**GIVE PSEUDO CODE TO RECONSTRUCT AN LCS FROM THE COMPLETED $c$ TABLE AND THE ORIGINAL SEQUENCES $X = \langle x_1, x_2, \dots, x_m \rangle$ AND $Y = \langle y_1, y_2, \dots, y_n \rangle$ IN $O(m+n)$ TIME WITHOUT USING THE $b$ TABLE.**

Since the b table is created to simplify the process of printing the longest common string, and since it is created using specific cases within the $c$ table, we can replace the calls to the $b$ table with the specific cases in the $c$ table to create a PRINT-LCS without the use of the $b$ table.

```
PRINT-LCS(c, X, Y, i, j)
if i == 0 or j==0
        return
if x_i == y_j
        PRINT-LCS(c, X, Y, i-1, j-1)
        print x_i
else if c[i-1, j] ≥ c[i, j-1]
        PRINT-LCS(c, X, Y, i-1, j)
else
        PRINT-LCS(c, X, i, j-1)
```

Considering that each of these new cases can be computed in $O(1)$ time, the existing time complexity of $O(m+n)$ still holds, as it decrements either $i$, $j$, or both in each recursive call.

**SUPPOSE THAT IN A 0-1 KNAPSACK PROBLEM, THE ORDER OF THE ITEMS WHEN SORTED BY INCREASING WEIGHT IS THE SAME AS THEIR ORDER WHEN SORTED BY DECREASING VALUE. GIVE AN EFFICIENT ALGORITHM TO FIND AN OPTIMAL SOLUTION TO THIS VARIANT OF THE KNAPSACK PROBLEM, AND ARGUE THAT YOUR ALGORITHM IS CORRECT.**

Set-up:

Knowing that each item has a weight and value, let $I = \langle i_1, i_2, \dots, i_n \rangle$ be the set of n items, $V = \langle v_1, v_2, \dots, v_n \rangle$ be the set of values for each $n^{th}$ item and $W = \langle w_1, w_2, \dots, w_n \rangle$ be the set of weights for each $n^{th}$ item. We can create the table:

| | | | | |
|---|---|---|---|---|
| $i_1$ | $i_2$ | $i_3$ | | $i_n$ |
| $w_1$ | $w_2$ | $w_3$ | | $w_n$ |
| $v_1$ | $v_2$ | $v_3$ | … | $v_n$ |

where each item has a corresponding value and weight, and $V$ and $W$ share the following relationship:

$$w_1 \leq w_2 \leq w_3 \leq \cdots \leq w_n$$

$$v_1 \geq v_2 \geq v_3 \geq \cdots \geq v_n$$

Considering that in this relationship, the items with the highest values are also those with the lowest weight, a greedy algorithm would be excellent in choosing the items to optimize the value in the bag. This is because the current problem exhibits both the *greedy-choice property* and an *optimal substructure*.

The greedy-choice property is exhibited through the following proof:

Let $B$ be the knapsack, of maximum weight capacity $M$, with the optimal selection of items. If $i_1$ is the object with the highest value and least weight, then we can safely assume that $i_1 \in B$ as the most optimal item must be part of the optimal set of items. However, if $i_1 \notin S$, then let $i_k$ be the least optimal item in the backpack $B$, then $i_1$ is in backpack $B'$, which does not include $i_k$. Considering that $w_1 \leq w_k$, the weight of $B' \leq$ the weight of $B$, which are both $\leq M$, and are thus valid backpack arrangements. Since $v_1 \geq v_k$, the value of $B'$ must be $\geq$ the value of $B$, which contradicts the notion that $B$ is the optimal set. Thus, the optimal set must include the most optimal item in the subset.

The optimal structure property can be proven quite simply. If we let $B_j$ be the optimal backpack with the $j$ optimal items, and knowing that $i_1$ is one of them, $B_{j-1} = B_j \setminus i_1$. We know this is optimal for the set of items $\langle i_2, i_3, \dots, i_n \rangle$, where $M_{j-1} = M_j - w_1$. If $B_{j-1}$ is not optimal, then $B_j$ can be improved by improving its predecessor $B_{j-1}$. ■

Now that it has been proved that this is a problem that can be solved using a greedy algorithm, the algorithm is as follows:

```
BestKnapsack(w, L)
In: the maximum weight, w, and the list of items L
Out: the optimal backpack arrangement, B
Let B be the optimal backpack
for i = 0 to n and w - L[i].weight ≥ 0
        S[0] = L[i]
        w = w - L[i].weight
end for
return B
```

**SUPPOSE YOU ARE GIVEN TWO SETS $A$ AND $B$, EACH CONTAINING N POSITIVE INTEGERS. YOU CAN CHOOSE TO REORDER EACH SET HOWEVER YOU LIKE. AFTER REORDERING, LET $a_i$ BE THE $i^{th}$ ELEMENT OF SET $A$, AND LET $b_i$ BE THE $i^{th}$ ELEMENT OF SET $B$. YOU THEN RECEIVE A PAYOFF OF $\prod_{i=1}^{n} a_i^{b_i}$.**

