15-451/651 Algorithms, Spring 2020
Recitation #4 Worksheet

Recap of this week's lectures:

- Picking a random prime, and the density of primes
- String equality testing and Karp-Rabin Fingerprinting
- Dynamic programming: top-down and bottom-up
- Examples: Knapsack, Independent set in trees, LCS, etc.

Fingerprinting

A String Matching Oracle: In this recitation we generalize the fingerprinting method described in lecture. Let $T = t_0, t_1, \ldots, t_{n-1}$, be a string over some alphabet $\Sigma = \{0, 1, \ldots, z-1\}$. Let $T_{i,j}$ denote the substring $t_i, t_{i+1}, \ldots, t_j$. We want to preprocess T such that the following test can be answered (with a low probability of a false positive) in constant time:

$$\text{Test if } T_{i,i+\ell} = T_{j,j+\ell}$$

First of all let's define the fingerprinting function. Let p be a prime, along with a base b (larger than the alphabet size). The Karp-Rabin fingerprint of T is

$$h(T) = (t_0 b^{n-1} + t_1 b^{n-2} + \cdots + t_{n-1} b^0) \bmod p$$

From now on we will omit the mod p from these expressions.

Now, to preprocess the string T, we will compute the following arrays for $0 \le i \le n$: (*Don't forget we are omitting the* mod*s!*)

$$\begin{aligned} r[i] &= b^i \\ a[i] &= t_0 b^{i-1} + t_1 b^{i-2} + \cdots + t_{i-1} b^0 \end{aligned}$$

Give algorithms for computing these in time $O(n)$: **Solution:**

$$r[i] = \begin{cases} 1 & \text{if } i = 0 \\ r[i-1] * b & \text{otherwise} \end{cases}$$

$$a[i] = \begin{cases} 0 & \text{if } i = 0 \\ a[i-1] * b + t_{i-1} & \text{otherwise} \end{cases}$$

Now write and prove a simple expression for $h(T_{i,j})$ in terms of the $r[]$ and $a[]$ arrays computed above.

Solution: Here is an expression

$$h(T_{i,j}) = a[j+1] - a[i] \cdot r[j-i+1]$$

$$\begin{array}{rcccc} a[j+1] & = & t_0 b^j + t_1 b^{j-1} + \cdots + t_{i-1} b^{j-i+1} & + & t_i b^{j-i} + \cdots + t_j b^0 \\ & = & a[i] \cdot b^{j-i+1} & + & h(T_{i,j}) \end{array}$$

So the end result is that we can test if $T_{i,i+\ell} = T_{j,j+\ell}$ by comparing $h(T_{i,i+\ell})$ with $h(T_{j,j+\ell})$.

The probability of a false positive can be made as small as desired by picking a sufficiently large random prime p, as seen in lecture. (Here we are not concerned with bounding the false positive probability.)

Extension to string comparison: Suppose we want to know not just if $T_{i,i+\ell}$ equals $T_{j,j+\ell}$, but we want to know the result of comparing these two strings. (That is we want to know if the first is less, equal to, or greater than the second.) Give an algorithm to do this that runs in $O(\log \ell)$ time: **Solution:** Do a binary search to find the smallest k in $0 \le k \le \ell$ where $T_{i,i+k}$ and $T_{j,j+k}$ differ. Return the result of the comparison $t_{i+k} : t_{j+k}$.

Implementation notes:

- When implementing these algorithms it's important to understand the difference between the mathematical mod operator and the "`%`", or the "`mod`" operator that appears in many programming languages. Usually (e.g. in C, C++, Java, Ocaml,...) the value of "`a % b`" has the same sign as `a`. e.g. `(-3) % 5` is `-3`, but using the mathematical mod , as we are using in these notes, $(-3) \bmod 5 = 2$. You must take this into account, or your code will not work.

- The mod operator must be applied often enough so as to guarantee that no overflow occurs. For example if you're computing $\left(\sum_{i=0}^{1000} f_i * g_i\right) \bmod p$. Say you're working in 64-bit signed arithmetic. If the f_i and g_i are up to, say, 10^9, then this sum will probably overflow the available 64 bits. So what you have to do is take the mod after adding each $f_i * g_i$ term into the summation.

