CS 670 Spring 2015 - Solutions to HW 1

Problem 2.3-7

Algorithm:
Search_for_sum(S, x)
1. Sort S
2. for$(i = 0\ ;\ i < n\ ;\ i++)$
3. j=binary_search$(S, x - S[i])$
4. if$(j \neq -1)$
5. print $S[i], S[j]$
6. exit
7. print "No such sum exists"

Here, binary_search(A, y) is a subroutine that searches for y in the array A. If the search is successful, an index j such that $A[j] = y$ is returned. Otherwise -1 is returned.

Running Time: Using merge sort, step 1 can be done in $\Theta(n \log n)$. The iteration step is performed through at most n elements. Binary search has running time $O(\log n)$. Hence, the running time of the algorithm is $\Theta(n \log n)$.

Problem 3.2-5

Observe that $\lg^*(\lg(n)) = \lg^*(n) - 1, \forall n > 1$. Clearly, $\lg^*(n) - 1$ grows asymptotically faster than $\lg(\lg^*(n))$. Thus $\lg^*(\lg(n))$ grows asymptotically faster than $\lg(\lg^*(n))$.

Problem 4-5

(a) Lets divide the set of chips into G and B where G is the set of good chips and B is the set of bad chips. We already know the behavior of chips in G. Lets assume that all the sets in B collude to behave in a way that B is a set of good chips and G being the set of bad chips. This can be done in the following way, any chip in B:

1. declares any chip in B as good, and,
2. declares any chip in G as bad.

So, essentially we have a situation where one group declares itself as good and another group declares itself as good and the other bad. Without the information about the relative size of the groups, it is impossible to say that which group is lying. Note that we don't know whether G is bigger than B or otherwise. On the other hand, in case more than half of the chips are good, then depending on what the majority of the chips in the pairwise tests say about a particular chip, we can proclaim the chip to be either good or bad.

(b) Divide the set of chips into $\frac{n}{2}$ pairs and let the chips in each pair test each other (one chip will be unmatched and put aside if n is odd). Now,

1. For each pair which result in a test implying that at least one of them is bad, remove both the chips.
2. The remaining pairs all result in good-good tests, which implies that either both of them are good or both of them are bad. For each such pair we throw out one of the chips from the pair.

Note that in the remaining pile we still have more good chips than bad chips. Note also that at least one chip from each pair is removed. Hence we have reduced the problem size by nearly half.

(c) Note that if we have one good chip, then we can use it to test other chips using $\Theta(n)$ tests. So, the problem reduces to finding just one good chip. We shall use the strategy discussed in (b) for this purpose. Let $T(n)$ be the number of tests required to find one good chip amongst n chips. Then the above strategy yields the following recurrence:

$$T(n) = T(n/2) + \Theta(n)$$

Hence, $T(n) = \Theta(n)$ (Master's Theorem).

Problem 4-6

(a) If an array is Monge it trivially follows that

$$A[i, j] + A[i + 1, j + 1] \leq A[i, j + 1] + A[i + 1, j]$$

Now, lets prove the only if part. We can argue our case by induction on rows and columns separately. Induction on rows is shown here, a similar argument would work for columns. The base case is already given by the above inequality. Induction hypothesis:$\forall x < k$,

$$A[i, j] + A[i + x, j + 1] \leq A[i, j + 1] + A[i + x, j]$$

Taking $x = k - 1$, it follows from the above,

$$A[i, j] + A[i + k - 1, j + 1] \leq A[i, j + 1] + A[i + k - 1, j]$$

Taking $i = i + k - 1$ and $x = 1$, it follows that

$$A[i + k - 1, j] + A[i + k, j + 1] \leq A[i + k - 1, j + 1] + A[i + k, j]$$

Combining the above two inequalities we get,

$$A[i, j] + A[i + k, j + 1] \leq A[i, j + 1] + A[i + k, j]$$

Hence, the induction hypothesis holds.

(b) Replace 7 on second row and third column by 4.

(c) We prove by contradiction. Let k be the smallest row such that $f(k) > f(k + 1)$. So $A[k, f(k + 1)]$ appears before $A[k, f(k)]$ on row k. But since $A[k, f(k)]$ is the leftmost minimum element on row k we have $A[k, f(k + 1)] \geq A[k, f(k)]$. But $A[k, f(k + 1)]$ cannot be equal to $A[k, f(k)]$ (otherwise $f(k + 1)$ is the index of the leftmost minimum element on row k, that is $f(k) = f(k + 1)$ a contradiction). Hence $A[k, f(k)] < A[k, f(k + 1)]$.

Since $A[k+1, f(k+1)]$ is the leftmost minimum element on row $k+1$ and $A[k+1, f(k+1)]$ appears before $A[k + 1, f(k)]$ on row k we have $A[k + 1, f(k + 1)] \leq A[k + 1, f(k)]$.

Combining the two inequalities we get,

$$A[k, f(k)] + A[k+1, f(k+1)] < A[k, f(k+1)] + A[k+1, f(k)]$$

This is a contradiction since A is a Monge array and $f(k) > f(k+1)$.

(d) The recursive solution to the problem of computing f is based on the observation that if A is a Monge array and A' is a subarray obtained by selecting a subset of rows of A, then A' is also Monge. Assuming we already computed the $f(i)'$s for the even i's, we can compute each odd index $f(2k+1)$ by observing that $f(2k) \leq f(2k+1) \leq f(2k+2)$ by the previous problem. Hence in order to compute $f(2k+1)$, we only need to check entries from $A[2k+1, f(2k)]$ to $A[2k+1, f(2k+2)]$. Therefore, the required computation can be done in

$$f(2) + \sum_{i=1}^{\frac{m}{2}} (f(2i+2) - f(2i) + 1) = f(2) + f(m) - f(2) + m \leq m + n$$

because $f(m) \leq n$. The term $f(2)$ appears as $f(2)$ entries have to be checked to compute $f(1)$. Hence the worst-case running time is $O(m+n)$.

