CSC373 Fall 2017 Assignment 1

1. You are given an array A of n complex numbers. Some of these numbers might be identical. For $i \neq j \in [n]$ you can invoke a comparison procedure that returns whether the numbers A[i] and A[j] are identical or not. Design a divide and conquer algorithm to decide whether there is a number in the array that appears more than $\lceil n/3 \rceil$ times in A using $O(n \log n)$ invocations of the comparison procedure.

   a) In English, state the high level idea of your algorithm.
   - **If A has an element i, such that it appears more than $\lceil n/3 \rceil$ times in A, then i must appears more than $\lceil n/6 \rceil$ times in either the left or the right half of the array A.**
   - **So i must also appears more than $\lceil n/3 \rceil$ times in either one or both of these two arrays.**
   - **Running time $T(n) = 2\,T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$.**

   b) Describe your algorithm in pseudocode.

```
Input: an array $A$ of $n$ complex numbers
Output: True (there exists a number in $A$ s.t. it appears more than $\lceil n/3 \rceil$ times)
        or False (there is no number in $A$ that it appears more than $\lceil n/3 \rceil$ times)
CheckIfMoreThanN/3($A[a_1, a_2, \dots, a_n]$)
    if size of $A$ = 1:
        return $A[0]$
    mid = $n/2$                    ← since we can assume $n = 2^k$, $n$ is divisible by 2
    target = $\lceil n/3 \rceil$
    CheckIfLeft = CheckIfMoreThanN/3 ($A[a_1, \dots, a_{mid}]$)
    CheckIfRight = CheckIfMoreThanN/3 ($A[a_{mid+1}, \dots, a_n]$)
    if CheckIfLeft == CheckIfRight:
        return CheckIfLeft
    if CheckIfLeft appears more than target times in the right half of the array:
        return majority_left
    if CheckIfRight appears more than target times in the left half of the array:
        return majority_right
    return False
```

   c) Provide a correctness argument for your algorithm.
   - **Prove: if A has an element i, s.t., it appears more than $\lceil n/3 \rceil$ times in A, then i must appears more than $\lceil n/3 \rceil$ times in proceeding iterations .**
   - **Let $x_A = \lceil n/3 \rceil$ meaning the number of times that i appears in an array A with size n**
   - **Let $x_B = \lceil n/3 \rceil$ meaning the number of times that j appears in an array B with size m**
   - **We need to prove that $x_B > m - x_B$**
   - **There can be three possible scenarios between x_A and x_B:**
     - i. **i appears $\geq x_A$, j appears $\geq x_B$, say possibility to happen is w**
     - ii. **i appears $\geq x_A$, j appears $< x_B$, say possibility to happen is $x_A - 2w$**
     - iii. **i appears $< x_A$, j appears $< x_B$,**
   - **Then the possibility of iii to happen is $n/3 - w - (x_A - 2w) = n/3 + w - x_A$**
   - **Then situation ii will not appear B and it will have at most 2 elements from situation i,**
   - **In the worst-case, B will have 2 elements that would cause situation iii.**
   - **Consider the worse case, $m = w + 2(n/3 - x_A + w)$**
   - **Then $m - w = 2n/3 - 2x_A + 2w$ which means $x_B > m - x_B$**

d) State the recurrence that describes the number of invocations of the comparison procedure.

- **Let $T(n)$ be the number of invocations of comparisons in CheckIfMoreThanN/3(A).**
- **Then** $T(n)\begin{cases} = 0 & , n = 1 \\ \leq 2\,T\left(\frac{n}{2}\right) + O(n), & n > 1 \end{cases}$
- **Then** $T(n) = 2\,T\left(\frac{n}{2}\right) + O(n)$
- **Since** $a = 2, b = 2$
- **Then** $n^{\log_b a} = n,$ **and** $f(n) = n$
- **Since the Master theorem gives that** $T(n) = O\left(n^{\log_b a}\log n\right)$
- **Then** $= O(n \log n)$

2. You are given two sorted lists A_1 and A_2, each with n real numbers. Assume all numbers in $A_1 \cup A_2$ are distinct. Design a divide and conquer algorithm that returns the median of $A_1 \cup A_2$ using $O(\log n)$ comparisons that test whether or not $x \leq y$. The median of m numbers $x_1 \leq x_2 \ldots \leq x_m$ is $x_{\lceil m/2 \rceil}$.

a) In English, state the high level idea of your algorithm.

