Greedy Algorithms: Main Ideas

Car Fueling problem:

Input: A car which can travel at most L kilometers with full tank, a source point A, a destination point B and n gas stations at distances $x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_n$ in kilometers from A along the path from A to B.

Output: The minimum number of refills to get from A to B, besides refill at A.

Subproblem:

A subproblem is a similar problem of smaller size.

Example:

$$\text{LargestNumber}(3, 9, 5, 9, 7, 1) = \text{"9"} + \text{LargestNumber}(3, 5, 9, 7, 1)$$

Min number of refills from A to B = first refill at G + min number of refills from G to B.

Safe move:

A greedy choice is called safe move if there is an optimal solution consistent with this first move.

Lemma:

To refill at the farthest reachable gas station is a safe move.

Proof:

Let route R with the minimum number of refills. G_1 be position of first refill in R. G_2 be next stop in R (refill or B). G be the farthest refill reachable from A. If G is closer than G_2, refill at G instead of G_1. Otherwise, avoid refill at G_1.

Case 1 (G is closer than G_2)

A -- G_1 -- G -- G_2 -- B

Refill at G instead of G_1 does not change the optimality because G is the farthest reachable point. An optimal path can be from A to G, from G to G_2, and from G_2 to B. So, it's a safe move.

Case 2 (G_2 is closer than G)

Avoid filling at G_1 and G_2 because G is reachable and the farthest. It contradicts that R is an optimal route. There is no such case.

```
MinRefills(x, n, L):
numRefills <- 0, currentRefill <- 0
```

```
while currentRefill <= n:
    lastRefill <- currentRefill
    while (currentRefill <= n and x[currentRefill + 1] - x[lastRefill]
<= L):
        currentRefill <- currentRefill + 1
        # We can continue to travel as currentRefill + 1 is reachable
from lastRefill position
    if currentRefill == lastRefill:
    # If currentRefill = lastRefill, not enough fuel to move to the next
position.
    # So return impossible.
        return IMPOSSIBLE
    if currentRefill <= n:
    # If currentRefill position is not at point B, we need to do
refilling
        numRefills <- numRefills + 1
return numRefills
```

Lemma:

The running time of `MinRefills(x, n, L)` is `O(n)`.

Proof:

1. `currentRefill` changes from 0 to $n + 1$. As inner while loop and outer while loop change the same variable, there is at most `O(n)` iterations even if there are two while loops.
2. `numRefills` changes from 0 to at most n, one-by-one.
3. Thus, `O(n)` iterations.

Greedy Algorithms: Grouping Children

Problem statement: Many children came to a celebration. Organize them into the minimum possible number of groups such that the age of any two children in the same group differ by at most one year.

```
MinGroups(C):
m <- len(C) # Intialize with the number of children
for each partition into groups
C = G[1]∪G[2]∪...∪G[k]: # Intially, k = m
    good <- true
    for i from 1 to k:
        if max{G[i]} - min{G[i]} > 1:
            good <- false
    if good:
        m <- min{m, k}
return m
```

Lemma:

The number of operations in `MinGroups(C)` is at least 2^n, where n is the number of children in C.

Proof:

Consider just partitions into two groups i.e. $C = G_1 \cup G_2$. For each $G_1 \subset C$, $G_2 = C \setminus G_1$. Note that size of C is n. Each item can be included or excluded from G_1. There are 2^n different G_1. Thus, at least 2^n operations.

Covering points by segments

Input: A set n points $x_1, \ldots, x_n \in \mathbb{R}$.

Output: The minimum number of segments of unit length needed to cover all the points

Save move: Cover the leftmost point with a unit segment which starts in this point.

```
Assume x[1] <= x[2] <= ... <= x[n]. (Points are sorted)
PointsCoverSorted(x[1], ..., x[n]):
R <- {}, i <- 1
while i <= n:
    [l, r] <- [x[i], x[i] + 1] # Unit length segment
    R <- R∪{[l, r]}
    i <- i + 1
    while i <= n and x[i] <= r:
    # If x[i] is still within unit length segment, we don't create a new segment for it.
    # So, add 1 to i.
        i <- i + 1
return R
```

Lemma:

The running time of PointsCoverSorted is `O(n)`.

Proof:

As i changes from 1 to n, for each i, there is at most 1 new segment. Overall, the running time is `O(n)`.

Greedy Algorithms: Fractional Knapsack

Fractional knapsack

Input: Weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ of n items; capacity W.

Output: The maximum total value of fractions of items that fit into a bag of capacity W.

Lemma (Safe move):

There exists an optimal solution that uses as much as possible of an item with the maximal value per unit of weight.

```
Knapsack(W, w[1], v[1], ..., w[n], v[n]):
A <- [0, 0, ..., 0], V <- 0
repeat n times:
    if W = 0: # No capacity and so select nothing
        return (V, A)
    select i with w[i] > 0 and max v[i]/w[i] # Pick an item with max
value per unit of weight
    a <- min(w[i], W) # Check if weighting exceed capacity
    V <- V + a * v[i]/w[i] # Add values by fraction
    w[i] <- w[i] - a # Reduce fraction of item weights by the amount put
into the bag
    A[i] <- A[i] + a # How each item is put into the bag
    W <- W - a # Reduce capacity by the fraction
return (V, A)
```

Lemma:

The running time of `Knapsack` is `O(n^2)`

Proof:

Select the best item on each step in `O(n)`. Main loop is executed n times. Overall, `O(n^2)`.

Assume $\frac{v[1]}{w[1]} \geq \frac{v[2]}{w[2]} \geq \ldots \geq \frac{v[n]}{w[n]}$.

```
Knapsack(W, w[1], v[1], ..., w[n], v[n]):
A <- [0, 0, ..., 0], V <- 0
for i from 1 to n:
    if W = 0: # No capacity and so select nothing
        return (V, A)
    a <- min(w[i], W) # Check if weighting exceed capacity
    V <- V + a * v[i]/w[i] # Add values by fraction
    w[i] <- w[i] - a # Reduce fraction of item weights by the amount put
into the bag
    A[i] <- A[i] + a # How each item is put into the bag
    W <- W - a # Reduce capacity by the fraction
return (V, A)
```

Now each iteration is `O(1)` and `Knapsack` after sorting is `O(n)`. So, `Sort` + `Knapsack` is `O(n*log(n))`.