CISC621 Algorithm Design and Analysis, Spring 2016
Homework set IV, problems 10, 11, 12,13, due Thursday, April 28, 7:00pm

Check the "homework sheet" from the syllabus for general homework details. In particular, each homework solution is on an entirely separate (set of) sheet(s) of paper and is identified with your name(s). Do not staple solutions to two or more problems together. Submit in 201 Smith Hall directly to Fanchao Meng or place on shelf marked CISC 621.

10. [Individual Problem — modular powering]

    Consider this proposition:

    Let n be a positive integer, then $a^{\phi(n)+1} = a \bmod n$ for all $a \in \mathbb{Z}_n^*$ (all $a \in (1, 2, \ldots, n-1)$).

    This proposition seemingly follows from theorem 31.30 in CLRS. However that theorem is misstated. It is valid if and only if a is relatively prime to n.

    (a) Show that the proposition is false when n is the square of a prime, $n = p^2$.

        Solution: For n = 4, $\phi(n) = 2$ but $2^3 = 8 = 0 \bmod 4$ in contradiction to the proposition. another example: $n = 9, \phi(n) = 6, 3^7 = 0 \bmod n$

    (b) Show that the proposition is true when n is a product of two distinct primes, $n = pq$. [Hint: Use Chinese remainder theorem]

        Solution: Here $\phi(n) = (p-1)(q-1)$, so let $e = \phi(n) + 1 = (p-1)(q-1) + 1 = n - p - q + 2$. If a is relatively prime to n then $a^e \bmod p$ is a, since $a^{(p-1)} = 1$, so $a^{(p-1)(q-1)} = 1$ and $a^e = a \bmod p$. Similarly $a^e = a \bmod q$. By CRT a^e is the unique number modulo n which is a modulo p and a modulo q, i.e. a itself.

        Otherwise a is a multiple of p or of q (not both, unless a = 0 — a trivial case). Suppose $a = bp \bmod n$. Then $a^e = b^e p^e \bmod n$. Here $b^e = b \bmod n$ by the above argument (b is relatively prime to n), and $p^e = 0 mod p$ while $p^e = p \bmod q$ (Fermat theorem for prime q). By CRT the unique value modulo n which is 0 mod p and p mod q is p. Therefore $p^e = p \bmod n$. ...Similarly for $q^e = q \bmod n$.

    (c) Extra credit: A positive integer n is called *square free* if $a^2 | n$ implies $a = 1$. Show that the proposition is true if and only if n is square free.

        Solution: Note that the above argument just uses that p and q are relatively prime to each other, except where $p^e \bmod q$ is determined to be $p \bmod q$ by Fermat's little theorem. Assuming just that q is relatively prime to p, replace that with inductive argument on size of n (q less than n) to deduce that $a^{\phi(n)+1} = a \bmod n$ when n is square free.

        In the first part we saw that $p^e = 0 \bmod p^2$, so if $n = p^k m$ for some $m > 1$ we have $p^e = 0 \bmod n$, but $p \neq 0 \bmod n$ Thus $p^e \neq p \bmod n$.

    Remark: This exercise shows that RSA works even in the unlikely event that the message encrypted is a multiple of one of the prime factors of n.

11. [Individual Problem — dynamic programming] We have studied LCS, the problem of finding the largest common subsequence of two sequences, and we have studied LIS, the problem of finding the largest increasing subsequence of a single sequence. Let's put these together. Find the length of LCIS, the largest common increasing subsequence of two given sequences. Concretely, let lcis(a,m, b,n) be a function which is given two arrays of numbers, a and b, of lengths m and n, respectively. It returns the length of a longest sequence $c = c_1, \ldots, c_k$ such that $c_i < c_{i+1}$ for $i \in 1..k-1$ and c is a subsequence of a and of b.

    The task is to design an efficient algorithm for lcis() and analyze it.

    Solution: Let `LCIS-end(a,m,b,n)` be the problem to give the length of the longest common increasing sequence *ending precisely at* position m of a and position n of b. Then LCIS-end has the recursive solution:

    (a) if m = 0 or n = 0 return 0

(b) if $a_m = b_n$ then return 1 + max(LCIS-end(a,i,b,j) such that $i < m, j < n$ and $a_i < a_m$

(c) else return 0

This is evidently correct and runs in $O(m^2n^2)$ time if memoized since indices from 0 to m and 0 to j are computed, with pair i,j taking $O(ij)$ time. The space is $O(mn)$, since the memo table must be about m by n. There is a very nice faster algorithm in "A fast algorithm for computing a longest common increasing subsequence" by I-Hsuan Yang, Chien-Pin Huang, and Kun-Mao Chao. Thanks to Fanchao Meng for finding this.