- **Calculate the midpoints of the two arrays; we can name the midpoints $m_1$and m_2**
- **If $m_1$and m_2 are the same, then we can just return one of them; that means $m_1$and m_2 will still be the midpoint if the two arrays combined.**
- **If $m_1 > m_2$, we check the first element of $A_1$to m_1, and m_2 to A_2's last element.**
- **If $m_1 < m_2$, we check the first element of $m_1$to the last element of A_1, and the first element of A_2 to m_2.**
- **This process will repeat until the size of the arrays left are less or equal to 2; by then we reached our base cases.**

b) Describe your algorithm in pseudocode.

```
getMedian($A_1[a_{11}, a_{12}, \ldots, a_{1n}]$, $A_2[a_{21}, a_{22}, \ldots, a_{2n}]$, $n$)
    if $n \leq 0$:
        return Error
    if $n == 1$:
        return $(A_1[0] + A_2[0])/2$
    if $n == 2$:
        return (max($A_1[0], A_2[0]$) + min($A_1[1] + A_2[1]$))/2

    $m_1$ = the median of the first array
    $m_2$ = the median of the second array

    if $m_1 == m_2$:
        return $m_1$
    elif $m_1 < m_2$:
        if n is divisible by 2:
            return getMedian($A_1[a_{1m-1}, \ldots, a_{1n}]$, $A_2[a_{21}, \ldots, a_{2m}]$, $n - \frac{n}{2} + 1$)
        return getMedian($A_1[a_{1m}, \ldots, a_{1n}]$, $A_2[a_{21}, \ldots, a_{2m}]$, $n - \frac{n}{2}$)
    else
        if n is divisible by 2:
            return getMedian($A_1[a_{11}, \ldots, a_{1m-1}]$, $A_2[a_{2m}, \ldots, a_{2n}]$,, $n - \frac{n}{2} + 1$)
        return getMedian($A_1[a_{11}, \ldots, a_{1m}]$, $A_2[a_{2m}, \ldots, a_{2n}]$,, $n - \frac{n}{2}$)
```

c) Provide a correctness argument for your algorithm.

I do not know how to answer this question.

d) State the recurrence that describes the number of invocations of the comparison procedure.

- **Let $\mathbf{T(n)}$ be the number of invocations of comparisons in getMedian(A_1, A_2, n).**
- **Then** $\mathbf{T(n)} \begin{cases} = \mathbf{0} & , \mathbf{n \le 2} \\ \le \mathbf{2\,T\left(\frac{n}{2}\right) + O(1)}, & \mathbf{n > 2} \end{cases}$
- **Then** $\mathbf{T(n) = 2\,T\left(\frac{n}{2}\right) + O(1)}$
- **Since** $\mathbf{a = 2, b = 2}$
- **Then** $\mathbf{n^{\log_b a} = n,\ and\ f(n) = 1}$
- **Since the Master theorem gives that** $\mathbf{T(n) = O\left(n^{\log_b a} \log n\right)}$
- **Then** $\mathbf{= O(\log n)}$

3. Consider the following question concerning the MST problem where the input is a connected graph $G = (V, E)$ with edge costs $c: E \to \mathbb{R}$.

a) Let $e = (u, v) \in E$ be an arbitrary edge. We wish to compute a MST $T = (V, E')$ for G subject to the constraint that $e \in T$. Consider the following divide and conquer type algorithm.

- Partition the vertices V into two sets V_1 and V_2 having sizes $\lceil n/2 \rceil$ and $\lceil n/2 \rceil$ such that $u \in V_1$ and $v \in V_2$.
- Compute a minimum spanning tree or minimum spanning forest $T_1 = (V_1, E_1)$ for the graph induced by V_1 and $T_2 = (V_2, E_2)$for the graph induced by V_2.
- Return $T = (V, E_1 \cup E_2 \cup \{e\})$

Is the solution T necessarily a spanning tree? If T is a spanning tree, is it necessarily a minimum spanning tree?

- **The solution T would be a spanning tree, but is not necessarily a minimum spanning tree.**
- **Since both T_1 and T_2 are MSTs, and all the nodes in both MSTs are connected, adding a new edge meaning that $e = (u, v)$ is connect to V_1first then can be reached by V_2 through u to v.**
- ***T* is not necessarily a minimum spanning tree, because let us assume that there is a connection *c* between V_1 and V_2, where its value is less than of the edge *e*. Since that *e* is already an edge, then *c* cannot be an edge as well. But we can replace *e* with *c* to reduce the total cost of the tree.**

b) Consider the constrained MST problem as above. Design a greedy algorithm for this problem. Give an informal but convincing argument as to why your algorithm computes an optimal spanning tree subject to the constraint.

I do not know how to answer this question.

c) Returning to the standard (unconstrained) MST problem, consider the case when all the edge costs are distinct. Is the MST unique? If so, provide a brief argument; otherwise, provide a counter-example.

