# University of Waterloo
# CS341 - Winter 2016
# Assignment 4

**Due Date: Thursday Mar 17 at 11:59pm**

## Problem 1 Longest Common Subsequence

**Answer:** a4q1.cpp

## Problem 2 Facility Location

**Answer:**
We can get the subproblems: each position $p_i$ for $1 \leq i \leq m$, each people $a_j$ for $1 \leq j \leq n$, the number of hospital left is $h$ for $1 \leq h \leq k$, and the previous(left) hospital position is L(if $h = k$, $L = 0$). And there are $m * k$ subproblems.
The function dp(p,a,h,L):

- p: current position
- a: current people
- h: the number of available hospitals
- L: the previous position

We should consider following 2 possibilities:

1. Put the hospital in this position $p_i$: In this case, we should update the distance $dist$, $\sum_{a_j}^{until\ a_{j+s} > p_i} min(a_j - L,\ p_i - a_j) + dp(p_{i+1}, a_{j+s}, h-1, p_i)$.
2. Don't put the hospital in this position $p_i$: In this case, we don't need to update $dist$, $dp(p_{i+1}, a_j, h, L)$.

Recurrence:
We know that the base case is when $k = m$, at this time, we don't need to consider k as a variable, just compute all position. For each step, we will update the total distance with a list of minimum distance between $a$ and $p$ if the hospital is putted in this position; not update otherwise. So, finally we can surely get the minimum total distance to the people.

Algorithm:

```
1.dp(p,a,h,L): (output a number)
2.   if h = 0 then
3.      return (sum(a_j - L) until a_(j+s) = a_n)
```

```
else if put a hospital in p_i then
   if L = 0 then
      return (sum(p_i - a_j) until a_(j+s) > p_i) + dp(p_(i+1),a_(j+s),h-1,p_i)
   else then
      return (sum(min(a_j - L , p_i - a_j)) until a_(j+s) > p_i)
             + dp(p_(i+1),a_(j+s),h-1,p_i)
else then (not put hospital)
   return dp(p_(i+1),a_j,h,L)
```

Time complexity:
Generally, each subproblem takes $\frac{n}{m}$ operations and we got $m * k$ subproblems.
So, it costs $O(m * k * \frac{n}{m}) = O(nk)$.

## Problem 3 Coin Game

**Answer:**
We can get the subproblems: in a turn, for $1 \leq i < j \leq n$, let $v_i$ is the leftmost coin and $v_j$ is the rightmost coin.

We should consider following 2 possibilities:

1. Player 1 chooses the i-th coin(leftmost) with value $v_i$, then player 2 can either chooses (i+1)-th coin or j-th coin. Player 2 will choose the coin which can make player 1 with minimum value.
   So, player 1 can collect values: $v_i + min(dp[i+2][j], dp[i+1][j-1])$.

2. Player 1 chooses the j-th coin(rightmost) with value $v_j$, then player 2 can either chooses i-th coin or (j-1)-th coin. Player 2 will choose the coin which can make player 1 with minimum value.
   So, player 2 can collect values: $v_j + min(dp[i+1][j-1], dp[i][j-2])$.

Recurrence:
We know that $dp[i][j]$ is the maximum value that player 1 can earn from the i-th coin to the j-th coin. So,
$dp[i][j] = max(v_i + min(dp[i+2][j], dp[i+1][j-1])\ , v_j + min(dp[i+1][j-1], dp[i][j-2]))$
And the base case should be:

1. $dp[i][j] = max(v_i, v_j)$ when $j = i + 1$.

2. $dp[i][j] = v_i$ when $j = i$.

For each step, we just need to calculate the $dp$ value for the certain length of v and add up until length equals n. So, we will use dynamic programming to avoid the overlapping subproblems; otherwise, dp[i+1][j-1] would be calculated twice.

Algorithm:

```
Given a list of values, v[1..n]; and the number of coins, n
Output the maximum total value player 1 earns
Find_Max(v[1..n],n): (below, we see v[1..n] as v[0..n] as array)
   dp[n][n], set each value 0
   for i from 0 up to (n - 1) do
      dp[i][i] <- v[i]
   end loop
   for len from 2 to n do
      for i from 0 to (n - len) do
         j <- i + len - 1
         if len equals to 2, then
            dp[i][j] <- max(v[i],v[j])
         else, then
             dp[i][j] = max(v[i] + min(dp[i+2][j],dp[i+1][j-1]),
                           v[j] + min(dp[i+1][j-1],dp[i][j-2]))
      end loop
   end loop
   return dp[0][n-1]
```

Time complexity:
Obviously, we do $(n-1+n-2+..+1+0)$ operations, so the time cost: $O(\frac{(n-1)n}{2}) = O(n^2)$.

