|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times executed** | | **Total Cost** |
| For (i = 1; i < size of course vector; i++) | 1 | N | | N |
| Create a variable j which is equal to i initially | 1 | 1 | | 1 |
| While j is less than courses.at(j – 1) | 1 | n-1 | | n-1 |
| Swap the values into the correct order | 1 | n-1 | | n-1 |
| For each course in the courses vector | 1 | N | | N |
| Print the course information in sorted order | 1 | N | | N |
| Create an integer variable to store the total number of prerequisites | 1 | 1 | | 1 |
| For each course in the vector courses | 1 | N | | N |
| Increment total prerequisites by the number of prerequisites in the current course | 1 | N | | N |
| For each course in the courses vector | 1 | N | | N |
| If the given course number matches | 1 | N | | N |
| Print the course information | 1 | 1 | | 1 |
| **Total Cost** | | | 9n-2 | |
| **Runtime** | | | O(9n-2) | |

The vector data structure or linked list data structure would be ideal for this situation, as they provide you with the ability to insert data, remove data, as well as to sort it. This means that it'd be easy to add or remove courses depending on if a student's course list changes, or if new course offerings become available. On the other hand, based on the possible complexity of these functions, the vector and linked list data structures may be too slow depending on the number of courses offered. The more courses that are added, the slower the data structure will be. Based on the Big O calculation for the runtime complexity of this data structure, it would in fact run slowly when compared to the other data structures analyzed below. So, while it is an ideal data structure in the sense that it makes sense for inserting, removing, and sorting data, it may not be fast enough.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times executed** | | **Total Cost** |
| int totalPrerequisites = 0; | 1 | 1 | | 1 |
| For each course in courses | 1 | N | | N |
| totalPrerequisites += size of course.prerequisites vector | 1 | N | | N |
| return totalPrerequisites; | 1 | 1 | | 1 |
| Hash the courseNumber and store it in a variable representing the bucket | 1 | 1 | | 1 |
| Set the number of buckets probed to zero | 1 | 1 | | 1 |
| While (bucket is not empty since start && the number of buckets probed is less than N | 1 | n-1 | | n-1 |
| if (the bucket is not empty && the bucket's courseNumber matches search query) | 1 | N | | N |
| Print the course number, the course title, and course prerequisites | 1 | 1 | | 1 |
| Increment the index of the bucket | 1 | 1 | | 1 |
| Increment the number of buckets probed | 1 | 1 | | 1 |
| Return null when nothing is found | 1 | 1 | | 1 |
| **Total Cost** | | | 4n-1 | |
| **Runtime** | | | O(4n-1) | |

The hash table data structure can be really useful when you are mapping keys to values, encrypting data, or storing common data in one "bucket". In this case, it sort of makes sense to use a hash table, as you could group courses together based on their prerequisites. However, hash tables can be complex to implement, and in the case of a course list, does not make much sense to me overall. The Big O calculation for the hash table functions written in pseudocode is given in the above table, and it indicates that it would possibly run slightly faster than the vector or linked list data structures. This is because less searching will have to take place, as if the courses are organized into buckets based on prerequisites, the course being searched for can be hashed into a prerequisite course or a general number representing each bucket, and found almost immediately.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times executed** | | **Total Cost** |
| If there is not a node anymore, return and exit the function | 1 | 1 | | 1 |
| Recursively execute this function on the node's left subnode | 1 | N | | N |
| Add the left node's course prerequisites to the prerequisites vector | 1 | 1 | | 1 |
| Add the right node's course prerequisites to the prerequisites vector | 1 | 1 | | 1 |
| Recursively execute this function on the node's right subnode | 1 | N | | N |
| Start with the root node of the tree | 1 | 1 | | 1 |
| While there is still a node that exists in the tree | 1 | N | | N |
| If the current node's course's course number matches the given courseNumber | 1 | 1 | | 1 |
| Print the matching node's course's title, course number, and prerequisites | 1 | 1 | | 1 |
| Return and exit the function | 1 | 1 | | 1 |
| If the given courseNumber is less than the current node's course's course number | 1 | 1 | | 1 |
| Move on to the left subnode and loop again | 1 | 1 | | 1 |
| else | 1 | 1 | | 1 |
| Move on to the right subnode and loop again | 1 | 1 | | 1 |
| Display that the course was not found if no node has the specified course | 1 | 1 | | 1 |
| **Total Cost** | | | 3n | |
| **Runtime** | | | O(3n) | |

The binary tree data structure is great, beccause you could also represent each node in terms of a prerequisite. For example, a prerequisite that all courses have in common could be represented as the root node, and then branched out from there based on which prerequisites each course has in common. Searching a binary tree has a low runtime complexity because the nodes are already organized, else it would not be a binary search tree. Searching through a binary search tree significantly reduces the number of nodes visited because each node is ordered, and the correct path can be followed to the desired node because of this. The Big O calculated in the above table indicates that the runtime is the lowest of the three data structures compared, meaning it could possibly be the fastest algorithm for the job.

Because binary search trees have the lowest runtime complexity, and it will likely be easy to partition the course list up into nodes, I will use a binary search tree as the data structure for the coding portion of the project.