**Practice Final Exam** CS601, Algorithms and Complexity, Fall 2024

**Instructions:**

1. **DO NOT OPEN THE EXAM UNTIL YOU ARE INSTRUCTED TO.**
2. There are **5** problems printed across **20** pages in total. Please make sure all of you have exams with all the problems and all the pages[1].
3. Answer all of the questions as well as you can. You have three hours to complete this exam.
4. **The exam is non-collaborative; you must complete it on your own.** If you have any clarification questions, please ask the course staff. We cannot provide any hints or help. No cell phone or electronic devices or any use of Internet is permitted in this exam. Indeed, **if you are caught engaging in unseemly practices, severe action will be taken.**
5. This exam is closed-book, except for
   (a) Up to *two double-sided sheets of paper* that you have prepared ahead of time. You can have anything you want written on these sheets of paper.

**Good Luck** **Total points: 100**
**Time Limit: 3 hours**

---

[1]The actual final might have 5 or 6 problems

1. **(20 pt.) (Dynamic Programming)** A thief is planning a robbing spree on the houses located on an entire street. He studied the daily schedule of people living there carefully. In particular, he can rob $n$ houses, and for each he knows when the owner leaves the house (at time $\ell_i$) and how long the owner is gone (for a duration $d_i$). He knows that because of the security systems, he has to get into the house (through a back door) exactly when the owner leaves the house and get out exactly when the owner is back. He also knows the value of goods that he could steal from each house. (Assume it is possible for the thief to get from a house to another in no time. He is an athlete after all).

   (a) **(5 pt.)** Briefly and precisely define the subproblems.

   (b) **(10 pt.)** Write the recurrence relation between the subproblems. Also write out the base case.

   (c) **(5 pt.)** Find the runtime of your algorithm.

**SOLUTION:**

(a) Let $f_i = \ell_i + d_i$ denote the time when the owner returns to the house and let $v_i$ denote the value of goods in house $i$. Sort all of the houses by $f_i$ so that $f_1 \leq f_2 \leq f_3 \leq \ldots \leq f_n$ and let

$$OPT_i = \text{ the optimal solution involving houses } 1, \ldots, i$$

(b) For the base case, observe that $OPT_0 = v_0$ and $OPT_1 = v_1$. Let us handle the DP recurrence next. Note that it is either the case that

i. the robber includes house $i$ on his schedule in which case he cannot rob any houses $j < i$ with $f_j > \ell_i$ so we have $OPT_i = v_i + max_{j:f_j \leq \ell_i} OPT_j$, or

ii. the robber does not rob house $i$ so that $OPT_i = OPT_{i-1}$.

Thus, we have the following recursive formula for dynamic programming

$$OPT_i = max\{OPT_{i-1}, v_i + max_{j:f_j \leq \ell_i} OPT_j\}.$$

(c) We can sort our array and fill in our dynamic programming table in time $O(n^2)$ in the straightforward way. This is because finding $OPT_i$ requires scanning at most $n$ cells. You can actually get a faster implementation using binary search[2] as well which is presented below.

**Initialize** $OPT_0 = 0$.

**For** $j = 1$ *to* $n$
**Run Binary Search to Find** $j = max\{k : f_k \ell_i\}$ (set $j = 0$ if no $k$ exists)
**Set** $OPT_i = max\{OPT_{i-1}, v_i + OPT_j\}$ Once we have computed $OPT_i$ for each $i \leq n$ we can work our way backwards to figure out which houses the robber should hit.

**Sort** houses so that $f_1 \leq f_2 \ldots$ **Initialize** $S = \{\}, j = n$
**While** $j \geq 1$

**If** $OPT_j > OPT_{j-1}$, **then**
**Set** $S = S \cup \{j\}$
**Set** $j = max\{k : f_k \leq \ell_j\}$ or **Set** $j = 0 if f_k > \ell_j$ for all jobs $k$.
**Else** (i.e., we have $OPT_j > OPT_{j-1}$)
**Set** $j = j - 1$

[Continued...]

[Continued...]