- **Yes, if all the edges are having different edge value, that means that there is no way this MST will be having two same values in weights, which is the definition for unique MST.**

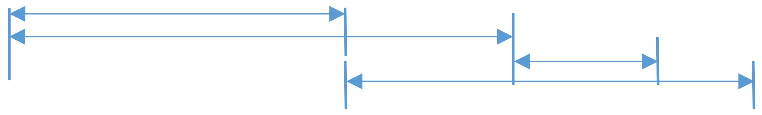
4. Recall the EFT algorithm for interval scheduling on one machine. We wish to extend this algorithm to m machines. That is, when considering the ith interval I_j we will schedule it if there is an available machine. But on which machine? That is, what is the tie breaking rule?

   a) Consider the First-Fit EFT greedy algorithm:

```
• Sort the intervals $\{I_j\}$ so that $f_1 \le f_2 \dots \le f_n$
• For $j = 1 \dots n$
      For $i = 1 \dots m$
          If $I_j$ fits on $M_l$ then schedule $I_j$ on $M_l$
      EndFor
  EndFor
```

   Is First-Fit an optimal algorithm? If yes, provide a proof; otherwise provide a counter-example.

   - **The first-fit EFT is not an optimal algorithm, here is a counter example:**



   - **In this case, First-Fit cannot pick up the two jobs that would be the more profitable solution.**

   b) Consider the Best-Fit EFT greedy algorithm:

```
• Sort the intervals $\{I_j\}$ so that $f_1 \le f_2 \dots \le f_n$
• For $j = 1 \dots n$
      If there is a machine $M_l$ on which $I_j$ can be scheduled, then
          schedule $I_j$ on $M_l$ where $l = argmax_k\{f_k \le S_j$ and $I_k$ scheduled on $M_k\}$
  EndFor
```

   Is Best-Fit an optimal algorithm? If yes, provide a proof; otherwise provide a counter-example.

   - **The best-fit solution would be an optimal algorithm, since it will not cause extra and repetitive comparisons to move items into machines.**
   - **Assuming that best-fit is not optimal,**
   - **And let k be the first**

5. You are given two arrays D (of positive integers) and P (of positive reals) of size n each. They describe n jobs. Job i is described by a deadline D[i] and profit P[i]. Each job takes one unit of time to complete. Job i can be scheduled during any time interval $[t, t+1)$ with t being a positive integer as long as $t + 1 \le D[i]$ and no other job is scheduled during the same time interval. Your goal is to schedule a subset of jobs on a single machine to maximize the total profit - the sum of profits of all scheduled jobs. Design an efficient greedy algorithm for this problem.

   a) Describe your algorithm in plain English.

   - **Sort all jobs in decreasing order of profit.**
   - **Initialize an empty list with the first job in the rank, which will be returned later as the result.**

- **For each of the jobs, if it can fit into the current working list and it can be finished before its deadline, add this job to the return list.**
- **Else, ignore and move on.**

b) Describe your algorithm in pseudocode.

```
schedulJobs($D$, $P$, $n$)
    sort all jobs according in decreasing order of profit
    sort the corresponding $D$ as well
    workingList = [$D[0]$]
    for $i$ from 0 to $n - 1$:
        deadline = D[i]
        time = (time of $D[i]$) $- 1$        # when do we plan to start on job $D[i]$
        for $j$ from 0 to sizeof(workingList)-1:
            if scheduled time of workingList[j] == time:
                time = time $-1$        # shift early one hour to see if that space is available
        if there are space left between now and the required deadline:
            workingList.append(D[i])
```

c) Prove correctness of your greedy procedure. One possible proof is to argue by induction that the partial solution constructed by the algorithm can be extended to an optimal solution. But you can use any type of valid proof argument.

- **Assume that the jobs have been sorted in the decreasing order of profit**
- **Let l_i be the currently working list from the last iteration i, for $0 \leq i \leq n$**
- **We can proof through Induction:**
- **Base case: let $i = 1$, then $l_i = [\,]$, since it would be empty**
- **Induction Hypothesis: assume that l_i is approaching the optimal solution l^* with jobs from i to n,**
- **consider the current iteration i**
- **Case 1: i does not fit for the list l_i**
  - **Then all the time between now and i's deadline is taken in l_i**
  - **Since l_i is approaching l^*, such that l^* contains l_i**
  - **Then all the time between now and i's deadline is taken in l^* as well**
  - **Then i does not fit for the list l^* as well**
  - **Then l_{i+1} is still approaching l^***
- **Case 2: i does fit for the list l_i**
  - **Then there is time between now and i's deadline in l_i**
  - **Since l_i is approaching l^*, such that l^* contains l_i**
  - **Since if there is space and i is the next in line**
  - **We will put i in the l_{i+1}**
  - **Then l_{i+1} is still approaching l^***
- **Thus, l^* is the optimal solution to the problem with l_{i+1}**

d) Analyze the running time of your algorithm.

