Fantasy Turkey Farm Sweepstake:

Congratulation! You have just won our annual Fantasy Turkey Farm Sweepstake. Your prize includes an all expenses paid 1-week vacation at our Fantasy Turkey Land. At the conclusion of your trip; you are also entitled to bring home with you all the turkeys you can carry from our world-renounced organic turkey farm. If you do desire, you may exchange the turkeys you carry out for their equivalent cash values. In order to protect our guests from injuring themselves when carrying their beloved turkeys, you will be required to take a strength test in advance so as to determine the maximum weight (in lb) you can carry, and the total weights of the turkeys you are allowed to carry must not exceed this predetermined amount. As you may have already known, an overweighed turkey does not always worth more than a well-conditioned turkey since our flagship Tinkerbelle turkey is worth many times more than our meaty CouchPotato turkey! To assist you with your selection, all ready-to-pick turkeys in our farm will carry a tag with its weight and equivalent cash value clearly marked.

Q: How do you design an algorithm to help with your selections so as to maximize your cash reward?

Formalization:

Input: Given a set of n objects (turkeys) $S = \{x_1, x_2, \ldots, x_n\}$ and a total weight $W > 0$; each $x \in S$ has a weight $w(x) > 0$ and a cash reward $c(x) > 0$.

Output: A subset $H \subseteq S$ such that $\sum_{x \in H} w(x) < W$ and $\sum_{x \in H} c(x)$ is maximized.

Objective function: Maximizing $\sum_{x \in H} c(x)$.

Constraints: $\sum_{x \in H} w(x) < W$.

Remark: This is a classical optimization problem.

Optimization Problem:

Given an objective function and a set of constraints, compute a solution that will satisfy the given set of constraints and also optimize the objective function.

Feasible solution: Any solution that satisfies the given set of constraints.
Optimal solution: A feasible solution optimizing the objective function.

Happy Turkey Day:

Algorithm 1: Selection by max cash rewards.
Step 1: Sort the objects in decreasing cash rewards.
Step 2: During each iteration, select the object with maximum cash reward and its total weights < W.

Algorithm:

```
H ← ∅;
totalWeight ← 0;
totalCash ← 0;
Q ← Sort the objects in decreasing cash rewards;
while S ≠ ∅ and totalWeight < W do
    x ← deque(Q);
    if W ≥ totalWeight + w(x)
        then H ← H ∪ {x};
             totalWeight ← totalWeight + w(x);
             totalCash ← totalCash + c(x);
    endif;
endwhile;
```

Complexity:
T(n) = O(nlgn). **(HW)**

Q: How good/bad is this algorithm in generating an optimal solution?

Algorithm 2: Selection by min weight.
Step 1: Sort the objects in increasing weights.
Step 2: During each iteration, select the object with minimum weight.

Algorithm:

```
H ← ∅;
totalWeight ← 0;
totalCash ← 0;
Q ← Sort the objects in increasing weights;
flag ← true;
while S ≠ ∅ and flag do
    x ← deque(Q);
    if W ≥ totalWeight + w(x)
        then H ← H ∪ {x};
             totalWeight ← totalWeight + w(x);
             totalCash ← totalCash + c(x);
        else flag ← false;
    endif;
endwhile;
```

Complexity:
T(n) = O(nlgn). **(HW)**

Q: How good/bad is this algorithm in generating an optimal solution?

Algorithm 3: Selection by max reward per unit weight ratio, $\frac{c(x)}{w(x)}$.

Step 1: For each object x ∈ S, compute r(x) = $\frac{c(x)}{w(x)}$.

Step 2: Sort the objects in decreasing reward per unit weight ratio, $\frac{c(x)}{w(x)}$.

Step 2: During each iteration, select the object with max reward per unit weight ratio and its total weights < W.

Algorithm:

```
H ← ∅;
totalWeight ← 0;
totalCash ← 0;
for each x ∈ S do
    r(x) = c(x)/w(x);
endfor;
Q ← Sort the objects in decreasing reward per unit weight ratio;
while S ≠ ∅ and totalWeight < W do
    x ← deque(Q);
    if W ≥ totalWeight + w(x)
        then H ← H ∪ {x};
             totalWeight ← totalWeight + w(x);
             totalCash ← totalCash + c(x);
    endif;
endwhile;
```

Complexity:

T(n) = O(nlgn). **(HW)**

Q: How good/bad is this algorithm in generating an optimal solution?