**GIVE AN ALGORITHM THAT WILL MAXIMIZE YOUR PAYOFF. PROVE THAT YOUR ALGORITHM MAXIMIZES THE PAYOFF, AND STATE ITS RUNNING TIME.**

Sorting $A$ and $B$ in either increasing or decreasing order would yield the highest payoff. The proof for this is as follows:

Assuming the sets are ordered in decreasing order,

$$a_1 \geq a_2 \geq \cdots \geq a_n$$

$$b_1 \geq b_2 \geq \cdots \geq b_n$$

given $p < q$ $a_p^{b_p} \geq a_q^{b_q}$ for all $p > n$ and $q \geq n$ in $S$, the optimal set. Assuming that the above solution does not hold and $S$ is not the optimal set, then let $S'$ be the optimal set where $a_p^{b_q}$ and $a_q^{b_p}$ replace the two previous terms appropriately. Knowing that $a_p \geq a_q$ and $b_p \geq b_q$ we know that:

$$\frac{Payout\ S}{Payout\ S'} = \frac{\prod_S a_i^{b_i}}{\prod_{S'} a_i^{b_i}} = \frac{a_p^{b_p} \cdot a_q^{b_q}}{a_p^{b_q} \cdot a_q^{b_p}} = \left(\frac{a_q}{a_p}\right)^{b_p - b_q} < 1$$

which contradicts the notion that $S'$ is more optimal than $S$. This process can be repeated for the remaining elements in the lists. ■

Assuming the elements are already sorted before calculating the payoff, the algorithm is of $O(n)$. Should sorting be included in the time complexity, the algorithm becomes of $O(n \cdot \log n)$ should a comparison sort (i.e. heapsort) be used.

Sorting in increasing order would yield the same result since the order of multiplication is irrelevant.

**SHOW THAT NO COMPRESSION SCHEME CAN EXPECT TO COMPRESS A FILE OF RANDOMLY CHOSEN 8-BIT CHARACTERS BY EVEN A SINGLE BIT. (HINT: COMPARE THE NUMBER OF POSSIBLE FILES WITH THE NUMBER OF POSSIBLE ENCODED FILES.)**

A compression scheme relies on the probability that a character will appear multiple times in a sequence. Thus, should a file be constructed with randomly selected characters, the likelihood of their being a pattern in the sequence of characters is minimal due to the high entropy caused by the randomness. This situation requires that, for each potential unique variation of the file, there be a unique variation of its encoding. That being said, given that the files are of n bits, there are $2^{n+1} - 1$ possible files, and thus, the same number of encodings. Since any compression algorithm must maintain this uniqueness property, it is not possible to expect compression of such a file.

**COMPUTE THE *next*[ ] FUNCTION (THE PREFIX FUNCTION $\pi$ IN THE TEXT BOOK) FOR THE PATTERN *P=babbabbabbababbabb*.**

| | **b** | **a** | **b** | **b** | **a** | **b** | **b** | **a** | **b** | **b** | **a** | **b** | **a** | **b** | **b** | **a** | **b** | **b** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| i: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| n: | 0 | 0 | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 2 | 3 | 4 | 5 | 6 | 7 |

**MODIFY THE KMP STRING MATCHING ALGORITHM TO FIND THE LARGEST PREFIX OF $P$ THAT MATCHES A SUBSTRING OF $T$. IN OTHER WORDS, YOU DO NOT NEED TO MATCH ALL OF $P$ INSIDE $T$; INSTEAD, YOU WANT TO FIND THE LARGEST MATCH (BUT IT HAS TO START WITH $p1$).**

Understanding the nature of the KMP string matching algorithm, we can recycle the existing algorithm and make minor changes to find the longest substring of $P$ found within $T$ like so:

```
KMP-MATCHER(T, P)
n = T:length
m = P:length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0
maxQ = 0

for i = 1 to n
        while q > 0 and P[q + 1] ≠ T[i]
                q = π[q]
        end while
        if P[q + 1] == T[i]
                q = q + 1
        end if
        if q == m
                print "Pattern occurs with a shift" + i − m
                q = π[q]
        end if
        if q > maxQ
                max = q
        end if
end for
for  i = 0 to maxQ
        result = result + P[i]
end for

return result
```

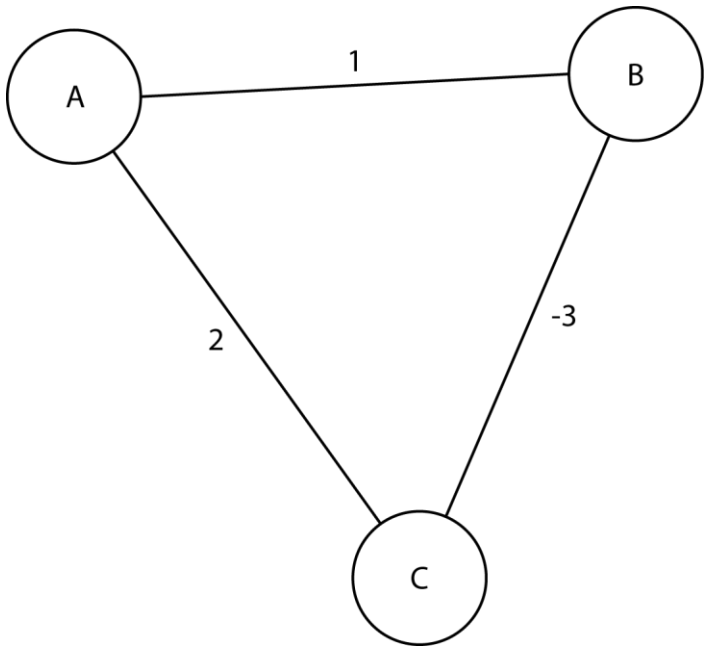
**MODIFY THE MINIMUM SPANNING TREE ALGORITHM TO FIND THE MAXIMUM SPANNING TREE.**

