
DSC 40B - Quiz 03

Name:

Student ID:

About the quizzes:

- Quizzes in DSC 40B are *optional* and graded pass/fail.
- A score of 70% or higher earns a “pass” and 1.5 credits toward your final grade.
- If you don’t pass, no credits are earned, but it doesn’t hurt your grade.
- You have 30 minutes to complete the quiz.
- Unfortunately, we can’t answer clarifying questions during the quiz. If you think a question has a bug or is unclear, please let us know in a private post on Campuswire after the quiz, and we’ll take it into account when grading.

Problem 1.

Consider the implementation of `binary_search` from lecture, reproduced below:

```
import math
def binary_search(arr, t, start, stop):
    """
    Searches arr[start:stop] for t.
    Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

Suppose `arr` is the array below, and that `binary_search(arr, 30, 0, len(arr))` is called.

[2, 5, 9, 14, 20, 27, 33, 41, 48, 56, 63, 70]

How many calls to `binary_search` are made in total, including the first call?

☐ 2 ☐ 3 ☒ 4 ☐ 5 ☐ 12

Solution: 4.

The calls are:

1. `start = 0, stop = 12`: the middle index is 6, and `arr[6]` is 33, which is greater than 30, so we search the left half.
2. `start = 0, stop = 6`: the middle index is 3, and `arr[3]` is 14, which is less than 30, so we search the right half.
3. `start = 4, stop = 6`: the middle index is 5, and `arr[5]` is 27, which is less than 30, so we search the right half.
4. `start = 6, stop = 6`: this is empty, so the function returns `None`.

This problem was similar to Problem #112 in the Quiz 03 practice problems.

Problem 2.

Consider the version of `binary_search` below. It is the same as the code from lecture, except that a `print` statement has been added which prints the value of `start` at the beginning of each call.

```
import math
def binary_search(arr, t, start, stop):
    print(start) # <-- added print statement
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
```

```

else:
    return binary_search(arr, t, middle+1, stop)

```

Suppose `binary_search` is called on a sorted array with `start = 0` and `stop = len(arr)`. True or False: the numbers printed are never decreasing. That is, if x is printed before y , then $x \leq y$.

- ☒ True
- ☐ False

Solution: True.

Each recursive call either keeps `start` the same (when it searches the left half, `arr[start:middle]`) or sets it to `middle + 1` (when it searches the right half, `arr[middle+1:stop]`). Since `middle` is at least `start`, `middle + 1` is larger than `start`. So `start` never decreases from one call to the next.

This problem was similar to Problem #111 in the Quiz 03 practice problems.

Problem 3.

The function below claims to count the number of times that `t` appears in `arr[start:stop]`.

```

import math
def count(arr, t, start, stop):
    if stop - start == 0:
        return 0
    if stop - start == 1:
        if arr[start] == t:
            return 1
        else:
            return 0
    middle = math.floor((start + stop) / 2)
    return (
        count(arr, t, start, middle)
        +
        count(arr, t, middle, stop)
    )

```

Suppose it is called with `start = 0` and `stop = len(arr)`. Will it always return the correct count?

- ☒ Yes: it will always return the correct count.
- ☐ No: it may recurse infinitely.
- ☐ No: it may try to access the array at an invalid index.
- ☐ No: it will run without error, but the count it returns may be wrong.

Solution: Yes: it will always return the correct count.

Let's check the three things from lecture. First, the base cases are correct: an empty range contains no copies of `t`, and a range of size one contains one copy exactly when its only element is `t`.

Second, the recursive calls are on smaller problems. We only reach them when `stop - start` is at least 2, so `middle` is strictly between `start` and `stop`, and both `arr[start:middle]` and `arr[middle:stop]` are smaller than `arr[start:stop]` (and non-empty). So the recursion always reaches a base case.

Third, assuming the recursive calls work, the whole function works. The two recursive calls count the copies of `t` in

`arr[start:middle]` and `arr[middle:stop]`,

which together contain every element of `arr[start:stop]` exactly once. So their sum is the number of copies of `t` in `arr[start:stop]`.

This problem was similar to Problem #118 in the Quiz 03 practice problems.

Problem 4.

Which recurrence describes the time taken by `boo` below on an input of size n , when $n > 1$?

