SCL 2020 Contest #1 (Design Doc)

Item Stock

Approach

I created two classes: an Item class, and an ItemManager class.

Item

- A reference to child nodes
- The item's ID
- The type of stock
- Quantity of stock
- A reference to the parent or null if its the root

ItemManager

- Root of the tree
- A list of items that are currently present in the tree

Has the following methods:

- update_stock(Item) - which takes the current node as root and updates the stock of everything in its subtree. The fixed stock is already fixed, so its stock will not have to be updated.
- addNewItem(Item) - this finds the parent stock if its dynamic, otherwise it will search for the youngest fixed stock ancestor and it will calculate that item's stock accordingly, which will then update the stock of the parent.
-

For each node inserted into the tree, it would be either fixed stock or dynamic stock.

A) For the case of the dynamic stock, store a reference to the parent upon node creation, and upon creation of the item, decide the stock based on the parent quantity.

   Operation A should take O(1) time.

B) For the case of a fixed stock, do the following:
   a) Insert accordingly into the tree structure
   b) Find the youngest ancestor that has a fixed stock.
   c) Subtract the quantity available from the fixed stock ancestor
   d) Do a breadth first search by updating everything rooted in the subtree of that fixed stock ancestor.

   Operation B is upper bounded by O(N), where N is the number of nodes in the item tree.

Solution: O(n) best case, O(n) average case, O(n^2) worst case, where n is the number of nodes in the tree | O(n) space

If there are n dynamic stocks and 0 fixed stocks: O(n) time since insertion would take constant time.

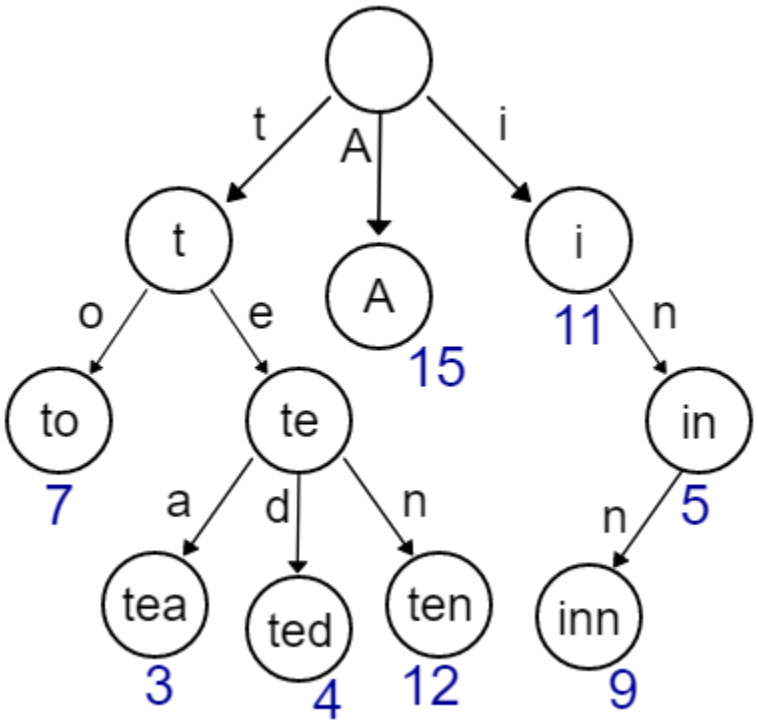
If there are 0 dynamic stocks and n fixed stocks: the distance to each subtree would be O(1), and fewer nodes rooted at the subtree of each stock.

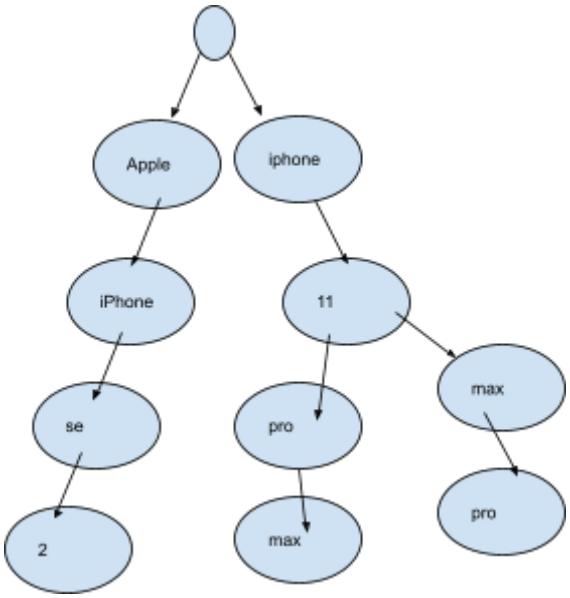
Worst case occurs when n-1 nodes **are children to the first node**, in which the insertion of 1 fixed stock node would take O(n) time because this requires the updating of quantity of every node in the tree. Doing this n times would then result in an O(n^2) time complexity.