To yield a maximum spanning tree by modifying the minimum spanning tree algorithm, one simply has to use a Max-Heap instead of a Min-Heap. In making that change we must also modify the algorithm to EXTRACT-MAX instead of EXTRACT-MIN.

**FIND A COUNTER EXAMPLE THAT SHOWS DIJKSTRA'S ALGORITHM DOES NOT WORK WHEN THERE IS NEGATIVE WEIGHT EDGE.**

The greedy nature of Dijkstra's algorithm requires that the cost of each edge be positive so that the greedy-choice property holds under the assumption that each new edge comes with an increased cost. Foregoing this assumption means that a greedy algorithm would not suffice as the best choice in each subset must now take into account the optimal choice for each subset's subset, since any new edge may *decrease* the total cost of the traversal. An example that illustrates this notion:

In this graph, starting at $A$, Dijkstra's would consider $\overline{AB}$ and $\overline{AC}$. Since $\overline{AB} < \overline{AC}$ the algorithm will select $\overline{AB}$ as the optimal path from $A$ to $B$, yet $\overline{AC} + \overline{CB}$ is the truly optimal path to $B$.



**GIVEN A WEIGHTED DIRECTED GRAPH $G = (V, E)$ SUPPOSE THAT THERE ARE NEGATIVE WEIGHTS IN $G$, BUT THERE IS NO NEGATIVE CYCLE IN $G$. IS ALL-PAIR-SHORTEST-PATH ALGORITHM STILL CORRECT? PROVE YOUR ANSWER.**

The all-pair shortest path algorithm accounts for negative values by first preprocessing the vertexes and edges to create a direct path matrix and a variety of indirect path matrices. The direct path matrix contains the direct edges that connect two nodes together, along with their associated costs. The indirect matrices, contain the cheapest path from all nodes, through a particular node unique to each matrix, to all other nodes is saved in each matrix. The cost of each path is calculated by summing the cost at each node—negative or otherwise—from its starting vertex, to the intermediate vertex, to the end vertex. Additionally, because this fragmented edge is compared to the direct edge, which is then updated should the indirect edge be shorter than the direct edge. If the direct edge is not updated, then it is shorter than the indirect edge, and is the shortest edge. Because of this, multiple negative intermediates cannot shorten the path unless a negative cycle exists.

The matrices (direct and indirect) are then combined using matrix multiplication to ensure the optimal path remains for each connection. Since all the operations involve addition or multiplication, the negatively weighted edges will be accounted for in determining the optimal paths. It is imperative, however, that there exist no cycles that have a negative cost to traverse, as these cycles would never yield an optimal path to any vertices, as traversing this cycle will always cause the cost to decrease.

**LET $G = (V, E)$ BE A WEIGHTED DIRECTED GRAPH WITH NO NEGATIVE CYCLE. DESIGN AN ALGORITHM TO FIND A CYCLE IN $G$ WITH MINIMUM WEIGHT. THE ALGORITHM SHOULD RUN IN TIME $O(|V|^3)$.**

Considering that this graph does not necessarily have any negative edges, we can use Dijkstra's algorithm to help us find the shortest path from a vertex to all other vertices. We could then use these values to determine the weight of the shortest cycle.

```
SHORTEST-CYCLE(G,v)
minWeight = ∞
for each v in G
        DIKJSTRA(G,v)
        for each u∈[G]
                for each w incident on u
                        if w = v and minWeight > d[u] + WEIGHT(u,w) //d[u] is the distance of u
                                minWeight = d[u] + WEIGHT(u,w) //d[u] is the distance of u
                        end if
                end for
        end for
end for

return minWeight
```

Considering that there are $|V|$ iterations of the outermost loop, each calling DIKJSTRA, which has a worst case performance of $O(|V|^2)$. The inner loop also has $|V|$ iterations, where it makes, in the worst case $O(|V| - 1)$ calls. Putting all of this together we get a time complexity of:

$$
\begin{aligned}
&O\left(|V| \cdot \left(|V|^2 + |V| \cdot (|V| - 1)\right)\right) \\
&= O(|V| \cdot (|V|^2 + |V|^2 - |V|) \\
&= O(|V| \cdot (2|V|^2 - |V|) \\
&= O(|V|^3)
\end{aligned}
$$