12. [Group Problem — Hankel matrices] *Hankel matrices* are matrices having a pattern that arises in many applications including hidden Markov models. Specifically, A Hankel matrix is a matrix $h = (h_{i,j})$ such that $h_{i,j} = h_{k,l}$ whenever $i + j = k + l$. Let indexing be zero based so that the row and column indices run from 0 to n-1 rather than from 1 to n. A $n \times n$ Hankel matrix can be represented by the $2n - 1$ values in the first row and last column, $H_0, H_1, \ldots, H_{2n-2}$. We can represent h by $h_{i,j} = H_{i+j}$.

$$h = \begin{pmatrix} a & b & c & d \\ b & c & d & e \\ c & d & e & f \\ d & e & f & g \end{pmatrix}, H = (a, b, c, d, e, f, g)$$

(a) Give a linear time algorithm to add two Hankel matrices when they are represented as indicated above.

Solution: if

$$h = \begin{pmatrix} a & b & c & d \\ b & c & d & e \\ c & d & e & f \\ d & e & f & g \end{pmatrix},$$

is represented as $H = (a, b, c, d, e, f, g)$ and

$$k = \begin{pmatrix} h & i & j & k \\ i & j & k & l \\ j & k & l & m \\ k & l & m & n \end{pmatrix},$$

is represented as $K = (h, i, j, k, l, m, n)$ then

$$hk = \begin{pmatrix} a+h & b+i & c+j & d+k \\ b+i & c+j & d+k & e+l \\ c+j & d+k & e+l & f+m \\ d+k & e+l & f+m & g+n \end{pmatrix},$$

is represented as $HK = (a + h, b + i, c + j, d + k, e + l, f + m, g + n)$. It costs $2n - 1 = O(n)$ additions to compute HK, which represents the product.

(b) A Hankel matrix, represented as above can be multiplied by a vector as follows.

```
mvHankel(H, v, n) {
  for i from 0 to n-1 do
     w[i] = 0;
     for j from 0 to n-1 do
        w[i] = w[i] + H[i+j]v[j];
  return w
}
```

This algorithm runs in $\Theta(n^2)$ time just as general matrix vector product does. Write and analyze an algorithm to compute Hankel matrix times vector product in $O(n \log(n))$ time.

Hint: Hankel matrix times vector product corresponds to (part of) polynomial multiplication.

Solution: To see what is going on, let's look at the example where $n = 4$ and expand the Hankel matrix to 3n-2 by n.

$$\begin{pmatrix} * \\ * \\ * \\ w_0 \\ w_1 \\ w_2 \\ w_3 \\ * \\ * \\ * \end{pmatrix} , = \begin{pmatrix} 0 & 0 & 0 & h_0 \\ 0 & 0 & h_0 & h_1 \\ 0 & h_0 & h_1 & h_2 \\ h_0 & h_1 & h_2 & h_3 \\ h_1 & h_2 & h_3 & h_4 \\ h_2 & h_3 & h_4 & h_5 \\ h_3 & h_4 & h_5 & h_6 \\ h_4 & h_5 & h_6 & 0 \\ h_5 & h_6 & 0 & 0 \\ h_6 & 0 & 0 & 0 \end{pmatrix} , \begin{pmatrix} v_3 \\ v_2 \\ v_1 \\ v_0 \end{pmatrix} ,$$

With v reversed in this way, you can check that the product has the indicated structure with the wi's in the middle positions. Thus doing the polynomial product of the $2n-1$ term polynomial whose coefficients are H[i] with the n term polynomial whose coefficients are the v[i] in reverse order gives a product in $O(n \log(n))$ time using FFT and containing the desired w[i]'s.

More precisely write like the following, but get the indices right! Let h be the polynomial given by $h(x) = \sum_{i=0}^{2n-2} H[i]x^i$ and let g be $g(x) = \sum_{j=0}^{n-1} v[j]x^{n-1-j}$. Then the product f = hg satisfies $f_i = \sum_{k=0}^{3n-3} h[k]v[j]$ where $n-1-j = i-k$. Thus $k = -n+1+i+j$. If we substitute $-n+1+i$ for i we have the formula for w[i]. In other words $w[i] = f_{1+i-n}$.

(c) Design and analyze an efficient algorithm for product of two $n \times n$ Hankel matrices. You can get a complexity much better than Strassen's general purpose matrix multiplication algorithm in $O(n^{2.81})$.

Solution: (This works when the second matrix is any matrix, not just a Hankel one.) To compute HB we can compute Hb_i by the method of part (b) for b_i being the i-th column of B. Since there are n columns each costing $O(n \log(n))$, the total cost is $O(n^2 \log(n))$. See "Fast Algorithms for Toeplitz and Hankel Matrices" by Georg Heinig and Karla Rost for more information on algorithms for these structured matrices.

