

# Assignment 4

Course: COMPSCI 2C03

Date: 10/31/2025

Author: Sulagna Nandi

MacID: nandis

Student Number: 400575118

Professor: Dr. Vincent Maccio

## Problem 1.1

This algorithm correctly joins the sets of pairs  $F$  and  $S$ . There are no assumptions about whether  $F$  and  $S$  are lexicographically sorted. So, using the student ID, name, and courses example (where  $F$  contains pairs (student ID, name) and  $S$  contains pairs (student ID, course)), for each student ID in  $F$ , the entirety of  $S$  will be searched for the same student ID. For each entry in  $S$  where a student ID match is found, the triple (student ID, name, course) will be added to the list of triples named result. There is guaranteed to be no duplicates in result because, for each student ID from  $F$ ,  $S$  is fully traversed through exactly once, which means each course that the student with that particular student ID is in will be added within its corresponding triple exactly once (for that particular student). There is guaranteed to be no triples missing because each student ID is looked at and, for each student ID, the entirety of all the courses that every single student is taking (list  $S$ ) is traversed. This algorithm would have a time complexity of  $\Theta(|F| \times |S|)$  where  $|F|$  and  $|S|$  are the lengths of lists  $F$  and  $S$ , respectively. This is because, for each element in  $F$ , we are traversing the entirety of list  $S$ .

## Problem 1.2

---

**Algorithm 1** BetterComputeJoin(F, S)

---

```
1: Pre: F and S are lexicographically sorted.
2: result := empty list of triples
3: i := 0                                     ▷ index in F
4: j := 0                                     ▷ index in S
5: while  $i < |F|$  and  $j < |S|$  do
6:   if  $F[i].id = S[j].id$  then
7:     Add ( $F[i].id, F[i].name, S[j].course$ ) to result
8:      $j := j + 1$ 
9:   else if  $F[i].id < S[j].id$  then
10:     $i := i + 1$ 
11:   else
12:     $j := j + 1$ 
13:   end if
14: end while
15: return result
```

---

## Problem 1.3

Given the precondition that F and S are lexicographically sorted, this algorithm will correctly generate and return a list of triples that are the elements F and S joined based on the first entry in each element of F and S. Let us consider the student ID, name, and course example, where F contains pairs (student ID, name) and S contains pairs (student ID, course). Since F and S are both sorted primarily based on student ID, for each  $studentID_i$  in F, we know that the pairs  $(studentID_i, course_1)$ ,  $(studentID_i, course_2)$ , etc., in S are all grouped together. Therefore, for each  $studentID_i$ , that section of S can linearly be added to result in the form  $(studentID_i, name_i, course_j)$  until the next student ID is reached in S. When the next student ID is reached,

the same process will be repeated for  $i+1$ . This will have a worst-case time complexity of  $O(|F| + |S| + |result|)$  since we are linearly traversing through  $F$  and, for each student ID in  $F$ , we are only looking at the corresponding section of  $S$  (we never look at one pair in  $S$  more than once) and, for each match, we are adding one triple to result. If you count each of those three as a step, there will be at most  $|F| + |S| + |result|$  steps.

## Problem 2.1

The Master Theorem can be used, since QuickSort is a divide and conquer algorithm.

$$\begin{aligned} T(1) &= \Theta(1), \\ T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right) + O(n). \end{aligned}$$

Using the Master Theorem:

$$a = 2, \quad b = 2, \quad f(n) = O(n).$$

Then  $n^{\log_b a} = n$ , so  $f(n) = \Theta(n^{\log_b a})$ , therefore

$$T(n) = \Theta(n \log n) = O(n \log_2 n).$$

## Problem 2.2

Worst-case recurrence:  $T(1) = \Theta(1)$ ,

$$T(n) = T(n - 1) + \Theta(n),$$

since the worst pivot would move only one element to its final place and partitioning costs  $\Theta(n)$ .

Solve by unraveling each  $T(i)$  term:

$$\begin{aligned} T(n) &= \Theta(n) + \Theta(n - 1) + \cdots + \Theta(1) \\ &= \Theta\left(\sum_{i=1}^n i\right) = \Theta(n^2). \end{aligned}$$

So, this means that the worst-case time complexity to find the median ( $k \approx n/2$ ) is  $\Theta(n^2)$ .

## Problem 2.3

Let us assume  $n = |L|$ .

Worst case recurrence:  $T(1) = \Theta(1), \quad T(n) = T(0.7n) + cn \quad (c > 0).$

Claim:  $\exists C > 0, n_0$  s.t.  $T(n) \leq Cn$  for all  $n \geq n_0$ .

Proof by induction on  $n \geq n_0$ . Base: choose  $C$  large so  $T(n) \leq Cn$  for  $n \leq n_0$ .

Inductive step: assume  $T(m) \leq Cm$  for all  $m < n$ .

$$T(n) = T(0.7n) + cn \leq C(0.7n) + cn = n(0.7C + c).$$

Choose  $C$  so that  $0.7C + c \leq C \iff C \geq \frac{c}{0.3}$ .

With this as our  $C$  (and the base choice) the inequality holds, so  $T(n) \leq Cn$ .

$\therefore T(n) = O(n)$ . (Since our partitioning costs  $\Omega(n)$  as well,  $T(n) = \Theta(n)$ .)

## Problem 2.4

The existence of the ThreeTen method means that we can always choose a pivot that splits the list into reasonably balanced parts (30/70). This guarantees that QuickSort never becomes extremely unbalanced.

In other words, each recursive step reduces the problem size by a fixed fraction, so the recursion depth grows only logarithmically.

$$T(n) = T(0.3n) + T(0.7n) + O(n)$$

Solving this recurrence will give you

$$T(n) = O(n \log n).$$

This means that with the ThreeTen function, QuickSort can achieve a guaranteed worst-case time complexity of  $O(n \log n)$ , rather than just an average  $O(n \log n)$  like in randomized QuickSort.