Q: What are the characteristics of the above algorithms?

Observation:

Solution is constructed iteratively. During each iteration, local optimization criteria are used to select a component among a set of possible candidates so as to form a feasible solution. This is a "greedy" approach in problems solving.

Basic Characteristics of Greedy Algorithms:

1. Iterative in nature. It computes a feasible solution that may or may not be optimal in a piece-meal fashion. Solution is constructed component by component until a feasible solution is obtained if possible.
2. There exists a candidate set, C, in which component of a feasible solution can be selected.
3. There exists a selection function, select(C), with which a new component will be selected from C according to some local optimization criteria.
4. There usually exists a feasibility test, feasible($S \cup \{x\}$), that determines whether the current partial solution S together with the newly selected component x may still lead to a feasible solution. If true, S is updated to $S \cup \{x\}$; otherwise, x is rejected. Either case, x will be removed from the candidate set. In some greedy algorithms, no separate feasibility test is required since it will be incorporated into the selection rule.
5. There exists a solution test, soln(S), to determine whether S is a feasible solution. If true, the algorithm is terminated and S is returned. Otherwise, the process continues so long as $C \neq \emptyset$.

Generic Greedy Algorithm:

```
Greedy(C: Set): S;
    S ← ∅;
    while C ≠ ∅ and !soln(S) do
        x ← select(C);
        C ← C – {x};
        if feasible(S ∪ {x})
            then S ← S ∪ {x}
        endif
    endwhile;
    if soln(S)
        then return S;
        else return ("no solution")
    endif;
endGreedy;
```

Examples:

1. *0-1 Knapsack Problem.*

Given a set of n objects S = {x_1, x_2, …, x_n} and a maximum capacity C > 0; each $x \in S$ has a capacity $c(x) > 0$ and a value $v(x) > 0$. Find a subset $H \subseteq S$ such that $\sum_{x \in H} c(x) < C$ and $\sum_{x \in H} v(x)$ is maximized.

Observe that the above turkey problem is a 0-1 Knapsack Problem.

Greedy Approach 1:

The candidate Set: The set of currently available objects.
Selection Function: Select an object with the highest value from the candidate set.
Feasibility Test: The total capacity of the chosen set of objects must not exceed C.
Solution Test: No more objects can be added to the knapsack without exceeding the capacity C.

Greedy Approach 2:

The candidate Set: The set of currently available objects.
Selection Function: Select an object with the smallest capacity from the candidate set.
Feasibility Test: The total capacity of the chosen set of objects must not exceed C.
Solution Test: No more objects can be added to the knapsack without exceeding the capacity C.

Greedy Approach 3:

The candidate Set: The set of currently available objects.
Selection Function: Select an object with the maximum value/capacity ratio from the candidate set.
Feasibility Test: The total capacity of the chosen set of objects must not exceed C.
Solution Test: No more objects can be added to the knapsack without exceeding the capacity C.

HW. Formalize these greedy algorithms. Prove or disprove that these greedy algorithms will always generate an optimal solution.

2. Minimum Changing Coins Problem.
Given a finite set of coins, how do you give change to a customer using the smallest number of coins?
Greedy approach:
The candidate Set: The set of currently available coins.
Selection Function: Select the highest-valued coin in the candidate set.
Feasibility Test: The total value of the chosen set of coins must not exceed the amount to be given to your customer.
Solution Test: The total value of the chosen set of coins equals to the amount to be given to your customer.

HW. Formalize this greedy algorithm. Prove or disprove that this greedy algorithm will always generate an optimal solution.

3. *Matrix Chain Product Problem.*
Given a sequence n matrices $M_1, M_2, \ldots, M_n$ with dimensions $<d_0, d_1, d_2, d_3, \ldots, d_{n-1}, d_n>$, where matrix M_i is of dimension $d_{i-1} \times d_i$.