- The danger of false positives increases with the increased number of tests being done. So be aware that this must be considered in any deployment of these algorithms. One way to mitigate the effect of false positives is to compute the hash function two or more times using different primes in place of p, or different base values in place of b. Another way is to judiciously check tests that return "equal" to guarantee that the overall algorithm is computing the correct result. (I am not aware of how to do this efficiently in the case of the comparison algorithm described here.)

Dynamic Programming

Longest Increasing Subseqeuence: Given an array A of n integers like `[7 2 5 3 4 6 9]`, find the longest subsequence that's in increasing order (in this case, it would be `2 3 4 6 9`). Give a dynamic-programming algorithm that runs in time $O(n^2)$ to solve this problem.

1. To keep things simple, first let's say you just need to output the *length* of the longest-increasing subsequence. E.g., in the above case, the length is 5. **Solution:** Suppose that for each $i' < i$ you have computed the length of the LIS of $A_{0\ldots i'}$ that ends with $A[i']$. How would you use this to solve the corresponding problem for i? $L[i] = \max\{L[i'] + 1 : i' < i, A[i'] < A[i]\}$, or $L[i] = 1$ if there are no such i'.

2. Now extend your solution to actually find the LIS. **Solution:** One approach is to also have a separate array that stores the argmax, that is, the index i' such that $L[i] = L[i'] + 1$ when computing the max above. One can then read off the sequence by going backwards from the end.

Balanced Partition: You have a set of n integers each in the range $0, \ldots, K$. In time $O(n^2K)$, partition these integers into two subsets such that you minimize $|S_1 - S_2|$, where S_1 and S_2 denote the sums of the elements in each of the two subsets. **Solution:** Let S be the sum of all the integers. Then $S \leq nK$. To minimize $|S_1 - S_2|$, it suffices to find a set A_1 whose numbers sum to $S_1 \leq \lfloor S/2 \rfloor$ that is as close to $S/2$ as possible. And this can be done by a dynamic program like for knapsack in time $O(nS) = O(n^2K)$.

Food for thought: Is it possible to get an algorithm with polynomial runtime dependent on n alone?

Solution: Partition is NP-complete, so probably not. (If you do, extra credit for you!)

Egg Drop II: Electric Boogaloo: Recall the egg drop problem. We are in a k-story building, and we have n weirdly strong eggs. We want to find the floor f, such that an egg dropped from f will break, but an egg dropped from $f-1$ will not break. For example, if we have 1 egg and 10 floors, we need 10 drops in the worst case because we drop on floor 1, then floor 2, ..., to floor 10 if the egg is extremely powerful.

Give and analyze a dynamic-programming algorithm that gives the smallest number of egg drops d, such that we can find f in at most d drops using n eggs. Your algorithm should run in polynomial-time in n and k.

Solution: Here is a solution that runs in $O(nk^2)$ time.

We initialize a table T with $O(nk)$ entries of the form (#eggs, #floors).

Here are the base cases: $T(1,k) = k$, $T(n,0) = 0$, $T(n,1) = 1$, $T(0,k) = \infty$.

Note that

$$T(n',k') = \min_{1 \le i < k'} \max \Big(T(n', k'-i), T(n'-1, i-1) \Big)$$

That is, if we start by dropping the egg at floor i, then two things could happen. If the egg stays intact, then we have narrowed our search to floors $i+1$ through k'. This means that we are searching through $k'-i$ floors, but still with n' eggs.

If the egg breaks, then we have narrowed our search to floors 1 to $i-1$. This means that we are searching through $i-1$ floors, but with one fewer egg: $n'-1$.

If we choose to drop the egg from floor i, we have to be prepared for the worse case of these two possibilities, which is the max of the two values. In the end we want to find the floor with the best worst case scenario, so we take the min over all the possible choices of i.

This recurrence gives us that we have to take maxes and mins over $2k' \in O(k)$ items to fill in each table entry, and there are $O(nk)$ table entries, giving a total of $O(nk^2)$ work.