COMP3821 Assignment 3

April 2021

1.1 **Solution:** The height of the recursion tree will have 2^{n-1} leaves, and many sub-problems are repeatedly solved. Thus the time complexity of this procedure is exponential, $O(2^n)$.

1.2 **Solution:**

Pseudo-Code:

```
// let r[0..n][0..k] be a global lookup array
    for i=0 to n
        for j=0 to k
            r[i][j] = -1

memoized-valid-numbers(n, k) {
    // base case
    if n = 1 then
        return min(k,9)

    if (r[n][k] != -1)
        return r[n][k]

    q = 0
    for d = 0 to min(k,9) do
        q = q + memoized-valid-numbers(n-1, k-d)

    r[n][k] = q
    return q

}
```

1.3 **Solution:** Ignoring the time complexity of creating a lookup table, the time complexity of memoized-valid-numbers(n, k) is $0(n * k)$ which resolves to $O(n^2)$.

1.4 **Proof:** Using induction, we can verify that both algorithms return the same result for a given input using induction. The algorithms return optimal solutions.

Base case: let n=1, k=1, then valid-numbers(1, 1) returns 1. memoized-valid-numbers(1,1) also returns 1.

Induction hypothesis: For some $n, k >= 1$, assume valid-numbers(n-1, k-1) returns x and memoized-valid-numbers(n-1,k-1) also returns x.

Induction step: Prove that it is true for n,k.
As per the algorithm, valid-numbers(n,k) and memoized-valid-numbers(n,k) will make recursive calls. From the hypothesis, we know that calls to valid-numbers(n-1, k-1) = memoized-valid-numbers(n-1,k-1) are taken care of. This leaves us with the case valid-numbers(n-1,k) and memoized-valid-numbers(n-1,k).

Since All the sub-problems of the both algorithms are optimal, the call to this case is also optimal. This is because if the sub-problems are sub-optimal then this would contradict the fact that the solution is optimal.
Since the algorithms are optimal, and every case until (n-1,k-1) is optimal, it follows that by induction the case (n-1,k) is also optimal. Thus by induction, we have proven that the algorithms return the same result.

2.1 **Solution:** To compute the maximum weighted independent set, let the tree T(n) be rooted at node m. Let O be an optimum solution to the question.

Starting at root node m,
- If $r \in O$: None of m's child nodes can be in O. $O - \{m\}$ contains an optimum solution for each subtree of T hanging at a grandchild of m.
- If $r \notin O$: O contains an optimum solution for each subtree of T hanging at a child node of m.

Thus the optimal solution can have either of the two properties below:

- 1) The maximal sum of v_w's selected from subtree T(m) which includes root node m and satisfies constraints
- 2) The maximal sum of v_w's selected from subtree T(m) which does not include root node m and satisfies constraints

The base cases: For an empty tree, the weight is zero, for a tree with with one node, the node is in the maximal set and it's weight is the maximum weight.

The above two properties can be tested by recursion through subtrees in T, where the subproblem are subtrees of T. This algorithm will have an expoential running time, that once memoized, will reduce to a linear running time.

The reccurence relation between the subproblems is as follows:

$$O(m) = max \left\{ \begin{array}{l} \sum O(x). \text{ (x is a child of m)} \\ v_w + \sum O(x). \text{ (x is a grandchild of m)} \end{array} \right\}$$

.

2.2 **Solution:** Since we don't know beforehand whether the optimal solution contains 1) or 2) (from 2.1), we can can recurse through both options to come to a solution, and that solution would have an exponential running time, which would be the worst-case time complexity. We can then memoize the algorithm, thus the cost is amortized and the algorithm will have a linear runtime.

3.1 **Solution:** The dynamic programming algorithm to compute q(n) is as follows:

To solve this problem let us consider the fact that we are partitioning n marbles into distinct partitions/groups.

Let us have a partition array p[] that stores partition values; values <= n that are used to sum to n. We must recursively go from n to 1 (since 1 is the minimum possible partitioned value) and keep appending numbers used to form a sum in p[].

When this sum is n, we increment a counter variable *count* that keeps track of the number of ways to divide n marbles. We start each partition of n with the maximum number of the previous partition, stored in a variable $maxVal$, which is initially set to n. The variable $remainingSum$ is caclualted as $n - i$ (i is a variable decrements from $maxVal$ to 1).

When we append numbers to the array, we check to see if that number is already present in the partition, if so we ignore this iteration and do not update the counter. This is because we want unique groups to avoid instances like [2,2], when n=4 for example. $count - 1$ will remove the singleton set and return the correct output.

Thus, the base case is when $remainingSum = 0$ as that means the partition has summed to n. The subproblems are calculating the number of partitions for the $remainingSum$.

3.2 **Solution:** The time complexity of this algorithm is $0(2^n)$ since there are 2^n possible partitions that we go through. Using a lookup table for $remainingSum$ subproblems in a memoized implementation of the above algorithm results in a linear running time.

4. **Solution:** To design an efficient algorithm we can use a modified version of the Floyd-Warshall algorithm (which calculates the shortest path between all pairs in a graph). Let A[i,j] denote the probability of getting from island i to island j.

A[i,i] = 0 (the same island so not relevant), and initially:
A[i,j] = ∞ (for some islands i and j that do not have a direct path calculated)

Modifying the calculations for A[i,j] in the Floyd-Warshall algorithm:
in the third inner loop of the algorithm
A[i,j] = min(A[i,j] , (A[i,k]) * (A[k,j]))

where k increments from 1 to n (n is the number of islands as given). A lower probability value equates to a safer crossing. The matrix $A^n[i, j]$ will represent the probability of the safest crossings between 2 islands as it will have a minimal probability.

(As a side note, it might be worth setting custom multiplications for probability values when $p_e = 1$ or 0, so someone crossing a bridge that is guaranteed to break won't actually cross it. That would be most unfortunate.)

Since the algorithm takes $O(n^2)$ for one matrix, for n matrices, the running time for the algorithm is $O(n^3)$.

5. **Solution:** Have an array that stores the vertices in a clockwise or anti-clockwise orientation. Follow the basis that if two line segments that do not have common endpoints will not intersect if their endpoints are also in this clockwise orientation (the start and end points are at corresponding locations in the array).
For all vertices, draw all legal lines (where $a_{i,j} = 1$) while maintaining this orientation. If there is an illegal line attempted to being drawn, check if the

endpoints of the lines in conflict are the same (then it's legal and can be drawn). Ensure that the vertex is not the start and endpoint at the same time. Continue to to do the above until n-1 line segments are drawn.

The function of drawing legal lines for n-1 times will have $O(n^2)$ running time. For n different orientations of the lines, the running time will be $O(n^3)$.

END.