Q: How do we design and analyze a greedy algorithm to multiply this sequence of matrices together so that the total number of multiplications required is minimized? Prove or disprove that your greedy algorithm will always generate an optimal ordering for multiplications.

4. *Minimum Cost Communication Network Problem:*
Given a set of communication stations $S = \{s_1, s_2, \ldots, s_n\}$ to be connected together so that between any two stations s_i and s_j, s_i can always communicate with s_j either directly through a link or indirectly through a communication path consisting of other stations serving as transmitting stations. If the cost for building a direct link between any two stations s_i and s_j is known in advance, design an algorithm to build a cheapest communication network.

This problem is the same as the Minimum Spanning Tree Problem.

Minimum Spanning Tree Problem.
Given a connected undirected graph G = (V,E), |V| = n, |E| = m, and a cost function c: $E \rightarrow R^+$.
Construct a minimum cost spanning tree $T = (V, E_T)$ such that $\sum_{e \in E_T} c(e)$ is minimized.

Greedy Approach:
Feasible solution: A spanning tree.
Candidate Set: Set of edges E.
Selection Rule: Pick the minimum cost edge among all available edges.
Feasibility Test: No cycle formed for a set of selected edges.
Solution Test: A graph with no cycle and exactly n – 1 edges.

Kurskal's Algorithm:

```
Kruskal(E: Set): ET;
    ET ← ∅;
    while E ≠ ∅ and ET ≠ n – 1 do
        x ← select min cost edge in E;
        E ← E – {x};
        if ET ∪ {x}does not form a cycle
            then ET ← ET ∪ {x}
        endif
    endwhile;
    if ET = n – 1
        then return ET;
        else return ("no solution")
    endif;
endKruskal;
```

Two major difficulties:

1. **Selecting min cost edge**:
   Need priority queue; better yet, sort the set of edges in E.
2. **Detecting cycle**:
   Use union-find disjoint set operations.

HW. Review ADT graph, priority queues and disjoint set.

Implementation:

```
Kruskal(E: Set): E_T;
    for i = 1 to n do
        makeset(i)
    endfor;
    Q ← sort(E);
    E_T ← ∅;
    while Q ≠ ∅ and E_T ≠ n – 1 do
        x = (v,w) ← deque(Q);
        if (find(v) ≠ find(w))
            then  E_T ← E_T ∪ {x};
                  union(find(v),find(w))
        endif
    endwhile;
    if E_T = n – 1
        then return E_T;
        else return ("no solution")
    endif;
endKruskal;
```

Complexity Analysis:

Recall that:

Sorting: O(mlgm)

Disjoint set operations: $O(m\,\alpha(m,n))$

Hence,

$T_w(n) = O(mlgm) + O(m\,\alpha(m,n))$, where $\alpha(m,n)$ is the inverse Ackerman function.

Theorem: Kruskal's algorithm correctly computes a minimum spanning tree for any given connected graph G = (V,E), |V| = n.

Proof. Assume not to obtain a contradiction. Let $E_T = \{e_1, e_2, \ldots, e_{n-1}\}$ be the set of edges in the order selected by Kruskal's algorithm. Hence, $c(e_1) \le c(e_2) \le \ldots \le c(e_{n-1})$ and $T = (V, E_T)$ does not form a MST of G. Let $T^* = (V, E_{T^*})$ be a MST of G with $E_{T^*} = \{e_1^*, e_2^*, \ldots, e_{n-1}^*\}$, $e_1^* \le e_2^* \le \ldots \le e_{n-1}^*$. Observe that $E_T \ne E_{T^*}$, or else T is a MST of G. Let $e_i \in E_T$, $1 \le i \le n-1$, be the edge with the smallest i such that $e_j \in E_T \cap E_{T^*}$, $1 \le j \le i-1$, and $e_i \notin E_{T^*}$. Consider the graph formed by the set of edges $E_{T^*} + e_i$, which is the MST T* together with a unique cycle C. Observe that the edge $e_i \in E_T$ is part of the cycle. Consider the set of edges in this cycle C. There must exist at least one edge x in C, $x \ne e_i$, such that $x \notin E_T$. If not, each and every edge in C must all belong to E_T, resulting in a cycle in the spanning tree formed by E_T. Now, consider the graph formed by the set of edges $E_{T^{**}} = E_{T^*} + e_i - x$, which again forms a spanning tree $T^{**} = (V, E_{T^{**}})$ of G. Observe that when e_i is selected by Kruskal's algorithm in the construction of T, edge x is also available but is not selected. There are three cases that x will not be selected by Kruskal's algorithm.