- **Runtime would be the 2 nested sorting times and the 2 for-loops**
- $n^2 + 2(n \log n) = O(n^2)$

6. Consider the following $\{0, 1, 3\}$ knapsack problem. You are given a knapsack size bound S and n items $\{(S_1, V_1), \dots (S_n, V_n)\}$. A feasible solution σ is a sequence $< C_1, \dots, C_n >$ in $\{0, 1, 3\}^n$

such that $\sum_i c_i s_i \leq S$ and the value of such a solution is $V(\sigma) = \sum_i c_i v_i$. That is, every item in the knapsack occurs exactly once or three times. Give a polynomial time dynamic programming algorithm for optimally solving this problem assuming each size s_i is a positive integer and the knapsack bound $S \leq n^3$.

a) Describe the semantic array.
   - **Because we are dealing with a knapsack problem, the semantic array would be:**
   - **$V[i, S]$ = max(profit by a set of the first i items, where $0 \leq i \leq n$), but make sure that it will not go over the bound S.**

b) Describe the computational array. Don't forget the base case.
   - $$\mathbf{V'[i, S]} = \begin{cases} \mathbf{0}, & \textbf{if } \mathbf{i = 0} \textbf{ or } \boldsymbol{S = 0} \\ \mathbf{max}\{\boldsymbol{V'[i-1, S - js_i] + jv_i, j = 0, 1, 3}\}, & \textbf{if } \boldsymbol{s_i \leq} \mathbf{S} \end{cases}$$

c) Justify why the above two arrays are equivalent.
   - **The desired optimum value is $\boldsymbol{max\{V'[i-1, S - js_i] + jv_i\}}$, at this moment, it does not matter if $\boldsymbol{s_i = 0}$ or $\boldsymbol{s_i > 0}$.**
   - **Therefore, the semantic array will be equivalent as the the computational array.**

d) What is the running time of your algorithm in terms of n.
   - **For each of the $\boldsymbol{V'}$, it only requires constant runtime $\boldsymbol{O(1)}$ to compute its iteration before**
   - **The size of the array is at most S x (n items)**
   - **Since $\boldsymbol{S \leq n^3}$**
   - **We can assume that $\mathbf{(1)}$ x $\boldsymbol{S}$ x ($\boldsymbol{n}$ items) = $\boldsymbol{n^3}$ x ($\boldsymbol{n}$ items) = $\boldsymbol{O(n^4)}$**

7. You are given an array A of $n \geq 3$ points in the Euclidean $2D$ space. The points $A[1], A[2], \ldots, A[n]$ form the vertices (in clockwise order) of the convex polygon P. A triangulation of P is a collection of $n-3$ interior diagonals of P such that the diagonals do not intersect except possibly at the vertices. The total length of a triangulation of P is the sum of the lengths of the $n-3$ interior diagonals used to form that triangulation. Design an efficient dynamic programming algorithm to find the total length of a triangulation with the minimum total length.

a) Describe the semantic array.
   - **$\boldsymbol{V[i, j]}$ = min(total lengths of the triangulations $\boldsymbol{P_1, \ldots, P_j}$)**

b) Describe the computational array. Don't forget the base case.
   - $$\boldsymbol{V[i, \mathrm{j}]} = \begin{cases} \mathbf{0}, & \boldsymbol{if\ j = \mathrm{i}} \\ \boldsymbol{min\{V[i,k] + V[k+1, \mathrm{j}] + timeToCompute[i-1, \mathrm{k}, \mathrm{j}]\}, \mathrm{i} \leq \mathrm{k} < \mathrm{j}} & \textbf{otherwise} \end{cases}$$

c) Justify why the above two arrays are equivalent.
   - **The optimal triangulation has to have one side of its triangle from the polygons of $\boldsymbol{A[i-1], \ldots, A[j]}$, one side from , and other side from $\boldsymbol{A[k], \ldots, A[j]}$ and another side of $\boldsymbol{A[k], \ldots, A[i-1]}$**
   - **Therefore, the semantic array will be equivalent as the the computational array, and other side from $\boldsymbol{A[k], \ldots, A[j]}$**

d) What is the running time of your algorithm in terms of n.

- **The computational array yields a runtime of n^2 possibilities for getting the optimal minimum length of the triangulation.**
- **To put each of the possibilities into the testing array/temp output will take n time.**
- **Then $\left(n^2 \text{ items}\right)$ x $n = O(n^3)$**