2. **(20 pt.) (Tonian Cycles and Hamiltonian Cycles)** A Hamiltonian cycle in a graph $G$ is a cycle that goes through every vertex of $G$ exactly once. Deciding whether an arbitrary graph contains a Hamiltonian cycle is NP-hard. A Tonian Cycle in a graph $G$ is a cycle that goes through at least half of the vertices of $G$. Show the following.

   (a) Show that Tonian Cycle $\in$ NP.

   (b) Assuming that HAM-CYCLE is complete for the class NP, prove that Tonian Cycle is NP-Complete.

**SOLUTION:**

(a) Given a graph $G$ and a subset of vertices claimed to be a Tonian Cycle, one can easily verify whether the subset induces a cycle and whether this cycle contains at least half the vertices in polynomial time. This means that Tonian Cycle belongs to the class NP.

(b) To prove that Tonian Cycle is NP-Hard, we show the reduction HAM-CYCLE $\leq_p$ TSP. Given a graph $G = (V, E)$, the reduction just produces a graph $H$ which just consists of two disjoint copies $G_1, G_2$ pf $G$. Clearly, this takes polynomial time. Now, let's prove that this reduction takes a YES (resp. NO) instance of HAM-CYCLE to a YES (resp. NO) instance of Tonian Cycle.

Let's suppose $G$ contains a Hamiltonian Cycle. Then in the graph $H$, both $G_1$ and $G_2$ contain a replica of this cycle. Let us focus on the cycle in $G_1$. It contains at least half of the vertices of $H$ confirming that $H$ has a Tonian Cycle.

Next, suppose the graph $H$ obtained after the reduction has a Tonian Cycle. So, this cycle contains at least half of the vertices. Since $G_1$ and $G_2$ are disjoint and contain the same number of vertices, this means the Tonian Cycle in $H$ either lives entirely in $G_1$ (and thus there is another Tonian Cycle living entirely in $G_2$ as well as these graphs are replicas of each other). In particular, these Tonian Cycles have lengths equal to number of vertices in $G$ and thus $G$ contains a Ham-Cycle as desired.

[Continued...]

3. **(20 pt.) (Max Flow)**

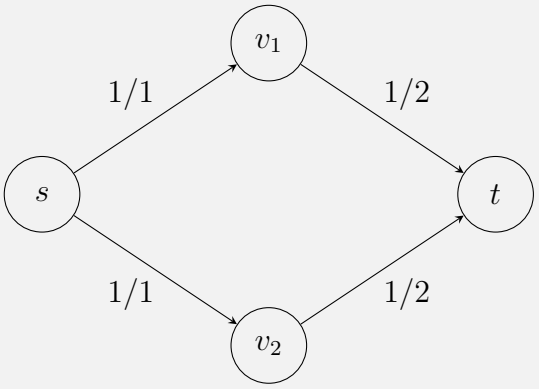
Let $G = (V, E)$ be a flow network, with a source node $s$, target node $t$, and positive integer edge capacities $c(e)$ for each edge $e \in E$. For each of the following claims, state whether it is true or false. If it is true, explain why. If it is false, draw (and briefly explain) a flow network that shows a counterexample.

**Claim 1** If $f$ is a max flow, then $f$ saturates every edge into $t$ (i.e., $f(e) = c(e)$ for all edges $e$ into $t$).

**Claim 2** Let $(A, B)$ be a min cut. If we multiply the capacity of every edge in $G$ by 2, then $(A, B)$ is still a min cut

**SOLUTION:**

**Counterexample to Claim 1:** This claim is false. This can be seen through the following counterexample.




The value of max-flow in this example is 2. Indeed, the given flow has value 2 and the cut $\Big(\{s\}, \{v_1, v_2, t\}\Big)$ has capacity 2 as well. The presented flow is a maximum flow and it does not saturate any of the two edges entering $t$.

**Proof of Claim 2** Yes, this cut will remain a min-cut. The value of every single cut gets scaled up by a factor two and the same holds for the min-cut as well.

[Continued...]

[Continued...]