Case 1: Edge x forms a cycle with $\{e_1, e_2, \ldots, e_{i-1}\}$: This is not possible since $\{e_1, e_2, \ldots, e_{i-1}\}$ and x are all in T*, implying that T* contains a cycle.

Case 2: $c(x) > c(e_i)$: If true, we have $c(E_{T^*} + e_i - x) < c(E_{T^*})$, which is again a contradiction since E_{T^*} is a MST.

Case 3: $c(x) = c(e_i)$: If true, we have $c(E_{T^*} + e_i - x) = c(E_{T^*})$. Recall that $\{e_1, e_2, \ldots, e_i\} \subseteq E_T \cap E_{T^{**}}$. Replacing T* with this new MST T**, we can repeat the above construction, and analysis, by finding an edge $e_k \in E_T$, $i+1 \le k \le n-1$, such that $e_j \in E_T \cap E_{T^{**}}$, $1 \le j \le k-1$, and $e_k \notin E_{T^{**}}$. Since G is a finite graph, after a finite number of steps, we will either obtain a contradiction (Case 1 and 2) or we will be able to transform T to a MST without decreasing the cost of T, implying that T is indeed a MST of G. Hence, a contradiction is reached.

Q: Can we avoid the process of detecting cycle?

If cycles can never be formed, no need to detect them.

Prim's algorithm:

Starting at an arbitrary vertex v to grow just one tree $T = (V_T, E_T)$. During each iteration, a min cost edge x = (v,w) is selected with $v \in V_T$ and $w \notin V_T$. Hence, no cycle can be formed.

Remark: Using this approach, there will be no feasibility test required.

Prim's Algorithm for MST:
Input: Connected graph G = (V,E), V = {1, 2, …, n} and cost function c: E $\rightarrow R^+$.
Output: MST T = (V_T, E_T).

Approach: Starting at vertex 1, grow a spanning tree for G.

Prim's Algorithm:

```
Prim(E: Set): E_T;
    E_T ← ∅;
    V_T ← {1};
    // create a priority queue Q containing all edges incident from V_T such that
    // (v,w) ∈ Q implies that v ∈ V_T and w ∉ V_T.
    Build(Q);                       // Q = {(v,w) | v ∈ V_T and w ∉ V_T}.
    while Q ≠ ∅ and V_T ≠ n do
        (v,w) ← deque(Q);           // Select min cost edge (v,w)
        E_T ← E_T ∪ {(v,w)};
        V_T ← V_T ∪ {w};
        Update(Q);                  // Update Q to include all edges incident from V_T
    endwhile;
    if E_T = n – 1
        then return E_T;
        else return ("no solution")
    endif;
endPrim;
```

Theorem: Prim's algorithm correctly computes a minimum spanning tree for any given connected graph G.

Complexity Analysis:
Using a priority queue such as min-heap, we have $T_w(n) = O(mlgm)$.

Remark: We can also avoid the use of a priority queue using the following labeling technique without using a priority queue.

Consider two group of vertices, $v \in V_T$ or $v \notin V_T$.
For each vertex $v \in V_T$, define

$\min(v)$ = min cost from v to a vertex in V_T,
other(v) = $u \in V_T$ such that $c(v,u) = \min(v)$.

Prim's Algorithm

```
E_T = ∅;                          // initialization
V_T = {1};
min(1) = ∞;
for v = 2 to n do
    min(v) = c(1,v);
    other(v) = 1;
endfor;
while |V_T| < n do                // growing tree from vertex 1
    min = ∞;                      // find vertex w ∉ V_T such that min(w) is minimized;
    for v = 2 to n do
        if min(v) < min
            then  min = min(v);
                  w = v;
        endif;
    endfor;
    V_T = V_T ∪ {w};              // update tree
    E_T = E_T ∪ {(w,other(w))};
    min(w) = ∞;
    for v = 2 to n do             // update vertex v ∉ V_T
        if min(v) > c(w,v)
            then  min(v) = c(w,v);
                  other(v) = w;
        endif;
    endfor;
endwhile;
return E_T;
endPrim;
```

Complexity Analysis:

$T_w(n) = O(n^2)$.

