$ for all j, using the result of part (a), $\mathbb{E}[C_i] \leq m/t$ for every i. Hence,

$$\mathbb{E}[FP] = (n-m+1)\frac{m}{t} = O(\frac{nm}{t})$$

(e) Using parts above, design a randomized algorithm that determines if there is a match between a pattern and a target text in $O(n+m)$ expected running time. The algorithm should always return the correct answer.

Solution: We will determine K (the upper bound on p) at the end, for now we will treat it as a variable. Our algorithm works as follows:

1. sample a random prime from the range 1 to K. This can be accomplished using either the blackbox described in the introduction of the problem in $O(1)$ time. Or we can do it ourselves as follows:

First we generate a random number from 1 to K, not necessarily prime, this should take $O(1)$ time in our model if K is poly(n,m). Typically, pseudo random numbers are generated with "seeds" using modular arithmetic, but these can all be done in $O(1)$ time. We note here that deterministic primality testing algorithms exist that run in polylog(n) time. That is to say, given any number less than n, we can determine whether it is prime or composite in polylog(n) time. Thus, to pick a random prime number from the set, we random sample a number, use the deterministic primality testing algorithm, and repeat until we obtain a prime number. From the prime density theorem, the probability of us finding a prime number is at least $\pi(K)/K = 1/\log(K)$. So we need to try expected $O(\log(K))$ times before we actually find a prime number. This brings our total runtime to $\log(K) \times polylog(K) = polylog(K)$.

2. Compute $f_p(Y)$ and $f_p(X(1))$. Both of them are m-bit numbers, arithmetic operations will take $O(m)$ time.

3. Using $f_p(X(1))$, compute all $f_p(X(j))$ using the formula from part (c). We argued that each additional value only takes $O(1)$ time to compute, and there are $O(n - m + 1) = O(n)$ of them in total, so this step takes $O(n)$.

4. Compare $f_p(Y)$ to each of $f_p(X(j))$ computed above, using the Las Vegas version of the algorithm in part (b). Notice that the algorithm terminates as soon as we encounter the first exact match. Thus, evevn if the original text contains many exact matches, we will spend at most $O(m)$ time on a position where $g(X(j)) = g(Y)$. For all other cases, we may then assume $g(X(j)) \neq g(Y)$. We can then use the result from part (d). The total work to check is $O(n) + \mathbb{E}[FP] \times O(m)$. Since for every false positive, we need to do $O(m)$ work. This gives us a run time of $O(n + \frac{nm^2}{t})$.

Correctness: It is quite clear that the result of the algorithm is always correct, since we used the Las Vegas version of the algorithm in part (b). Whenever the algorithm returns a match at position k, we know that it is indeed a match because we checked the m-bit strings directly. Likewise, whenever the algorithm reports no match, it is also guaranteed to be correct. Thus, the first instance of a positive result is the position of the first exact match.

Run time analysis: The total runtime of the algorithm is $O(polylog(n) + m + n + \frac{nm^2}{t})$ or $O(m + n + \frac{nm^2 \log(K)}{K})$. We want to make the term $O(nm^2 \log(K)/K)$ small. For example, we can pick $K = \Omega(nm^2 \log(nm^2))$, so that $O(nm^2 \log(K)/K) = O(1)$, and hence arriving at a $O(n + m)$ algorithm. In fact, we can make K arbitrarily large, as long as K is poly(n,m), so that operations on $log(K)$ bit numbers can be done in $O(1)$ time.

(f) Provide a bound for the probability that the running time is more than 100 times the expected running time.

Solution:

Let T be the random variable that describes the running time of the algorithm. By Markov's inequality, we have

$$Pr(T \geq 100\mathbb{E}[T]) \leq \frac{\mathbb{E}[T]}{100\mathbb{E}[T]} = \frac{1}{100}$$

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012