Question 1

Approach. The methodology behind this algorithm is to use a modified version of binary tree searching (*from CLRS, page 290*) to find the first node with the given prefix x and use a modified version of an in-order traversal (*from CLRS, page 288*) to traverse until the node keys no longer contain prefix x.

Algorithm 1: Prefix Search

```
1  FIND-STRINGS-WITH-PREFIX(r, x):
2      f = PREFIX-TREE-SEARCH(r, x) // First node that matches prefix.
3      PREFIX-BASED-INORDER-WALK(f, x)
4      return
5  PREFIX-BASED-INORDER-WALK(r, x):
6      if r ≠ NULL and x == r.key[0 : len(x)] then
7          PREFIX-BASED-INORDER-WALK(r.left, x)
8          print r.key
9          PREFIX-BASED-INORDER-WALK(r.right, x)
10     return
11 PREFIX-TREE-SEARCH(r, x):
12     if r == NULL or x == r.key[0 : len(x)] then
13         return r
14     else if x < r.key[0 : len(x)] then
15         return PREFIX-TREE-SEARCH(r.left, x)
16     else
17         return PREFIX-TREE-SEARCH(r.right, x)
```

We assume in the above that the $<$ and $>$ operations are defined correctly for strings. ("ABC" $<$ "ACB")

Proof of correctness and termination. The binary tree in-order traversal procedure is equivalent to that in CLRS except for the condition that the key of the current node, r, must contain the prefix x. It operates the same as described in CLRS except for when the key of r does not contain the prefix x, at which point it terminates on line 10.

The tree searching procedure is equivalent to that in CLRS except for checking whether the key of the current node, r, contains the given prefix, x, instead of matching a given key. Therefore it operates the same as described in CLRS.

Based on the correctness of the in-order traversal procedure and the tree searching procedure we can prove the correctness of the main algorithm by contradiction. *Assume by contradiction* that there is a node matching the prefix that this algorithm does not find, there are two cases:

1. *The node is not in the sub-tree of node f found on line 2.* This is not possible in a binary tree, since, if a prefix-matching node existed outside the sub-tree then it would have been selected as f by the tree searching procedure.

2. *The in-order traversal ends before reaching a prefix-matching node.* This is not possible as the result of an in-order traversal is sorted and in a sorted list of strings, those sharing a matching prefix will always be consecutively next to one another.

Time and space complexity. The time complexity of this algorithm for n strings where k contain prefix x is the sum of each of the procedures: the search algorithm is $O(log(n))$ and the in-order traversal is $O(k)$ (since it will always stop running after k nodes). Therefore the time complexity overall is $O(log(n) + k)$ The space complexity is O(n) for the binary tree, since the algorithm does not store anything.

Question 2

Part A

Let $T(n)$ be the number of nodes in a full binary tree of height n (where $n \geq 1$). Then the recurrence representing $T(n)$ can be stated as:

$$T(n) = 2 \cdot T(n-1) + 1,\ T(1) = 1$$

This is logical given that adding 1 level of height to a full binary tree consists of doubling the previous level (two children per node) and adding 1 for the root node in the tree.

Part B

We want to show that there are more leaf nodes than non-leaf nodes in a full binary tree. A leaf node is a node with no children, which in this case is the last level of the binary tree.

For a tree of height n, the number of non-leaf nodes is represented by $T(n-1)$, therefore the number of leaf nodes can be written as:

$$\begin{aligned}
\#\text{ leafs} &= \#\text{ nodes} - \#\text{ non-leafs} \\
&= T(n) - T(n-1) \\
&= 2T(n-1) + 1 - T(n-1) \\
&= T(n-1) + 1
\end{aligned}$$

Since $T(n-1) + 1 > T(n-1)$, it is clear that the number of leaf nodes is always greater than the number of non-leaf nodes.

Question 3

1) $1 \mod 11 = 1$
2) $22 \mod 11 = 0$
3) $54 \mod 11 = 10$
4) $13 \mod 11 = 2$
5) $12 \mod 11 = 1 \rightarrow$ Collision!
   (a) $(1{+}3(12) \mod 4{+}1) \mod 11 = 3$
6) $3 \mod 11 = 3 \rightarrow$ Collision!
   (a) $(3+3(3) \mod 4+1) \mod 11 = 5$
7) $27 \mod 11 = 5 \rightarrow$ Collision!
   (a) $(5{+}3(27) \mod 4{+}1) \mod 11 = 7$

Index	Data	Step #
0	22	2
1	1	1
2	13	4
3	12	5
4		
5	3	6
6		
7	27	7
8		
9		
10	54	3

Question 4

The greedy algorithm works as follows:

1. Sort set S by finishing time in ascending order
2. Take the first element in S, (s_1, f_1), and determine which other elements overlap with this interval
3. Using greedy, determine time t within (s_1, f_1) s.t. all overlapping tournaments will be visited at t
4. Remove all intervals that overlap with (s_1, f_1) (including (s_1, f_1)) from the set
5. Resort the set
6. Repeat until the S is empty

Algorithm 2: Tournament Greedy Function

```
TOURNAMENT-GREEDY(S):
    ResultArray[n]      // array of best times to go to the tournament
    while S is not empty do
        SortByFinishTime(S)
        t = S_1
        i = 0
        while i ≠ length(S) do
            if s_i ≤ f_1 then
                if s_i > t then
                    t = s_i
                DeleteFromSet(s_i, f_i)
            else
                i++
            AppendToResultArray(t)
        end
    end
```

Existence of Subproblems. After t is added to *ResultArray* and all overlapping intervals (let's say k intervals) are deleted, the new set will be smaller, as $n < n - k$.

