Solutions to Problem Set 1

1. (Description on the Problem Set)

Unless stated, log is always base 2.

(a) Using the Master Theorem case 3. Note that
\[ f(n) = n \log n = \Omega(n) = \Omega(n^{\log_4 2 + 1/2}) = \Omega(n^{\log_b a + \epsilon}), \quad \epsilon = 1/2 > 0 \]
Moreover, \( f(n) \) satisfies the regularity condition. In fact
\[ af(n/b) = \frac{2^n}{4} \log \frac{n}{4} \leq \frac{1}{2} n \log n = cf(n), \quad c = 1/2 < 1 \]
Thus, \( T(n) = \Theta(f(n)) = \Theta(n \log n) \).

(b) Using the Master Theorem case 2. Note that
\[ f(n) = \sqrt{n} = \Theta(n^{1/2} \cdot 1) = \Theta(n^{\log_b a \cdot \log k n}), \quad k = 0 \]
Thus, \( T(n) = \Theta(n^{\log_b a \log (k+1) n}) = \Theta(n^{1/2 \log n}) \).

(c) Using the Master Theorem case 1. Note that
\[ f(n) = n = O(n^{\log 3 - 0.1}) = O(n^{\log_b a - \epsilon}), \quad \epsilon = 0.1 > 0 \]
Thus, \( T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log 3}) \).

(d) We study the recursion tree. The recursive formula goes like:
\[
T(n) = T(2n/3) + T(n/3) + n/6 \\
= T(2n/9) + T(4n/9) + \frac{1}{6}(2n/3) + T(n/9) + T(2n/9) + \frac{1}{6}(n/3) + n/6 \\
= T(n/9) + 2T(2n/9) + T(4n/9) + 2n/6 \\
= T(n/27) + 3T(2n/27) + 3T(4n/27) + T(8n/27) + 3n/6 \\
= \text{etc...}
\]
At each complete level of the recursion tree (note the structure of the Pascal triangle), an additional \( \frac{1}{6} n \) operations are performed. Unfortunately, the tree is unbalanced: the most left path from the root to a leaf will be the shortest, while the most right path will be the longest. Thus, not every level is contributing exactly \( \frac{1}{6} n \). By looking at the most right path (the longest) which is of the form \( T(\frac{2^i}{3} n) \) at level \( i \), we can compute the height of the tree using \( \frac{2^i}{3} n = 1 \) which is \( \frac{3^i}{2} = n \) and,
solving by $i$, we have $i = \log_{3/2} n = \text{height}$. The intuition is that incomplete levels performs less than $\frac{1}{6}n$ operations, thus we are expecting a $O(n \log n)$ bound. We formally prove this by using the Substitution Method.

$$T(n) = T(2n/3) + T(n/3) + n/6$$

$$= d \left( \frac{n}{3} \right) \log \frac{n}{3} + d \left( \frac{2n}{3} \right) + \frac{n}{6}$$

play with logs $$= dn \log n - d \left( \frac{n}{3} \right) \log 3 - \left( \frac{2n}{3} \right) \log \frac{2}{3} + \frac{n}{6}$$

play more $$= dn \log n - d \left( \log 3 - \frac{2}{3} \right) + \frac{n}{6}$$

setting $d \geq \frac{1}{6 \left( \log 3 - \frac{2}{3} \right)} = O(n \log n)$

2. (Description on the Problem Set)

(a) ... mumble ... mumble.

(b) Let the two jobs be $i$ and $j$. If $d_i c_j \leq d_j c_i$ we execute $i$ before $j$. Otherwise we execute $j$ before $i$. In fact, writing the two possible cost functions, it’s easy to check that

$$d_i (c_i + c_j) + d_j c_j \leq d_j (c_i + c_j) + d_i c_i \iff d_i c_j \leq d_j c_i$$

(c) We extend point (b) by noticing that two adjacent jobs share this property: let $1, \ldots, i, j, \ldots, n$ our sequence. Let $C = \sum_{k=j+1}^{n} c_k$ be the sum of the costs for the jobs not yet executed after $j$ completes. Note that the total costs $A = \sum_{k=1}^{i-1} c_k t_k$ and $B = \sum_{k=j+1}^{n} c_k t_k$ are equal both in a sequence with $(i, j)$ or with swapped $(j, i)$. We can now write the total cost for the two sequences: (first the $(i, j)$ sequence, then $(j, i)$)

$$A + d_i (c_i + c_j + C) + d_j (c_i + C) + B \leq A + d_j (c_i + c_j + C) + d_i (c_i + C) + B$$

Again, we improve (reduce) our total cost by swapping job $i$ with $j$ iff $d_i c_j > d_j c_i$.

