

CS513 Design and Analysis of Algorithms

Fall 2008 - HW5 solution

Instructor: S. Muthukrishnan
TA: André Madeira

Problem 1:

First, note that we can use Schwartz's polynomial identity testing as seen in class. We simply pick r_1, \dots, r_n values at random from a finite field and compute $q = Q(r_1, \dots, r_n) = \det(M) - \prod_{i < j} (r_i - r_j)$ to check if q equals 0. Since the $\det(M)$ can be computed efficiently in $T(n)$ time (as stated in the exercise) the entire computation is efficient.

Unfortunately, the problem with this approach is that the computation of q can be extremely large and we were asked to make sure the numeric computations do not get too large. Observe that our goal is not in computing q exactly but merely verifying that $q = 0$. We avoid the expensive computation by verifying that $q \bmod p \equiv 0$ instead for some small and carefully chosen random prime p . Note that this calculation avoids computing q since we can use arithmetic modulo p while evaluating q , such that each intermediate value is smaller than p . Our claim is that if the identity holds it should hold modulo p as well. Therefore, we only need to bound the probability that $q \bmod p \equiv 0$ whenever $Q(x_1, \dots, x_n) \neq 0$. We omit the details, but it can be shown that for a sufficiently large p , the probability of error does not increase significantly over the error given by the Schwartz-Zippel result.

Problem 2:

First, note that $\frac{a}{b} \bmod n$ is equivalent to $ab^{-1} \bmod n$, but it is only defined if b is invertible modulo n . In turn, b is invertible modulo n iff $\gcd(b, n) = 1$. Thus, we can use the extended-Euclid algorithm to calculate $\gcd(b, n)$ and get x, y such that $bx + ny \equiv d \bmod n$. If $d \neq 1$, then return that there's no solution. Otherwise, note that in this case $bx \equiv 1 \bmod n$ and thus x is the multiplicative inverse of b . Therefore, we can just output $ax \bmod n$ and that takes only one extended-Euclid algorithm run.

Problem 3:

Basically, the algorithm performs a linear search in S (from $S[1]$ to $S[n - m + 1]$) to compare if $S[i \dots i + m - 1]$ differs by no more than k places from T and outputs any i that succeeds. Specifically, for each location i , the idea is to *skip* over the prefixes of $S[i \dots i + m - 1]$ that match corresponding prefixes of T and count the number of mismatches until more than k is found (or we've reached location $i + m - 1$). Using fingerprints, we show that counting mismatches can be done efficiently in $O(k \log m)$, thus requiring a total of $O(nk \log m)$ comparisons.

Let $f(\cdot)$ denote the fingerprint function. At any given location j , $i \leq j \leq i + m - 1$, consider the following phases and transitions. Let q_i denote the number of current mismatches for iteration corresponding to location i .

- 1) *when* $S[j] \neq T[j - i + 1]$: Update q_i with the number of mismatching locations immediately from location j onwards. This can be performed by a simple loop (no fingerprints involved) until a matching character is found or location $i + m - 1$ is reached. Move to phase 2) in case a matching character was found.
- 2) *when* $S[j] = T[j - i + 1]$: The idea is to skip over as many matching locations as possible. To do that, we first start comparing fingerprints of length $r = m - (j - i)$, or checking if

$$f(S[j \dots j + r - 1]) = ? f(T[j - i + 1 \dots j - i + 1 + (r - 1)]). \quad (1)$$

If (1) matched with this length, we're done and output location i if $q_i \leq k$, otherwise we halve r while (1) is still mismatching or until $r = 1$. Eventually, a match must occur because at $r = 1$, $f(S[j]) = f(T[j - i + 1])$ by definition of j in this phase. Furthermore, such exit condition guarantees that the previous fingerprint match (of length $2r$) failed. Therefore, the first mismatch after location j must be between locations $j + r$ and $j + 2r$. We can perform this procedure recursively in this new range. However, before each recursive call we must check if the immediately next location is the culprit mismatching character. If so, exit the recursion and we're back to phase 1).