## Problem 4 Covering Set

**Answer:**
As we know, the input is a tree $T = (V, E)$ in the adjacency list representation, a weight $w_v$ for each $v \in V$. In other words, we input a list of weight numbers to represent the nodes.

We can get the subproblems: for $v \in V$, size of smallest covering set in subtree rooted at $v$. And there are n subproblems, where n is $|V|$.

We should consider following 2 possibilities for root and recursively for all nodes down the root.

1. Root is in the Covering Set: In this case, root covers all children edges. And left with children subtrees.

2. Root is not in the Covering Set: In this case, all children must be in cover otherwise the edges adjacent to v will not be covered. And left with grandchildren subtrees.

Recurrence:
From above, the case 1 we can get: $sol1 \leftarrow 1 + sum(dp(c)\ for\ c\ in\ v.children)$
the case 2 we can get: $sol2 \leftarrow (num\ of\ v.children) + sum(dp(g)\ for\ g\ in\ v.grandchildren)$
So, $dp(v) = min(sol1, sol2)$. And the solution for this question should be $dp(root)$.

Algorithm:

```
Given a list of nodes, set the size of cover set vc, vc <- 0
Output a covering set cs
dp(*root): (pointer of root, containing the whole tree)
  if (root is NULL or there is only one node), then
     return 0
  if (root->vc is not 0), then
     return root->vc
  sol1 <- 1 + sum(dp(c) for c in v.children)
  sol2 <- (num of v.children) + sum(dp(g) for g in v.grandchildren)
  root->vc = min(sol1, sol2)
  if (sol1 < sol2), then
     put this root into the cs
  return root->vc
end of function dp
Finally, we can get the Covering Set, cs
```

Time complexity:
Because the edges going out of node $v$ are visited at most twice by the recursion: one for computing parent, and one for grandparent.
Therefore, actually the time costs: $O(num\ of\ subproblems) = O(n)$.

## Problem 5 Trading

**Answer:**
For this question, we can use Bellman-Ford algorithm to do all-pair shortest path to do loops by n items $a_1..a_n$.

We get the subproblem: $D[i, j]$ is the maximum amount of item $a_j$ which can be traded by $a_i$. And $P[i, j]$ is the previous item of $a_j$ starting trading from item $a_i$.

Recurrence: (base on the correctness of Bellman-Ford Algorithm)
For the loop, we get the value
$D[i, m] = max(max(D[i, k] * T[k, m])\ for\ all\ k\ that\ 1 < k \leq n, D[i, m])$.
In the loop, when it is in the i iteration:

- If there exists a trading path from $a_i$ to $a_j$, $D[i, j]$ is at most the product of i weights.
- If $D[i, j] \neq 0$, $D[i, j]$ is $D[i, k] * T[k, m]$.

The base case is: $i = 1$, the $D[s, s] = 1$(s represents source), and for all other $D[i, j]$ should be 0 because there are no path through them yet.

For the inductive case, an item is updated by $D[i, m] = D[i, k] * T[k, m]$, where $D[i, k]$ is the path from $a_i$ to $a_k$ and $D[i, m]$ is the path from $a_i$ to $a_m$. If every edge with less-than-1 trading rate, then each item is visited at least once. So if there are no improvement(times a larger-than-1 rate) in n-th loop, then for any cycle $D[i, i]..D[i, k - 1]$, the final rate product should be less than 1. In addition, if there is a cycle with the product larger than 1, then in the n-th loop, we would get a path from $a_i$ to $a_j$ starting with $a_l$, $D[l, i] * T[i, j] > D[l, j]$, which means that the value of trading path is improved for every time doing loop.

Algorithm:

```
Trading:
 for i from 1 to n do
    for every j (1 <= j <= n) do
       D[i,j] <- 0
       P[i,j] <- nil
    D[i,i] <- 1
    for i from 1 to n-1 do
       for k from 1 to n do
          for m from 1 to n do
              if D[i,k]*T[k,m] > D[i,m] then
                 D[i.m] <- D[i,k]*T[k,m]
                 P[i.m] <- k
    for k from 1 to n do
       for m from 1 to n do
          if D[i,k]*T[k,m] > D[i,m] then
             return true
 return false
```

Time complexity:

Bellman-Ford Algorithm costs $O(|V||E|) = O(n * n^2) = O(n^3)$. And we run it for n times, so the final time should be $O(n^3 * n) = O(n^4)$.