Search Engine

Approach

Most likely optimal solution for this question would be using a trie (or k-ary search tree).





The image above shows an example of a trie - it has a root, and each edge at depth i represents the substring of the string that ends at index i. In the context of the question, we should see something like this for the second test case (see image on the right

This would be how we can efficiently represent 3 queries:
⇒ Apple iPhone se 2
⇒ iphone 11 max pro
⇒ iphone 11 pro max

The implementation of this will be a little different from a standard multi string search in the sense that:

- It has to be able to search for **all possible queries that may or may not be the first node in the query** - so there has to be some reference to where all the nodes which share the same name are.

Did this using 2 classes:

TrieNode:

- Name of substring
- The list of next nodes that it points to (the successive word in its substring)
- **A set of possible sequence numbers**

Trie:

- A hash table / dictionary storing a list of all vertices key: name of substring, value: TrieNode

Has the following methods:

- add_item(): adds an item to the trie
- search_item(): searches for an item in the trie

And a bunch of helper methods which call these basic methods.

After building the trie, I process each query by:

- Starting at the root, initially taking all nodes in the set of possible queries.
- Upon entering the first query in the string, the possible queries set is reduced to those nodes which have a reference to the first query of the string.
- Then see if any of the next nodes in the set of possible queries matches the next query item.

Once the query has finished processing, out of all the nodes remaining, count the number of unique sequences that satisfy these queries.

O(E+Q) time | O(E+Q) space

Judging Servers

Key points:

- Out of S items with some cost P, pick N items (where 1 <= N <= S) such that total cost P is minimised.
- If item i is picked, P[i-1] or P[i+1] = 0 **but not both**

My Approach

Limitation:
This method does not work for a sorted array.
Could you think of a work around?

Considering the second test case:
6 servers available out of which you pick 3
Costs for servers: 1000 560 30 85 100 900

Express the problem as a 2 dimensional array:
Let *n* represent the number of servers we can buy. Let *s* represent the number of servers available. Let **LeftMinCost[i][j]** represent the **minimum cost** we have to pay if we have the **first *s* servers** and **we pick *n* servers** from the nth:

LeftMinCost

n\s	S[0]	S[0…1]	S[0…2]	S[0…3]	S[0…4]	S[0…5]
:0						
1						
2						
3						

In the end, we are interested in only the cell in green, because 6 servers and 3 choices is what we need.

LeftMinCost

n\s	S[0] = 1000	S[0…1] = 560	S[0…2] = 30	S[0…3] = 85	S[0…4] = 100	S[0…5] = 900
:0	INF	INF	INF	INF	INF	INF
1						
2						
3						

Since it's not possible to pick 0 servers, so everything in the first column is infinity.
For 1 server or 2 servers, this value will be the cost of the minimum server.

LeftMinCost (For the purpose of our analysis, let's assume that we are picking 4 servers instead of 3 too):

n\s	S[0] = 1000	S[0…1] = 560	S[0…2] = 30	S[0…3] = 85	S[0…4] = 100	S[0…5] = 900
:0	INF	INF	INF	INF	INF	INF
1	1000	560	30	30	30	30
2	INF	560	30	30	30	30
3	INF	INF				
4	INF	INF	INF			

Notice that for every odd number of servers that we pick from, we will have to pick 1 extra server: The minimum cost at any point when the number of servers are odd would be dependent on the following:

- The cost of picking the server at index j and add it to the optimal solution when we pick 2 servers (**A**)
- The cost of not picking the server at index j and use the optimal solution when we pick 2 servers from a smaller subset of servers **(B)**

If the number of servers we need to pick is even, we don't need to pick any additional server. Then the minimum cost at any point would be dependent on the following:

- If there is an optimal cost using the solution at **A** when the number of servers was odd
- If there is an optimal cost using the solution at **B** when the number of servers was odd
- If there is an optimal cost if we selected a smaller subset of servers and a smaller choice of servers when the number of servers was odd.