HW. Research different variations of spanning tree problems.

5. *Minimum Dominating Set Problem.*
Given a connected undirected graph G = (V,E). A *dominating set* $D \subseteq V$ is a set of vertices in V such that $\forall\ u \in V$, if $u \notin D$, then $\exists\ w \in D$ such that $(w,u) \in E$. A minimum dominating set of G is a dominating set with minimum number of vertices.
Q: Given G, how do we compute a minimum dominating set for G?

Greedy Approach:
Candidate Set: The set of available vertices.
Selection Rule: Pick a vertex with maximum degree.
Feasibility Test: None required.
Solution Test: No more vertices that are not adjacent to a vertex in D.

Greedy Minimum Dominating Set Algorithm:

```
D ← ∅;
while V ≠ ∅ do
    v ← select max degree vertex in V;          // Ties will be broken according to label
    D ← D ∪ {v};
    updating G, and hence V, by removing v and all vertices adjacent to v; // HW
endwhile;
return D;
```

Complexity Analysis:
During each iteration, at least one vertex will be deleted. Hence, the while-loop will be executed O(n) times. To select a max degree vertex and to update G take O(n) time, hence, $T(n) = O(n^2)$.

HW. Formalize the updating step and this greedy algorithm. Prove or disprove that this greedy algorithm will always generate an optimal solution.

6. Minimum Vertex Cover Problem.
Given a connected undirected graph G = (V,E). A *vertex cover* $C \subseteq V$ is a set of vertices in V such that $\forall\ e = (u,w) \in E$, if $u \notin C$, then $w \in C$. A minimum vertex cover of G is a vertex cover with minimum number of vertices.
Q: Given G, how do we compute a minimum vertex cover for G?

Greedy Approach:
Candidate Set: The set of available vertices.
Selection Rule: Pick a vertex with maximum degree.
Feasibility Test: None required.
Solution Test: No more edges that are not incident to a vertex in C.

Greedy Minimum Vertex Cover Algorithm:

VC $\leftarrow \varnothing$;
while E $\neq \varnothing$ do
    v $\leftarrow$ select max degree vertex in V; // Ties will be broken according to label
    VC $\leftarrow$ VC $\cup$ {v};
    updating G, and hence E, by removing v and all edges incident to v; // HW
endwhile;
return VC;

Complexity Analysis:
For each iteration, at least one vertex will be selected. Hence, the while-loop will be executed O(n) times. To select a max degree vertex and to update G take O(n) time, hence, $T(n) = O(n^2)$.

Example showing non-optimality of both algorithms:

2 ——4 ——1 ——5 ——3

Greedy dominating set = {1, 2, 3}.
Optimal dominating set = {4, 5}.

Greedy vertex cover = {1, 2, 3}.
Optimal vertex cover = {4, 5}.

7. *Maximum Independent Set Problem.*
Given an undirected graph G = (V,E). An *independent set* I $\subseteq$ V is a set of vertices in V such that $\forall$ u, w $\in$ I, (u,w) $\notin$ E. A maximum independent set of G is an independent set with maximum number of vertices.

Q: Given G, design and analyze a greedy algorithm to compute a maximum independent set for G. Prove or disprove that your greedy algorithm will always generate a maximum independent set for G.

8. Traveling Salesperson problem (TSP).
Given a completed graph G = (V,E) with n vertices and a cost function w: E $\rightarrow R^+$. A *tour* C is a cycle containing all n vertices in V once and exactly once, and a minimum tour is a tour C* such that $\sum_{e \in C} w(e)$ is minimum.

Q: Given G, design and analyze a greedy algorithm to compute a minimum tour for G. Prove or disprove that your greedy algorithm will always generate a minimum tour for G.