**Last Name:** *Chen* **First Name:** *Robert* **Email:** *rchen87@gatech.edu*


**Problem 1: Subset Sum, Dynamic Programming (25 points)**
You are given $n$ items $\{1, 2, \ldots n\}$, where each item has a given positive weight $w_i$, $1 \leq i \leq n$. You are also given an upper bound $W$. You would line to select a subset $S$ of the items so that $\sum_{i \in S} w_i \leq W$, and, subject to this restriction, $\sum_{i \in S} w_i \leq W$ is as large as possible. Give an $O(nW)$ algorithm, justify correctness and running time.

**Answer:**

We are going to solve this with dynamic programming.

```
## subset sum, dynamic programming - pseudocode

## given n items (each with increasing weight), and W upper bound
## compute the best set of items to put in the set S
## so that you get the highest weight thats still
## less than the max weight W

## 1. build look-up table M
## 2. work backwards from max weight to find
##    what items to put in the set S
##

## n: number of items
## w: array with weight of each item ; n total items
## W: max weight the sum can be
##

w = array of weights
def subset_sum(n, W):
    ## initialize the look-up table
    for r = 0, ... ,W
        M[0,r] = 0
    for j = 1, ..., n
        M[j,0] = 0

    for j = 1, ... , n
        for r = 0 , ... , W
            if w[j] > r:
                M[j,r] = M[j-1, r] #if this item too heavy, cannot include;
                                   # therefore max sum is whatever it was
                                   # with the all items up to the last j
            M[j , r ] = max(  M[j-1, r],
                              w[j] + M[ j-1, W-w[j] ] )  #find max of these two
    return M[n,W] ## after table has been filled, we simply look it up
```

There are $n \times W$ items in the lookup table. Therefore it takes $O(nW)$ to fill up the table. It takes $O(n)$ to follow the path backwards to get the answer. Therefore $O(nW + n) = O(nW)$.

**Problem 2: Balanced Tree, Dynamic Programming (25 points)**.
Let $T(V, E)$ be a directed acyclic graph with $V = \{v_1, \ldots v_n\}$.
Suppose that $T$ is given in topologically sorted order, that is, if $v_i$ is an ancestor of $v_j$ then $i < j$.
Suppose further that each vertex $v_i \in V$ has a given positive cost $c(v_i) > 0$. Define the weight of a vertex $v_i \in V$ as the sum of the costs of all vertices that can be reached from $v_i$ (equivalently belong to the subtree rooted at $v_i$) : $weight(v_i) = \sum_{\substack{v_j \in V: \\ v_j \text{ is reachable from } v_i}} c(v_j)$

Say that $T$ is balanced if and only if, for every vertex $v_i \in V$, if $v_i$ has children $u_1, \ldots, u_k$, then

$$weight(u_1) = weight(u_2) = \ldots = weight(u_k)$$

Give a polynomial time algorithm that decides if a directed tree with costs on its vertices is balanced. Justify your answer and argue running time.

**Answer:**

**Problem 3: Max Independent Set, Dynamic Programming (25 points)**
(a) Consider a line graph on vertices $\{1, \ldots, n\}$ and edges $\{1, 2\}$, $\{2, 3\}$, ... , $\{(n-1), n\}$. Each vertex has a positive weight $w_i$, $1 \leq i \leq n$. Give an $O(n)$ algorithm that outputs the weight of a maximum weight independent set of the line graph. You may give a simple description of the algorithm, and/or pseudocode. You should include a short argument of correctness and running time.
(b) Consider a cycle graph on vertices $\{1, \ldots, n\}$ and edges $\{1, 2\}$, $\{2, 3\}$, ... , $\{(n-1), n\}$, $\{n, 1\}$. Each vertex has a positive weight $w_i$, $1 \leq i \leq n$. Give an $O(n)$ algorithm that outputs the weight of a maximum weight independent set of the cycle graph. You may give a simple description of the algorithm, and/or pseudocode. You should include a short argument of correctness and running time.

**Answer:**
Recall that a subset S of a graph G is an independent subset if there are no edges between any two elements in S.

**Part a**

Let MIS = max independent set. So say you have vertex $v_i$, then one of two scenarios exists: 1. $v_i$ is IN the MIS of $v_i$ and later, or 2. $v_i$ is NOT in the MIS and thus the MIS is the MIS of vertices after $v_i$. If $v_i$ is IN the MIS from $v_i$ and later, then the MIS is the weight of $v_i$, plus the total weight of the MIS from $v_{i+2} to v_n$. Otherwise, then the MIS from $v_i to v_n$ is the MIS from $v_{i+1} to v_n$. If you model the recurrence this way then the MIS from any $v_i to v_n$ is:

$$MIS(v_i) = MAX\{MIS(v_{i+1}), w_i + MIS(v_{i+2})\}$$

This can be solved in linear time ($O(n)$) because its a line and that the subproblem for each subsequent vertex is smaller than the one before it.

**Part b**

This is the same problem as part a except that we have a cycle. Arbitrarily choose to cut the graph at any edge and now you have a line graph. Find MIS of this in the way as in **part a**. Running time is $O(n)$ following the same logic.

**Last Name:** *Chen* **First Name:** *Robert* **Email:** *rchen87@gatech.edu*


**Problem 4: Longest Path, Dynamic Programming (25 points)**
Let $G(V, E)$ be a directed acyclic graph, where $V = \{v_1, \ldots, v_n\}$. The graph is presented in adjacency list representation, and with the property that $v_i \to v_j \in E$ only if $i < j$. Give an $O(|V| + |E|)$ algorithm that finds the length of the longest path (maximum number of edges) from $v_1$ to $v_n$. If there is no path from $v_1$ to $v_2$ then your algorithm should output $\infty$.
Give a short justification of correctness and running time.

**Answer:**

This can be solved with dynamic programming. It is $O(|V| + |E|)$ because you go thru all vertices in topological order and evaluate all edges once (all edges of each adjacent vertex in question).

```
## longest path, dynamic programming - pseudocode

## find length of longest path from v_1 to v_n
## in DAG G.
## assume G is topological sorted already
##


def longest_path_v1_vn(G, v_1, v_n):
    length_to_this_vertex = []
    for i in range(n):
        length_to_this_vertex.append( infinity) ## initialize all lenghths to
                                                ## vertices from source
                                                ## to be infinity

    top_sorted_G = topolotical sort G ## O(|V| + |E|)

    ## loop thru edges in topologically sorted order
    for i in range(len(top_sorted_G)):
        for j in range(i, len(top_sorted_G)): # it's an edge of v_j comes later than v_i
                                              # in top_sorted_G
            ## add the weight of edge v_i -> v_j if it's going to increase greatest length to v_j so far
            if length_to_this_vertex[top_sorted_G[j]] <= length_to_this_vertex[top_sorted_G[i]] + weight(G, i,j):
                length_to_this_vertex[top_sorted_G[j]] = length_to_this_vertex[top_sorted_G[i]] + weight(G, i,j)

    return max( length_to_this_vertex ) #returns the LENGTH of the longest path
```