Special Note: I have given very detailed answers. But you are required to be brief when you answer(!)

1. Consider the page cache problem. Suppose you are told that each page will be requested at most two times, Will the worst case performance of the LRU algorithm be better than the $k.OPT + k$ upper bound proved in the class? Derive the best possible bound upper bound for the cache performance for this case. Give a page request sequence that matches your bound. [3]

   Soln: An obvious bound that is better than $k.OPT + k$ can be derived as follows. Any request sequence of length n must involve $\lceil \frac{n}{2} \rceil$ distinct page requests as a page can be requested only two times. This means **any** algorithm will incur at least $\lceil \frac{n}{2} \rceil$ misses. Thus $OPT \geq \lceil \frac{n}{2} \rceil$. On the other hand, the "worst" possible algorithm will involve n misses at most. Thus, any algorithm (including LRU) requires at most $n \leq 2.\lceil \frac{n}{2} \rceil \leq 2.OPT$ misses. To get an concrete example, Consider cache size of k and the request sequence $1, 2, ...k, (k+1), 1, 2, ...(k-1)$. The request sequence has length $n = 2k$. LRU incurs $n = 2k$ misses whereas optimal policy would incur $k + 1$ misses. Thus, the actual number of misses is $2(k+1) - 2 = 2.OPT - 2$. If you come up with a tighter example, better (I couldn't).

2. Recall that the interval scheduling problem asks you to select a maximum collection of non-conflicting intervals from a given set of intervals. Consider the following modification of the optimal algorithm discussed in the class. We pick the interval that **starts last**. Remove the picked interval and all intervals which are in conflict with this one from the set of intervals and recursively select the best collection of non-conflicting intervals from the remaining set. Will this modified strategy yield an optimal schedule? Give a formal proof or a counter example. [3]

   Soln: Suppose an optimal solution S is given where the interval that starts last is not picked. Let (s_k, f_k) be the latest starting interval **in the set** S. Let (s_n, f_n) be the actual last item to start in the given set of intervals, which has been left out from S. Then, if we add the interval (s_n, f_n) to S, there will be a conflict. How many elements can be in conflict? No item in S other than (s_k, f_k) can start or finish after s_k, for then it is not possible for (s_k, f_k) to be the latest starting interval in S. Since $s_n > s_k$, none of these items can be in conflict with (s_n, f_n) as well. Thus, the only interval in S that can be in conflict with (s_n, f_n) is (s_k, f_k). We can now replace (s_k, f_k) with (s_n, f_n) to get a solution of the same size as S containing (s_n, f_n). Thus, we have shown that it is always possible to find an optimal solution containing (s_n, f_n), which is the latest starting interval.

   In any optimal (or otherwise) solution containing (s_n, f_n), we can't include intervals that conflict with $(s_n.f_n)$ and hence it is safe to forget (remove) intervals in conflict with (s_n, f_n) once (s_n, f_n) is picked. Any solution for the original problem containing (s_n, f_n) can't contain any of these removed intervals. Suppose S' is an optimal collection of non-conflicting intervals from this remaining set of intervals (call it the *residual set*) — which we may compute recursively, we claim that $S' \cup \{(s_n, f_n)\}$ is optimal for the original problem.

   The only possibility for our strategy to fail is that there is some set T' of intervals from the residual set such that $T' \cup \{s_n, f_n)\}$ is the optimal solution for the original problem. In such case, T' must be a valid schedule for the residual set. But, by definition, S' is an optimal schedule for the residual set. This means $|S'| \geq |T'|$. This means that the $S' \cup \{s_n, f_n)\}$ must be at least as big as $T' \cup \{s_n, f_n)\}$. Consequently, our solution, which is $S' \cup \{s_n, f_n)\}$ must also be optimal.

3. Suppose you are given n items with weights $w_1, w_2, ..., w_n$ Suppose your maximum weight carrying capacity is W. Consider the following algorithm for calculating the maximum number of items you can carry from the set (items cannot be taken fractionally): [3+3+3]
   $OPT(n, W)$

   - if $(n == 1)$ { if $(w_1 \leq W)$ return 1; else return 0; }
   - if $(w_n > W)$ return $OPT(n-1, W)$;
   - return $\max\{1 + OPT(n-1, W-w_n), OPT(n-1, W)\}$;

1. Is the above algorithm correct for the problem? Justify your answer. What is the complexity?

   Soln: Consider an optimal solution S for the problem. The item n is either present in it or not present in it. If item n is present in S, then the remaining items in S must be the optimal choice from 1 to $n-1$ with the budget of $W - w_n$ remaining after picking n. If item n is not present in S, then then S must be the optimal way to pick items from 1 to $n-1$ with the whole budget W available. (a formal argument establishing these will be similar to what was done for the previous question). Since the algorithm works exactly doing this, correctness follows.

   To analyze the complexity, Let $T(n)$ be the complexity of the algorithm when n items are present. an inspection of the algorithm leads to the observation that computation with n items involve two recursive computations involving $n-1$ items plus an $O(1)$ cost. This leaves us with the recurrence $T(n) = O(1) + 2T(n-1)$, yielding $T(n) = 2^n O(1)$.

2. Use dynamic programming to improve the running time of the above algorithm to $O(nW)$. What is the time complexity?

   Soln: We keep an array $COST[1..n][0...W]$ of size nW, initializing all costs to ∞. $OPT(n, W)$, once computed will be entered into the array so that any further recursive call to $OPT(n, W)$ will do a table look up an return the store value in the array in $O(1)$ avoiding further computation. The procedure (except the initialization step) is as follows: $OPT(n, W)$

   - if $(COST[n, W] < \infty)$ return $COST[n, W]$; // Value has been computed earlier
   - if $(n == 1)$ { if $(w_1 \leq W)$ return 1; else return 0; }
   - if $(w_n > W)$ return $COST[n, W] = OPT(n-1, W)$;
   - return $COST[n, W] = \max\{1 + OPT(n-1, W - w_n), OPT(n-1, W)\}$;

   Since each value of $COST(n, W)$ involves only one time computation, the cost of computation is determined by the number of distinct entries in the array cost, which is nW. Since each call to OPT involves $O(1)$ computation outside the (sub) recursive call costs which has been accounted above, the total cost is $O(nW)$.

3. Design an $O(n \log n)$ algorithm for the problem. Prove the algorithm correct.

   Soln: The following obvious Greedy strategy works. Pick the least weight item, say w_1. Now recursively find the optimal pick from the remaining elements from the budget $W - w_1$. A proof for the correctness of this strategy involves establishing two things - first, we must show that there is an optimal solution containing the minimum weight item w_1. This justifies the first step of the algorithm of picking the minimum weight item. Second, we must prove that after this, the problem indeed reduces to recursively solving the same problem on the remaining items with the remaining budget. For this, we need to show that if S is an optimal solution containing the minimum weight item w_1, then $S \setminus \{w_1\}$, is indeed an optimal selection from the remaining items for the residual budget $W - w_1$. The detailed formal argument involves arguments similar to the proof for the Greedy strategy used in the second question of this question paper.

   The simplest implementation is to sort the items in the increasing order of weight first and then consider them in order for inclusion into the solution. The sorting step takes $O(n \log n)$. The selecting step takes $O(n)$ ($O(1)$ to process each item in the sorted set). Thus, the total complexity is $O(n \log n + n)$ which is $O(n \log n)$.