(d) Using point (c), we can run a standard bubble-sort approach with runtime $O(n^2)$ where we swap two jobs $(i, j)$ to $(j, i)$ iff $d_i c_j > d_j c_i$ which is a function $f(c_i, d_i, c_j, d_j)$.

(e) From point (d) we know that sorting the jobs using the above function $f(c_i, d_i, c_j, d_j)$ is enough to minimize the cost function. Thus we choose a faster sorting algorithm such as Mergesort which operates in $O(n \log n)$.

3. (Description on the Problem Set)

(a) We show a reduction from the vertex cover (VC) problem. To transform an input instance for VC to an input instance for the scheduling problem we follow the hint: for each node $i \in [n]$ we create a machine $M_i$, and a job $J_i = (1, n, M_i)$ which can me assigned only to machine $M_i$ and takes all the resources of that machine. For each edge $e_l = (i, j) \text{ (with } l \in [m])$ where $i, j \in [n]$, we create
a job $J_{e_l} = (l, l, \{M_i, M_j\})$ which can be scheduled only on the machines $M_i$ or $M_j$ associated to the edge endpoints, and requires only the unit time $l$ to be completed.

(⇒) If a graph $G = (V, E)$ has a vertex cover $V'$ of size $|V'| = s$, then we can schedule at least $k = m + n - s$ jobs. To see this, we assign all the $m$ edge jobs to the machines associated to the nodes in $V'$. This can always be done because each edge $e_l = (i, j)$ has an endpoint (say $i$) in $V'$, thus we can run the job $J_{e_l} = (l, l, \{M_i, M_j\})$ on $M_i$. Moreover, the remaining $n - s$ machines which are empty receive the job associated to their vertices. The jobs scheduled are $m + n - s$.

(⇐) If we can schedule at least $k = m + n - s$ jobs then $G$ has a vertex cover of size at most $s$. Let $S$ (with $|S| \leq s$) be the set of unscheduled jobs. Notice that for each edge job $J_{e_l} = (l, l, \{M_i, M_j\})$ in $S$, machines $M_i$ and $M_j$ must run vertex jobs $J_i = (1, n, M_i)$ and $J_j = (1, n, M_j)$. We can remove one of these vertex jobs (say $J_i$) and schedule $J_{e_l}$ on $M_i$. Note that the size of $S$ does not change after this replacement. We perform this replacing step for each edge job in $S$ and, at the end, $S$ will contain only vertex jobs unscheduled. The corresponding vertices in $G$ are a vertex cover of size at most $s$. In fact, all the edge jobs must be associated to the machines where a node job is not associated. This is equivalent to a set of vertices covering all the edges in $G$.

(b) The jobs associated to the edges have already two machines to choose from. We only need to tweak the jobs associated to the nodes. A simple way to do that is to create $n$ additional machines $M_1, \ldots, M_n$. Moreover, for each node $i \in [n]$ we create 2 equal jobs: $J_i = (1, n, M_i)$ and $\hat{J}_i = (1, n, M_i, M_i)$. Note that they can both always be scheduled, unless $M_i$ is occupied by some edge: in this case only one can be scheduled on $M_i$. We only need to redefine $k = m + 2n - s/2$ and the correctness proof follows from part (a).

(c) Both problem are NP-Hard. It’s straightforward to show that the decisional version of each one of them has a polynomial verifier. The verifier $V$ will take the set of at least $k$ jobs that can be scheduled. Thus it checks if there are collision between any two jobs. In there are no collisions $V$ accepts, otherwise it rejects. Everything can be checked in polytime. Thus the problems are NP-complete.

