HW3 - S15
Warmuth

Justin Wong (jujwong)

4/21/15

1 Hadamard Matrices

When calculating the dot product, the number of positive terms and negative terms will be the same, thus when adding together the products, the first step of taking a dot product, will be equal to 0.

In the the case when $k = 1$:

$H_1 = \begin{bmatrix} \mathrm{H}_0 \, H_0 \\ \mathrm{H}_0 \, H_0 \end{bmatrix} = \begin{bmatrix} 11 \\ 1-1 \end{bmatrix}$ where the dot product is (1*1)+(1*-1) = 1 + -1. There is 1 positive product, 1, and 1 negative product, -1.

Let k $\geq$ 1, assume that the dot product of two rows of a Hadamard matrix is 0. In each successive matrix of size 2^k vs size 2^{k+1}, the the number of total elements in each row will double, but the ratio of positive and negative elements in each row will be the same as in the k size matrix. The uniqueness of the Hadamard matrix guarantees that the matrices will all have the same pattern of $\begin{bmatrix} ++ \\ +- \end{bmatrix}$ all of which can be reduced down to the smaller cases. As a result of Hadamard matrices being filled with the same value, just positive or negative, the terms will end up cancelling each other out. As in the 2x2 matrix example, where it there was 1 positive product, 1 negative product of the same value.

2 Prove that the Cashier's Algorithm is optimal for all amounts $x \in \mathbb{N}$ given the above set of coins for any $k, n \in \mathbb{N}$ s.t. $k \geq 2$, $n \geq 0$.

Assume the set of coins for any $k, n \in \mathbb{N}$ s.t. $k \geq 2$, $n \geq 0$. Let x = the amount of money required for coin-changing. The optimal way to make change is $c_k \leq x$ ¡ c_{k+1} where the algorithm will take coin k. The optimal solution will take coin k, if not than it needs to take the coins that add up to but don't exceed k. Thus using the x - c_k the Cashier's algorithm will simplify the problem.

For each coin k^j the algorithm will take the most value it can before exceeding the value to be returned. So if the algorithm does not take coin k^j it will take the value of smaller coins until it adds up to just before k^j.

When k = 1, c_k = 1 it is the smallest coin possible. As long as the value of x < c_2 then it will take an amount of c_1 coins until < c_2. This pattern will continue as $c_k < c_{k+1}$. Finding the maximum value of the coins leading up to coin k^j is in the set of coins $\{1, k, k^2, ..., k^{j-1}\}$. The maximum value using only coins smaller than k^j is seen through the use of the geometric sum of $\frac{k^j-1}{k-1}$. The value of the j^th coin in found with $\frac{k^{j+1}-1}{k-1}$, which is the sum of all the previous coins and the addition of one more larger coin which will always be greater than the rest of the coins added together. In the example where $k = 2, n = 5$: the set is $\{1, 2, 4, 8, 16, 32\}$ which sums to 63, but when $n = 6$ the new largest value is 64 larger than all the previous numbers summed.

3 Unbounded Array Array A: $A[1], A[2], A[3], \ldots]$ if k is in it. Run in O(logp) where p is the number of integers in the array ¡ k.

We start by using something similar to binary search, for the inputs low and high will be equal to 1. While A[high] ¡ k, then high = high*2. Double the search range each time if the value at the index is smaller. Run binary search(low, high).

In binarySearch calculate the mid as $low + (high - low)/2$.

While(high $\geq$ low)

- if $A[mid] > k$ return binarySearch(low, high-1)
- else if $A[mid] < k$ return binarySearch(low, high*2)
- else return mid
- return K not here

The run-time is O(logp) because each time the array grows in size, but the binary search allows us to reduce that as soon as we find a possibility for the value of the index to match k. As soon as 2*i goes past the value of k, then we stop increasing and start reducing down our search. At each step we only have to compare A[mid] with k. Since p will be the index of the element where k is, it takes O(logp) time.

4 Dominant Object in array

(a) $O(n \log n)$

Take the array A and split it into two sub-arrays, A_1 and A_2. If A has a majority then it must be true that either there is a majority of that element in A_1, A_2 or both, then there are 2 cases:

1 If both A_1 and A_2 have the same majority then the majority is of that element.

2 If either A_1 or A_2 have a majority then we need to count which element it is, and that is done by counting the number of times that element repeats in both sub-arrays. However in the case that both A_1 and A_2 have a majority we will need to count for each candidate.

A recurrence is found for this by splitting into 2 sub-problems both with size of n/2. $T(n) = 2T(n/2)+O(n)$, the counting is done in $O(n)$ time. Thus the runtime is $O(nlogn)$ by the Master Theorem.

(b) $O(n)$ When $n = 0$ there is no majority, because there are no elements. Beginning with majority-count = 0 and count = 1, we loop through the array.

- If a[maj-count] == a[i] then we increment count by 1.
- Else decrement count by 1.
- If count is 0, then change the maj-count to the element currently on and set count to 1 and continue through the array.
- return a[maj-count]

To check if maj-count is actually a majority we can check if it is $\geq \lfloor$ n/2 $\rfloor$ + 1

Iterating through the array will only take O(n) work as it goes through once. Checking the maj-count is also done in O(n) time because it is just a comparison of values. Therefore this algorithm runs in O(n) + O(n) reducing to O(n) time.

5 24 Hour Job Processing Using Interval Scheduling

This algorithm uses the same interval scheduling of earliest finish time. However since it is a 24-hour cycle we need to create a start time as the optimal schedule must have a starting point. We will take each job's start time and use that as a start time for each and use the interval scheduling algorithm to find the maximum optimality.

We must run the interval scheduling algorithm n times on each on the n jobs so this runs at O(n^2).

6 KT, problem 4, p 190

We have S of size n and S' of size m. We will traverse through both of the arrays to check if there is a match which we will store that value at another sequence c. If there is a match then, add the found value to c and increment both indexes for S and S'. If there is not a match then only move i up, and again compare S_i with S'_j. When S' it at its end then return the sequence that c was storing. If there are no matches between S and S' then nothing will be returned, as index at j will stay the same while the index at i will eventually compare every value in S to the first value at S'.

Initialize i and j = 1. While i ≤ n and j ≤ m if $S_i = S'_j$ then let $c_j = i$ and increase both i and j by 1. else only increase i by 1. End when both i and j reaches the end of their sequences then return the sequence that c was holding.

There are at most n iterations from S being size n. Each time the loop is run, the work done is only O(1). Thus the runtime is O(n).