(e) Let $T(m; n)$ be the time required for the algorithm. Then we have the following recurrence:

$$T(m; n) = T(\frac{m}{2}; n) + m + n = T(\frac{m}{4}; n) + \frac{m}{2} + n + m + n$$

$$= T(\frac{m}{8}; n) + \frac{m}{4} + \frac{m}{2} + m + 3n = \ldots = O(m + n \log m)$$

Problem 6.5-9

Algorithm.
1. Let the sorted arrays be: $A_1[1 \ldots n_1], \ldots, A_k[1 \ldots n_k]$.
2. Let A be the output array. Let $i = 1$.
3. Let $a_1 = a_2 = \ldots = a_k = 1$.
4. Let $S[1 \ldots k]$ be an array with $S[i] = A_i[a_i]$.
5. while(there are elements in S)
6. Heapify(S)
7. j = Heap Extract Min(S)
8. $A[i++] = A_j[a_j]$
9. if($a_j < n_j$)
10. a_j++
11. Heap Insert($A_j[a_j]$)

The while loop is the most time consuming step. The while loop is executed n times and at each loop it takes $O(\log k)$ time. Hence, the time complexity of the procedure is: $O(n \log k)$.

Problem 6-3

(a)

$$\begin{bmatrix} 2 & 3 & 4 & 5 \\ 8 & 9 & 16 & \infty \\ 12 & 14 & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

(b) It suffices to observe that $Y[1,1]$ contains the smallest element and $Y[m,n]$ contains the largest element in the tableau.

(c) Algorithm.
1. EXTRACT_MIN($Y[1 \dots m, 1 \dots n]$)
2. min=$Y[1,1]$
3. $Y[1,1] = Y[m,n]$
4. $Y[m,n] = \infty$
5. MAKE_TABLEAU($Y[1 \dots m, 1 \dots n]$)
6. return min

Algorithm.
1. MAKE_TABLEAU($Y[1 \dots m, 1 \dots n]$)
2. $Y[a,b] = min(Y[1,2], Y[2,1])$
3. if $Y[a,b] < Y[1,1]$
4. swap($Y[a,b], Y[1,1]$)
5. MAKE_TABLEAU($Y[a \dots m, b \dots n]$)

Note: To keep things simple, trivial cases like the tableau having single row, or single column, or both haven't been dealt here.

Time Complexity: Let $T(m,n)$ be the time required for the algorithm for an $m \times n$ tableau. Also, let $p = m + n$ and $S(p) = T(m,n)$. Then both the cases in the function reduce to the following recursion:

$$S(p) = S(p-1) + 1 = S(p-2) + 1 + 1 = S(p-3) + 3 = \ldots = O(p) = O(m+n)$$

(d) Algorithm.
1. INSERT($Y[1 \dots m, 1 \dots n], x$)
2. if $y[m,n] < \infty$
3. print "Overflow" and exit
4. $Y[m,n] = x$
5. $Y[a,b] = max(Y[m,n-1], Y[n-1,m])$
6. if $Y[m,n] < Y[a,b]$
7. swap(Y[m,n],y[a,b])
8. INSERT($Y[1 \dots a, 1 \dots b], x$)

Time Complexity: A similar analysis as in the previous problem would show that this function takes $O(m+n)$ time.

(e) We can use a $n \times n$ tableau. n^2 elements can each be inserted in $O(n)$ time (since $m = n$). So a total of $O(n^3)$ time and then using extract min n^2 times (each $O(n)$) the elements can be obtained in sorted order in $O(n^3)$ time.

(f) Algorithm.
1. SEARCH($Y[1 \dots m, 1 \dots n], x$)
2. if $Y[m,1] = x$
3. return$(m,1)$
4. else if $x < Y[m,1]$
5. return SEARCH($Y[1 \dots m-1, 1 \dots n], x$)
6. else
7. return SEARCH($Y[1 \dots m, 2 \dots n], x$)
8. return NULL

Time Complexity: Again it is fairly easy to see that the time complexity of this function is $O(m+n)$.

Problem 8-4(b)

For convenience, let us arbitrarily fix a labeling of the set of red jugs as $R := \{r_1, r_2, \ldots, r_n\}$ and the set of blue jugs as $B := \{b_1, b_2, \ldots, b_n\}$.

The input defines a one-to-one onto function $\pi : R \to B$ where $b_j = \pi(r_i)$ if and only if r_i and b_j have the same volume. There are at least $n!$ choices for π.

Assume that A is an algorithm that can determine underlying function π.

A can only perform a sequence of operations where each operation is of the following form: Pick i and j in the range $\{1, 2, \ldots, n\}$, compare the volumes of r_i and b_j and infer one of the three possible outcomes. Thus A can be identified with a complete ternary (each internal vertex can have three children) decision tree where each internal vertex corresponds to a comparison and each leaf vertex corresponds to a one-to-one and onto function from R to S that can be inferred from the outcomes of the comparisons corresponding to the ancestors of that leaf.

The worst case running time of the algorithm corresponds to the height of the decision tree.

Since there are at least $n!$ choices for π, the decision tree has to have at least $n!$ leaf vertices. Let h be the height of our decision tree. Being a ternary tree, it has at most 3^h leaves. Thus

$$3^h \geq n! \Rightarrow h \geq \log_3(n!) = \Omega(n \log n)$$

where the last equality follows from Stirling's formula for the factorial.