We now argue that both phases can be done efficiently. Clearly, Step 1) makes at most $O(k)$ character comparisons, which we assume take each $O(1)$. It can be shown that Step 2) performs at most $O(\log m)$ fingerprint comparisons. Thus, assuming we can compute every fingerprint used above in $O(1)$ time, it takes $O(k \log m)$ fingerprint comparisons because there are at most $2k + 1$ phase transitions (can you see why?). If that is the case, the total cost for location i is $O(k + k \log m) = O(k \log m)$.

We proceed in showing how to compute each fingerprint in $O(1)$ time. We pre-compute the fingerprints of all substrings $S[1 \dots i]$ and $T[1 \dots j]$ for every $i \in [n]$ and $j \in [m]$. There are $n + m$ fingerprints respectively and they can be computed incrementally in $O(n + m)$ time —by the properties of the Karp-Rabin fingerprint. Store these fingerprints in auxiliary arrays indexed by the end index. Then, it is easy to see that any particular fingerprint $f(S[i \dots j])$ can be computed from $f(S[1 \dots j])$ and $f(S[1 \dots i])$ after appropriate shifting.

Finally, notice that the algorithm above assumed the fingerprints do not err. For a comprehensive solution, one must bound the probability of failure for a suitable choice of fingerprint range and/or double-check when a fingerprint comparison succeeds and thus bound the number of possible such checks.

Problem 4:

The generalization is similar to the 2-d case given in class. Permute the n d -dimensional points x_1, x_2, \dots, x_n . We build a data structure representing a d -dimensional grid and update each point one at a time. At the end of each update, all points in the data structure will be assigned to a d -dimensional hypercube. For storage, we can use a dictionary (with perfect hashing) and thus retrieve any point in $O(d)$ time.

We now describe a single update step. We place item x_i in the grid. Now, we must check if there are points x_j , $j \neq i$, whose distance $d(x_i, x_j) < \delta_{i-1}$, where δ_{i-1} is the closest distance between any pair of points in x_1, x_2, \dots, x_{i-1} . If not, we continue to point x_{i+1} ; otherwise we must recompute the grid. For the same reasons as in the planar case (seen in class), we need to check at most 5^d hypercubes around x_i (recall each has side length $\delta_i/2$). Because each hypercube has at most a single point except (possibly) the hypercube containing x_i by the definition of δ_{i-1} , we need to check only $O(5^d)$ pairs and thus perform $O(d5^d)$ comparisons (one for each dimension). The recomputation of the grid can be done in $O(id)$ time using at most $O(d)$ computation for all the i points currently stored.

We analyze the running time. Let the indicator r.v. X_i represent the event that point x_i is one of the endpoints of the closest pair. Then, we have:

$$\begin{aligned}
 T(n) &= \underbrace{O(n)}_{\text{permutation}} + \underbrace{O(nd5^d)}_{\text{comparisons}} + \underbrace{E \left[\sum_i^n O(id) \cdot X_i \right]}_{\text{recomputation}} \\
 &= O(nd5^d) + \sum_i^n O(id) \cdot E[X_i] && \text{(linearity of expectation)} \\
 &= O(nd5^d) + 2nd && \text{(since } E[X_i] = 2/i) \\
 &= O(nd5^d)
 \end{aligned}$$

For a fixed d , the natural generalization runs in expected linear time.

Problem 5:

The following Las Vegas algorithm is very simple. Our algorithm assumes there is a prime between $[x, y]$. Simply pick a random number r between $[x, y]$ and repeat until the blackbox indicates r is a prime. We analyze its running time.

By the Prime Number Theorem, the $\lim_{n \rightarrow \infty} \frac{\pi(n)}{n/\ln n} = 1$, where $\pi(n)$ is the number of primes that are less than or equal to n . The approximation is quite good for even reasonably small n . Let P denote number of primes in the range $[x, y]$, or approximately $\pi(y) - \pi(x)$. Then, the probability that a randomly picked r in $[x, y]$ is a prime is approximately $P/(y - x + 1)$. Therefore, we expect to invoke the blackbox about $(y - x + 1)/(\pi(y) - \pi(x))$ times until we find a prime. If the blackbox takes time $T(n)$, the algorithm takes total expected time $O(T(y - x)(y - x + 1)/(\pi(y) - \pi(x)))$.