```
import math
def boo(n):
    if n <= 1:
        return
    boo(n / 3)
    for j in range(math.floor(n)):
        print(j)
    boo(n / 3)
```

- ☐ $T(n) = T(n/3) + \Theta(n)$
- ☒ $T(n) = 2T(n/3) + \Theta(n)$
- ☐ $T(n) = 2T(n/3) + \Theta(1)$
- ☐ $T(n) = T(2n/3) + \Theta(n)$

Solution: $T(n) = 2T(n/3) + \Theta(n)$.

`boo` makes two recursive calls, each on a problem of size $n/3$, which take $2T(n/3)$ time in total. The loop between them runs $\lfloor n \rfloor$ times, so it takes $\Theta(n)$ time.

This problem was similar to Problem #131 in the Quiz 03 practice problems.

Problem 5.

What is the solution of the recurrence below?

$$T(n) = \begin{cases} 3T(n/3) + \Theta(n), & n > 1 \\ \Theta(1), & n = 1 \end{cases}$$

- ☐ $\Theta(\log n)$ ☐ $\Theta(n)$ ☒ $\Theta(n \log n)$ ☐ $\Theta(n^2)$ ☐ $\Theta(n^3)$

Solution: $\Theta(n \log n)$.

This is solved just like mergesort's recurrence, $T(n) = 2T(n/2) + \Theta(n)$. Replace $\Theta(n)$ with n and $\Theta(1)$ with 1, and assume n is a power of 3. Unrolling,

$$\begin{aligned} T(n) &= 3T(n/3) + n \\ &= 9T(n/9) + 2n \\ &= 27T(n/27) + 3n, \end{aligned}$$

so after k steps, $T(n) = 3^k T(n/3^k) + kn$. We reach the base case when $n/3^k = 1$, that is, when $k = \log_3 n$. Plugging this in gives

$$T(n) = n \cdot T(1) + n \log_3 n = n + n \log_3 n = \Theta(n \log n).$$

This problem was similar to Problem #130 in the Quiz 03 practice problems.

Problem 6.

Suppose you are sorting an array using the in-place `selection_sort` from lecture. After some number of iterations of its outer loop, the array looks like this:

[1, 3, 4, 4, 6, 9, 7, 8, 10]

What is the **largest** number of iterations of the outer loop that could have run so far?

☐ 3 ☐ 4 ☒ 5 ☐ 6 ☐ 8

Solution: 5.

In lecture, we saw two loop invariants for selection sort: after the α th iteration, the first α elements are sorted, *and* each of the first α elements is $\leq$ each of the remaining elements. Together, they say that the first α elements are the α smallest elements, in sorted order.

The 5 smallest elements are 1, 3, 4, 4, 6, and they are the first 5 elements, in order, so 5 iterations is possible. 6 is not: the first 6 elements are 1, 3, 4, 4, 6, 9, but 9 is larger than 7, which comes after it.

This problem was similar to Problem #141 in the Quiz 03 practice problems.

Problem 7.

In lecture, we modified `selection_sort` so that it finds the k th order statistic (the k th smallest element) of an array. The code is below, along with the `find_minimum` function that it uses.

```
def select_k(arr, k):
    """Find kth order statistic."""
    n = len(arr)
    if n <= 1:
        return
    for barrier_ix in range(k):
        # find index of min in arr[start:]
        min_ix = find_minimum(arr, start=barrier_ix)
        # swap
        arr[barrier_ix], arr[min_ix] = (
            arr[min_ix], arr[barrier_ix]
        )
    return arr[k-1]