13. [Group Problem — dynagram] Dynagram is a village laid out neatly with numbered streets which run east to west and numbered avenues which run north to south. As a prize to recognize your algorithmic talents, the mayor of Dynagram has a bucket of coins waiting for you at each intersection. The matrix $B[i, j]$ contains the value of the bucket at the intersection of i street with j avenue. The challenge is to make a tour from intersection (0,0) to intersection (m,n) and back to (0,0) collecting the maximal possible prize from the buckets along the way. However you are restricted to go only east and south on the way from (0,0) to (m,n) and go only west and north on the return portion of the trip. Thus until you reach (m,n) the valid moves are to go from (i,j) to (i+1,j) or to (i, j+1). Then from (m,n) onwards the valid moves are to go from (i,j) to (i-1, j) or to (i, j-1). It may or may not be to your advantage to visit some intersection (i,j) both on the way out and on the way back. However you only get one bucket of coins at (i,j)!

Design and analyze an efficient algorithm which, given the payoff matrix B and farthest intersection (m,n), maximizes the prize obtained. What is the asymptotic run time of your algorithm in terms of m and n? What is the amount of memory you need (other than the storage of the given matrix B)?

Solution: Rather than a path that proceeds from 0,0 to m,n and back to 0,0, we will consider two paths emanating from 0,0 and, ultimately, ending at m,n. Without loss of generality we may speak of the lower path and the upper path since two paths, a and b, may cross repeatedly and it does not matter between points of intersection which segment is assigned to path a and which segment to

path b. We will assign all lower segments to one path and all upper segments to the other path.

For $i \leq j \leq k$, let function `best(i,j,k)` give best solution for a pair of paths of length k emanating from 0,0 with the lower path ending at $i, k-i$ and the upper path ending at $j, k-j$. We are keeping the path end points on the same "diagonal" by using k in this way. We want to go from parameters (0,0,0) representing the fact that both paths start at 0,0 to $(n, n, 2n)$ representing the arrival of both paths at n, n. Note that we are solving a more general problem: For each pair of points, a and b, equidistant from 0,0 give the optimal result of a pair of paths, one ending at point a, the other at b.

I'm calling the prize at intersection a, b as `gold(a,b)`.

The first attempt at a recursive solution for best goes like this: best(i,j,k) = gold(i,k-1) + gold(j,k-j) + max(best(i-1,j,k-1), best(i-1,j-1,k-1), best(i,j-1,k-1), best(i,j,k-1)). The idea is to have each path take one step either left or down, so that there are 4 possible previous positions. However, we have not considered the case i = j nor the bogus cases when i-1 or j-1 is less than zero or when k ¡ i or k ¡ j.

The cleaned up recursive solution is as follows:

```
best(i,j,k) {
    // return best solution for lower path from 0.0 to i,k-i and upper path from 0,0 to j,k-j.
    if (i > j or j > k) return 0; // violates required i <= j <= k.
    if (i < 0 or j < 0) return 0; // out of bounds
    if (k = 0) return 0 // base case, necessarily i = j = 0 also.
    if (k < i or k < j) return 0; // out of bounds
    if (i = j) // Both paths arriving at the same point on this diagonal.
        // You get the gold only ONCE here..
        return gold(i,k-i) +
            max{best(i-1,i-1,k-1),
                best(i,i,k-1),
                best(i-1,i,k-1)};
    else // i != j, Paths arriving at distinct points on this diagonal.
        // You get both pots of gold..
        return gold(i,k-i) + gold(j,k-j) +
            max{best(i-1,j-1,k-1),
                best(i-1,j,k-1),
                best(i,j-1,k-1),
                best(i,j,k-1)}
}
```

As an example, suppose we want the best solution to arrive at (4,8),(5,7). We call best(4,5,12), which checks:the 4 possibilities of one step for each path. best (3,4,11) representing the stepping from (3,8),(4,7), best (3,5,11) representing the stepping from (3,8),(5,6), best (4,4,11) representing the stepping from (4,7),(4,7), and best (4,5,11) representing the stepping from (4,7),(5,6). As another example, suppose we want the best solution to arrive at (4,5),(4,5). We call best(4,4,9), which checks:the only 3 possibilities of one step for each path. best (3,3,8) representing the stepping from (3,5),(3,5), best (4,4,8) representing the stepping from (4,4),(4,4), best (3,4,8) representing the stepping from (3,5),(4,4),

The solution to the given problem is then to compute best(n,n,2n).

This recursive best() is correct because it considers all possible ways to arrive at the goal at each snapshot where the two paths are on the same diagonal (have taken the same number of steps), summing the total gold on the two paths, by calculating the optimum solution for all possible snapshots of the paths in which each has taken k steps, for k = 0, 1, ... 2n. The final diagonal has only the one case, the pair of points (n,n),(n,n) represented by the parameters to best of (n,n,2n). Thus we have solved the original problem by solving this more general one.

Since the work outside of recursive calls is constant and there are $O(n^3)$ triples in which $-1 \leq i \leq j \leq k \leq 2n$, we may memoize best() to get a run time

of $O(n^3)$. The memory will also be $O(n^3)$. However if we convert to a bottom up iterative solution, we will only need to store the values for two successive values of k at a time. Thus the memory can be reduced to $O(n^2)$.