4. (Description on the Problem Set)

We precompute some simple data about the tree: for each node $v \in T$ we obtain its parent $p(v)$, its height in the tree $h(v)$ (with height of the root $h(r) = 0$). This precomputation can be executed in $O(m)$ by traversing $T$. We sort all the jobs according to their bottom-up height. Note that since the height of $T$ can be at most $m$, an insertion sort will require linear time $O(m + n)$. The key to get linear time is to explore the tree starting from the leaves and to partially build solutions while we reach the root $r$. We use the following definition:

- For a node $v \in T$, let $c_1(v)$ and $c_2(v)$ be the children of $v$ (note that there could be 0, 1 or 2 children).
- For a node $c \in T$, let $T_c$ be the subtree of $T$ rooted at $c$.
- For a subtree $T_c$, let $J(T_c)$ be the set of jobs correctly scheduled in $T_c$, and let $P(T_c)$ be a shortest length job that has its starting node in $T_c$, it’s not colliding with any job in $J(T_c)$ and has a lower height then $h(c)$ (note that $P(T_c)$ could not exist).
Algorithm 1 Tree Schedule (TS) Algorithm

1: procedure TS\( (G = (V, E)) \) \( \triangleright \) Jobs to schedule
2: Starting from the leaves of \( T \) up to the root
3: for all \( v \in T \) do
4: we report only the complex case where \( v \) has 2 children (the other cases follow)
5: if \( P(T_{c_1(v)}) \) and \( P(T_{c_2(v)}) \) exist then
6: Let \( P(T_{c_1(v)}) \) be the job that ends before \( P(T_{c_2(v)}) \) w.l.o.g.
7: \( J(T_v) = J(T_{c_1(v)}) \cup J(T_{c_2(v)}) \)
8: if \( P(T_{c_2(v)}) \) ends in \( v \) then
9: \( J(T_v) = J(T_v) \cup \{P(T_{c_1(v)})\} \)
10: else
11: \( P(T_v) = P(T_{c_1(v)}) \)
12: end if
13: end if
14: Case where only one job between \( P(T_{c_1(v)}) \) and \( P(T_{c_2(v)}) \) exists, can be handled similarly
15: if \( P(T_{c_1(v)}) \) and \( P(T_{c_2(v)}) \) don’t exist then
16: \( J(T_v) = J(T_{c_1(v)}) \cup J(T_{c_2(v)}) \)
17: if there are jobs starting at \( v \) then
18: Let \( J_v \) be the shortest job starting at \( v \)
19: \( P(T_v) = J_v \)
20: end if
21: end if
22: end for
23: end procedure \( \triangleright \) Solution is \( J(r) \)

Complexity: We explore the entire tree starting from the leaves up to the root. At each step we merge sets of jobs and we look at shortest jobs in constant time, since they are sorted by length and it’s easy to associate them to their starting nodes in the precomputation phase. Exploring the tree takes \( O(m) \) time; moreover creating the final set of scheduled jobs \( J(r) \) requires in total \( O(n) \) steps, since the jobs are \( n \). Thus the total complexity is \( O(m + n) \).

Correctness: By taking always the shortest jobs we informally ensure that more space is left for future jobs. Let \( S \) be our solution. Suppose for a contradiction, there is a moment in our algorithm where not choosing the shortest job leads to a better solution \( S' \). Let \( J' \) be the first job, during the execution of the algorithm, which is considered in place of \( J \in S \). Note that by removing \( J \) from \( S \) does not allow to insert additional jobs except for \( J' \) because the solution \( S'' \) obtained before adding \( J \) is already optimal (induction argument). Thus we have \( |S| = |S'| \). Let \( v \) be the node where \( J \) and \( J' \) intersect. Every job ending above \( v \), which does not intersect \( J' \), does not intersect \( J \) also otherwise (because of the tree structure) it should touch \( J' \) at \( v \). Moreover, by ending first, \( J \) leaves more space for other jobs to be scheduled, thus \( |S| \geq |S'| \). A contradiction.