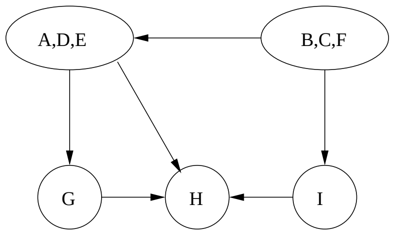
UC Berkeley—CS 170: Efficient Algorithms and Intractable Problems Handout Midterm 1 Solutions
Lecturers: Christos Papadimitriou and Umesh Vazirani

Notes Midterm 1 Solutions for CS 170

Problem 1.

The pre and post numbers should have been the following: A(1,10), B(11,18), C(13,14), D(2,9), E(3,6), F(12,17), G(7,8), H(4,5), I(15,16).

We have 5 strongly connected components:



Problem 2.

1. Since $F_{n+1} = F_n + F_{n-1}$ and $F_{n-1} < F_n$, we can say that $F_{n+1} \equiv F_{n-1} \bmod F_n$. So $gcd(F_{n+1}, F_n) = gcd(F_n, F_{n-1})$. We can repeatedly apply this relationship and get that $gcd(F_{n+1}, F_n) = gcd(F_1, F_0) = gcd(1, 1) = 1$.

2. A number in Z_{1331} has an inverse if and only if it is relatively prime to 1331. The numbers not relatively prime to 1331 are the multiples of 11 (since $1331 = 11^3$). There are $1331/11 = 121$ of them. So there are $1331 - 121 = 1210$ numbers with inverses.

3. We must choose e to be relatively prime to $(p-1)(q-1) = 60$. For convenience, we will choose e to be 7, the smallest we can make e since e cannot be 1. Since $ed \equiv 1 \bmod (p-1)(q-1)$, we must find the inverse of 7 modulo 60. Either through observation or the Extended Euclidean Algorithm, we can find that $7 \cdot 43 = 301 \equiv 1$ mod 60. So we should let $d = 43$.

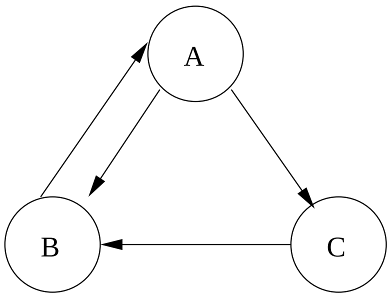
4. By Fermat's Little Theorem, we know $2^{46} \equiv 1 \bmod 47$. Now we need to find 2^{46} mod 46. We previously learned a lemma which stated that $a^{k(p-1)(q-1)+1} \equiv a \bmod pq$. Since $46 = 23 \cdot 2$, we have that $2^{46} \equiv 2^{22 \cdot 2+2} \equiv 2^2 \equiv 4 \bmod 46$. So $2^{2^{46}} \equiv 2^4 \equiv 16$ mod 47.

5. In each function call, we print one line and call the same function twice with input that's half the size. So we have, $T(n) = 2T(\frac{n}{2}) + 1$. We can use the Master Theorem to solve this recurrence. We have that $a = 2$, $b = 2$, and $d = 0$. Since $a > b^d$, we can say $T(n) = \Theta(n^{log_b a}) = \Theta(n)$.

6. We should choose ω to be a primitive fourth root of unity. So $\omega = e^{\frac{\pi i}{2}} = i$ will work. Our polynomial is $P(x) = 1 + x^2 - x^3$. $A_3 = P(\omega^3) = P(-i) = 1 - 1 - i = -i$.

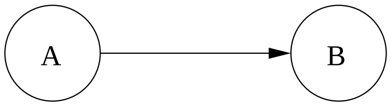
7. What allows us to divide the problem is the fact that our evaluation points (the n n^{th} roots of unity) are in positive-negative pairs. Specifically, $w^{\frac{n}{2}} = -1$. What allows us to keep dividing the problem is that the squares of these points maintain this property. Specifically, if ω is an n^{th} root of unity, then ω^2 is an $(n/2)^{th}$ root of unity. The inverse can also be done with an FFT using ω^{-1} since $1 + \omega + ...\omega^{n-1} = 0$. (If you had all of this, you got more than 5 points).

8. (a) FALSE.



If we begin our search at A, proceed to B, and then to C, the edge (C, B) is a cross edge.

(b) FALSE.



If we begin at A, then the only edge is a tree edge.

Problem 3

First consider the naive algorithm: sequentially check each element. This takes linear time. Our divide and conquer algorithm should try to beat this.

Let us first examine the middle element. If its value matches its index in the original array, we're done. If it is bigger than its index, then every subsequent element will also be bigger than its index since the array values grow at least as fast as the indices. Similarly, if it's less than its index, then every previous element in the array will be less than its index by the same reasoning. So after the comparison, we only need to examine half of the array. We can recurse on the appropriate half of the array. If we continue this division until we get down to a single element and we still have not found our desired element, then it is not possible.

```
Find(A,x) // A is the array, x is the offset from the original array
mid = length(A)/2
if (A[mid] = mid+x) then return TRUE
```

```
    if (length(A) = 1) then return FALSE
    if (A[mid] > mid) then return Find(A[0:mid-1],x)
    else return Find(A[mid+1,length(A)],x+mid+1)
```

We do a constant amount of work with each function call. So our recurrence relation is $T(n) < T(\frac{n}{2}) + O(1)$. We can apply the Master Theorem with $a = 1$, $b = 2$, and $d = 0$. Since $a = b^d$, we have $T(n) = O(\log n)$.