Proof of Correctness. Our goal is to minimize of times visited while visiting all siblings. Let R be the result array of the greedy solution.

Contradiction: Assume that R is not the optimal solution and instead there exists the optimal solution A where $length(A) < length(ResultArray)$.

$$\begin{array}{lllllll} A_1 & A_2 & A_3 & ... & & & \\ R_1 & R_2 & R_3 & ... & R_i & ... & R_k \end{array}$$

In order to satisfy the constraint of visiting all siblings, A_1 must lie in the interval (s_1, f_1) (when S is sorted in order by finishing time) as well as all other intervals that overlap with (s_1, f_1). A_2 lies in the interval (s_m, f_m) where (s_m, f_m) is the minimum of $S - [(s_i, f_i) \supset A_1]$, where $i \in [0, n]$. Similarly, A_3 lies in the interval of the $min(S - [(s_i, f_i) \supset [A_1, A_2]])$.

However, the greedy algorithm follows the same logic and also incorporates $R_1 \in (s_1, f_1), R_2 \in min(S - [(s_i, f_i) \supset R_1)$, etc. Since the greedy algorithm will always make the most locally optimal choice, it contradicts that there exists a more optimal algorithm and that $length(A) < length(result)$.

Proof of Optimal Substructure. When the optimal solution is chosen for the earliest finish time, this will maximize the size of the number of k games removed from the set - in other words, this minimizes $n - k$ in the following subproblem. By choosing the optimal solution for each subproblem, we also minimize the number of potential subproblems, thereby minimizing our total number of trips to the tournament.

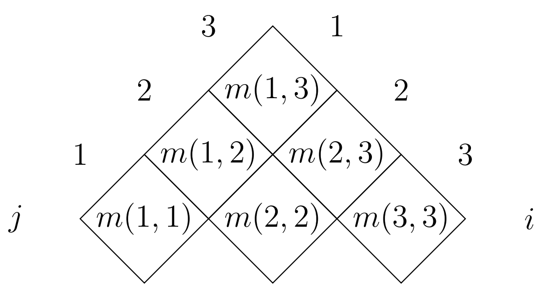
Question 5

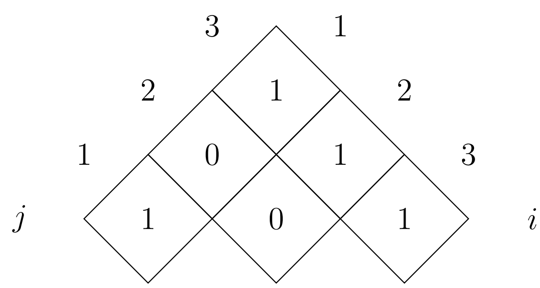
The problem to solve is as follows: We want to travel from points x_1 to x_n on a trail and we can choose to stop at any point x_i, $i \in [2, n-1]$ on the way. Every stop, x_i has an associated cost of stopping, b_i, and every two stops x_i and x_j have an associated cost of traveling without stopping, c_{ij}.

Approach. The larger problem from $[1, n]$ can be divided into small sub-problems where we are currently located at a stop x_i and must decide whether to continue without stopping to x_j. The goal is to choose the minimum cost between c_{ij} (not stopping) and the sum of b_i (stopping) with the cost of the rest of the journey. This can be expressed by the recursion below, let $m(i, j)$ represent the minimum cost of traveling between x_i and x_j.

$$m(i,j) = \begin{cases} 0, & \text{for i=j} \\ \min\{c_{ij},\ b_i + m(i+1,j)\}, & \text{otherwise} \end{cases}$$

A recursive algorithm from the recurrence above would solve the problem however in order to reduce the number of computations we would store computed $m(i, j)$ values in an upper half table and store a 1 in location (i, j) to indicate a stop at i on the way to j. An example of these tables is shown below for $n = 3$, note that the path from $(1, 3)$ to $(3, 3)$ shows the stops that need to be made, which would be returned by the algorithm.





Arguments of correctness. Assume by contradiction that there is a cost smaller than the one found by the recursion, there are two cases:

1. *The algorithm stopped at some x_i when it should have continued to an x_j.* This is impossible since the algorithm always picks the minimum of c_{ij} and $b_i + m(i+1, j)$, so if there was a smaller c_{ij} it would have been chosen.

2. *The algorithm did not stop at an x_i when it should have.* By the same logic as above, if the algorithm always picks the minimum, if there was a b_i which resulted in a lower cost it would have been chosen.

Time complexity analysis. Since we are storing intermediate results in the table, the run-time of the algorithm is the time to fill the table, which is $O(n^2)$.