Tom Fenyak

August 25, 2018

Questions (15 points):

1. For each of the following problems, propose the “action/operation” you would use to determine the time complexity (short answers are fine here).
   1. Find *x* in a list of names

* For finding x in a list of names you would use comparison to compare x to the n elements within the list.
  1. Transpose an n x n matrix M
* For transposing a matrix we would use addition to step through each row/column to produce the correct output.
  1. Multiply two n x n matrices
* We use comparison for multiplying matrices because we need to compare the columns of matrix one to the number of rows in matrix two.
  1. Sort a list of numbers
* We use comparison for sorting a list of numbers as we are comparing each element to be sorted to the other elements in the list.
  1. Traverse (to display e.g.,) a binary tree (represented as a linked structure where each node contains pointers to its left and right children)
* I would think that we would use addition to navigate through and place each node in the binary tree.

1. Research on line and in the textbook what an *in-place* algorithm. Use then your own words to explain the definition of an in-place algorithm and provide an example of an algorithm that is in-place.

* An in-place algorithm is an algorithm that is designed to produce the same memory space in the output that was given for the input. This allows for a minimal amount of RAM used for the algorithm. Some examples of an in-place algorithm would be an insertion sort or selection sort.

Exercise 1 (25 points)

This exercise explores the concept of best, average, and worst cases for the time complexity.

Consider the following problem: ***insertInSortedLis****t*

**Input**: a number *a* and a **sorted** linked list *L* of numbers

**Output**: a sorted list including *a* and *L*.

* Propose in pseudocode an algorithm for *insertInSortedList*.

insertInSortedList

insert a to the end of list L

[a] = L.length + 1

for i = L.length to 0

a = last element in list

if [a] <= [i]

//swap L[i] and L[a]

buffer = L[a]

L[i] = L[a]

L[a] = buffer

* Propose an action to analyze the time complexity of *insertInSortedList*
* For this algorithm we will use comparison as we are comparing the element **a** to each of the other elements in list **L** and swapping them until **a**  is at the correct index.
* Is the time complexity constant for any problem instance? If not, determine and justify the time complexity for the best, average, and worst case.
* No the time complexity is not constant.
* Best case scenario the time complexity will be constant at **n**
* Average case scenario is **n**^2/2
* Worst case scenario is **n**^2
* How do the time complexities for the best case, average case, and worst case grow?

The time complexities grow as **n**^2

* Analyze the space complexity. Is your algorithm in-place?

The space complexity will grow as **n** and this algorithm is in-place because it does not require us to copy and make a new array and our input and output are the same size.

Exercise 2 (60 points) Analyze transpose algorithm

Consider the problem to compute the transpose AT of an n x n matrix A.

Consider the pseudocode to compute the transpose of an n x n matrix A

transposeMatrix(A)

1: for i = 1 to n

2: for j = 1 to i – 1

3: // swap A[i][j] and

4: buffer = A[i][j]

5: A[i][j] = A[j][i]

6: A[j][i] = buffer

All the questions in this exercise are related to the **transposeMatrix(A)** algorithm. The objective of this exercise is to explore whether the time complexity will change if we count different “actions”

1. Comparison Action

In this case, we count the number of *comparisons* performed by the “for loops” (Lines 1 and 2). Answer the following questions to determine the total number of comparisons performed by the algorithm *transposeMatrix(A)*.

* 1. How many comparisons are performed by “for loop” in Line 1?

n + 1

* 1. Let us call ti the number of comparisons performed by “for loop” in Line 2 for a given value of i. Fill in this table:

|  |  |
| --- | --- |
| i | ti |
| 1 | n – 1 |
| 2 | n - 2 |
| 3 | n - 3 |
| i | n – i |
| n-1 | 1 |

* 1. Express the total number of comparisons performed by the “for loop” in Line 2 during the execution of *transposeMatrix(A).*

**n(n +1) / 2**

* 1. Express the function fc(n) that represents the overall total number of comparisons performed by the “for loops” in Lines 1 and 2 during the execution of *transposeMatrix(A).*
  2. The function fc(n) grows like which function?

**N^2**

1. Assignment Action

In this case, we count the number of *assignments* performed by Lines 4-6. Answer the following questions to determine the total number of assignments performed by the algorithm *transposeMatrix(A).*

* 1. Let us call ai the number of assignments performed by Lines 4-6 for a given value of i. Fill in this table:

|  |  |
| --- | --- |
| i | ai |
| 1 | 3(n – 1) |
| 2 | 3(n – 2) |
| 3 | 3(n – 3) |
| i | 3(n – i) |
| n-1 | 3(n-1) – (n-1) or 3 |

* 1. Express the function fa(n) that represents the overall total number of assignments performed by Lines 4-6 during the execution of *transposeMatrix(A).*
     + n(n – 1) / 2
  2. The function fa(n) grows like which function?

n^2

1. **Swap** Action

In this case, we count the number of *swaps* A[i][j]🡨🡪A[j][i]. **One** swap is performed by the three assignments in Lines 4-6. Answer the following questions to determine the total number of swaps performed by the algorithm *transposeMatrix(A)*.

* 1. Let us call si the number of swaps for a given value of i. Fill in this table:

|  |  |
| --- | --- |
| i | si |
| 1 | 3(n – 1) |
| 2 | 3(n – 2) |
| 3 | 3(n – 3) |
| i | 3(n – i) |
| n-1 | 3(n – n – 1) or 3 |

* 1. Express the function fs(n) that represents the overall total number of swaps during the execution of *transposeMatrix(A).*

n(n – 1) / 2

* 1. The function fs(n) grows like which function?

n ^ 2

1. **Compare** the functions fc(n), fa(n), and fs(n) and discuss their “growth”.

All of their growths are n ^2 which is the worst case scenario for this algorithm. That makes sense because to get that number we simply multiply the number of rows (n) by the number of columns (n).

1. **Space complexity**: answer the following questions. Is this algorithm is in-place? What is its space complexity? How does its space complexity “grow”?

Yes this algorithm is in-place. Its space complexity is n^2 +3 because the matrix takes up **n** x **n space** and there are three variables which take up 3 space units. The time complexity grows by n^2