Or in other words:

$$\text{if num of servers is odd:}$$
$$\text{LeftMinCost}[i][j] = \min(\text{LeftMinCost[i-1][j-1]} + \text{price[j]}, \text{LeftMinCost[i][j-1]})$$
$$\text{if num of servers is even:}$$
$$\text{LeftMinCost[i][j]} = \min(\text{LeftMinCost[i-1][j]}, \text{LeftMinCost[i-1][j-1]}, \text{LeftMinCost[i][j-1]})$$

See matrix on the next page.

n\s	S[0] = 1000	S[0…1] = 560	S[0…2] = 30	S[0…3] = 85	S[0…4] = 100	S[0…5] = 900
:0	INF	INF	INF	INF	INF	INF
1	1000	560	30	30	30	30
2	INF	560	30	30	30	30
3	INF	INF	560 + 30 = 590	30+85 = 115	115	115
4	INF	INF	INF	min (590,115, INF) = 115	115	115

LeftMinCost[4][5] or the case if we pick 4 servers from 5 available ones would be 115.

Time and space complexity of this is O(NS), where N is the number of servers to pick and S is the number of servers available, which brings the upper bound to O(S^2) and a space c

We can then compute the minimum cost from the right hand side by computing the minimum cost from the right. Because of the INF values in red (from the diagram), if the minimum cost servers were at the first few indices of the array, we would have neglected them.

So do the same process but compute the costs from the right hand side:
Let *n* represent the number of servers we can buy. Let *s* represent the number of servers available, starting from considering **only the last server,** then the **last 2** servers, and so on until we consider the full-list of servers. Let **LeftMinCost[i][j]** represent the **minimum cost** we have to pay if we have the **first *s* servers** and **we pick *n* servers** from the nth:
rightMinCost

S[0…5] = 1000	S[1…5] = 560	S[2…5] = 30	S[3…5] = 85	S[4…5]] = 100	S[5] = 900	***n\s***
INF	INF	INF	INF	INF	INF	:0
30	30	30	85	100	900	1
30	30	30	85	100	INF	2
min(1000 + 30, 115) =115	min(30 + 560, 115) = 115	min(30 + 85, 185) = 115	min(100 + 85, INF) = 185	INF	INF	3
115	115	min(115, 185, INF) = 115	INF	INF	INF	4

if i is odd :
$\text{rightMinCost[i][j]} = min(\text{rightMinCost[i-1][j+1]} + \text{prices[j]}, \text{rightMinCost[i][j+1]})$
if i is even:
$\text{rightMinCost[i][j]} = \min(\text{rightMinCost[i-1][j+1]}, \text{rightMinCost[i][j+1]}, \text{rightMinCost[i-1][j]})$

Answer will then be the minimum of the rightMinCost, and the leftMinCost.

Overall time and space complexity: O(NS) time | O(NS) space

Lucky Winner

Goal:

- Given a grid of N rows and 3 columns, where each item in matrix has a value:
- 1 token can get the value of 2 adjacent items on the grid (either horizontally or vertically)
- Maximise the value you can obtain with K tokens, and non-overlapping sections of the grid.

My Approach

Point to note: the **greedy solution to this question <u>does not work</u>** I have learnt this the hard way only after obtaining the answer to my solution:
If you look closely at the input sample:

100	-9	-1
-1	3	2
-9	2	3
2	5	1
3	3	4

Notice that the optimal solution of 3 tokens would be to take these 3 solutions as seen above to give a total score of 113

In my solution, while implementing the Greedy Algorithm for this solution, my implementation took this instead:

100	-9	-1
-1	3	2
-9	2	3
2	5	1
3	3	4

which would give the overall score of 112.

My Approach

We can convert the input matrix here into a graph where index i,j is connected to the indices (i+1, j), (i, j+1), (i-1, j) and (i, j-1), depending on its position in the matrix.

1. Traverse the matrix and compute the max possible sum for an index i,j by considering the neighbours. This takes O(n) time

2. Place all the sums as well as references to the nodes into a max heap.

3. Repeat the following steps K times:
    a. Pop the max valued item in the heap until we reach an index (i,j) that has not been visited before.

    b. Update the max_value for all the cells i,j that depends on the value of i,j for its maximum value.

    c. Update the max_values for all the cells i,j that depends on the value of the other cell that this chosen cell depends on.

    d. Restore the heap property when the maximum values are changed.

4. Return the sum of the max values after K iterations.

I implemented the following using 1 class as well:
PriceItem:
- row: integer
- col: integer
- value: integer
- other: PriceItem —> reference to node that the max value of this node depends on
- include: list(PriceItem) → reference to nodes that its value is counted into the max values of the adjacent cells
- taken: boolean → which signals whether this PriceItem has already been consumed
- max_val: integer → stores the maximum value of the node
- priority: used for max heap → stores the **negative** of max_val property.

Overloaded
- __lt__ method for max heap

Runtime: O(KN) time, and K is order N, so O(N^2) time, O(n) space

---