4. **(20 pt.) (Graph Cuts)**

You are given a connected undirected graph $G = (V, E)$ with positive edge weights $w_e$, and you are asked to find a cut in the graph where the largest edge in the cut is as small as possible (note that there is no notion of source or target; any cut with at least one node on each side is valid). Solve the following problems. Give an algorithm for this problem which runs in time $O(|E| \log |V|)$ and prove that your algorithm is correct.

**SOLUTION:** Sort the array of edge weights and binary search for the largest edge in the cut. For an edge with weight $w_m$, consider $G'$, the graph obtained by keeping only the edges from $E$ with weight $\leq w_m$. If $G'$ is connected, then the answer is $\geq w_e$. Otherwise, the answer is $< w_m$.

Once you find the max weight edge $e$ with weight $w_e$ in the cut, you can recover the cut by taking any connected component and its complement after removing all edges of weight $\geq w_e$. Overall runtime is $O(|E|log|V|)$.

[Continued...]

[Continued...]

5. **(20 pt.) (Divide and Conquer)**

**Definition:** An array $A[0 \ldots n-1]$ is said to have a 1/3-plurality element if some value $v$ appears $> n/3$ *(i.e., strictly greater than $n/3$)* times in the array; each such value $v$ is called a 1/3-plurality element.

Design a divide and conquer algorithm that given $A[0 \ldots n-1]$ outputs "Yes" and a 1/3-plurality element if $A[0 \ldots n-1]$ has a 1/3-plurality element or "No" if $A[0 \ldots n-1]$ does not have a 1/3-plurality element. (If $A[0 \ldots n-1]$ has multiple 1/3-plurality elements, the algorithm can return any of them.) Your algorithm should have $O(n \log n)$ running time and you must prove the correctness of your algorithm.

However, there is a special restriction. The only thing you are allowed to do with array elements is compare whether they are identical (test whether $A[i] = A[j]$ for some $i, j$ of your choice). The array elements are not from an ordered domain, so you cannot compare them using $<$ and $>$, and you cannot hash the array elements.

**SOLUTION:**

Let us design a divide-and-conquer method that finds all of the 1/3-plurality elements. There can be at most 2 such elements since each one has to occupy more than one third of the array. Then the idea is that if we break the array into two halves, each 1/3-plurality element must be a 1/3-plurality in at least one of the halves. So we can recursively find all 1/3-plurality elements in each of the halves, and in the end we have at most 4 candidates. We will denote the list of candidates maintained by the algorithm as $C$. We can check whether they are truly a 1/3-plurality by just counting how many times they occur in the full array, in $O(n)$ time.

The running time is $O(n \log n)$. The total number of items returned by each call can be at most 2, since there can be at most two 1/3-plurality elements in any array. Therefore C can have size at most 4, which means that line 9 gets executed at most 4 times, each time taking O(n) time. Now if T (n) is the running time of the algorithm for an array of size n, then we just proved that

$$T(n) = 2T(n/2) + O(n).$$

By appealing to the master theorem, we can see that this results in a running time of $O(n \log n)$.

Finally, we argue that this algorithm is correct. We do this using induction. Let $P(n)$ denote the assertion that, for all arrays $A$ of length $n$, the above algorithm correctly returns all 1/3-plurality elements of $A$. The base case is for $n = 1$ which trivially checks out. Assume the algorithm correctly finds plurality elements in all arrays with stricly fewer than $n$ elements. It suffices ot show that some true plurality element gets inserted in the list $C$ of candidates.

Consider one such element $x$. If $x$ does not get added to $C$ it means that it was not a 1/3-plurality in $A[0 \ldots n/2-1]$, so it appeared at most 1/3 fraction of times in this part (by the induction hypothesis). Similarly if it does not get added in the recursive call on the right

half, it means that it appeared at most $1/3$ fraction of times in $A[n/2 \dots n-1]$. So overall, the number of times it can appear in $A$ is at most $n/3$. But this is in contradiction with the assumption that $x$ was a $1/3$-plurality element. So every $1/3$-plurality element will be added to $C$ as we wanted to show.

[Continued...]

[Continued...]

[Continued...]

[Continued...]

[Continued...]

[Continued...]

[Continued...]