def find_minimum(arr, start):
    """Finds index of minimum. Assumes non-empty."""
    n = len(arr)
    min_value = arr[start]
    min_ix = start
    for i in range(start + 1, n):
        if arr[i] < min_value:
            min_value = arr[i]
            min_ix = i
    return min_ix
```

Let n be the length of `arr`, and suppose that k can be any integer from 1 to n .

a) What is the **best case** time complexity of `select_k`?

☐ $\Theta(1)$ ☐ $\Theta(\log n)$ ☒ $\Theta(n)$ ☐ $\Theta(n \log n)$ ☐ $\Theta(n^2)$

b) What is the **worst case** time complexity of `select_k`?

- ☐ $\Theta(1)$ ☐ $\Theta(\log n)$ ☐ $\Theta(n)$ ☐ $\Theta(n \log n)$ ☒ $\Theta(n^2)$

Solution: Best case: $\Theta(n)$. Worst case: $\Theta(n^2)$.

The time taken doesn't depend on the order of the elements in `arr`: `find_minimum` always loops over all of `arr[start:]`, whether the array is sorted or not. What changes from one input to another is `k`, which is part of the input. The outer loop runs `k` times, and on the iteration with `barrier_ix = i`, `find_minimum` takes $\Theta(n - i)$ time.

Best case. The fewest iterations happen when $k = 1$ (finding the minimum). Then the loop runs once, and `find_minimum` loops over the whole array, taking $\Theta(n)$ time.

Worst case. The most iterations happen when $k = n$ (finding the maximum). Then the loop runs n times, and the total time is proportional to

$$n + (n - 1) + (n - 2) + \cdots + 1 = \frac{n(n + 1)}{2} = \Theta(n^2).$$

Finding the median ($k = \lceil n/2 \rceil$) is also $\Theta(n^2)$. In general, `select_k` takes $\Theta(kn)$ time.

This problem was covered in Lecture 07, where it was an exercise.

Problem 8.

Consider the `mergesort` function from lecture, shown below. Remember that `merge` takes $\Theta(n)$ time on arrays with n elements in total.

```
import math
def mergesort(arr):
    """Sort array in-place."""
    if len(arr) > 1:
        middle = math.floor(len(arr) / 2)
        left = arr[:middle]
        right = arr[middle:]
        mergesort(left)
        mergesort(right)
        merge(left, right, arr)
```

True or False: if the input array is already sorted, `mergesort` as written above takes $\Theta(n)$ time.

- ☐ True
☒ False

Solution: False.

`mergesort` does the same work no matter how the input is arranged: it always splits the array in half, recursively sorts both halves, and merges them, and `merge` always loops over every element of `out`. So its time is described by $T(n) = 2T(n/2) + \Theta(n)$ on every input, including sorted ones, and its best case is $\Theta(n \log n)$, the same as its worst case.

This problem was similar to Problem #127 in the Quiz 03 practice problems.

Problem 9.

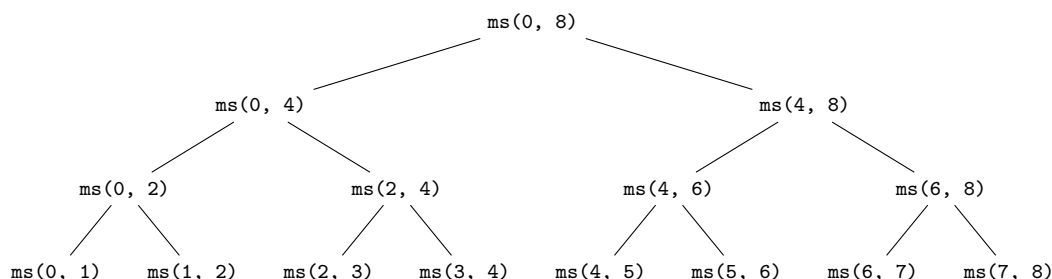
Suppose `mergesort` is called on an array of size n . In total, how many calls to `merge` are made over all calls to `mergesort`? For simplicity, you may assume that n is a power of two.

- ☐ $\Theta(\log n)$
- ☐ $\Theta(n)$
- ☐ $\Theta(n \log n)$
- ☐ $\Theta(n^2)$

Solution: $\Theta(n)$.

Each call to `mergesort` on an array with more than one element makes exactly one call to `merge`, and calls on arrays of size one make none. So we need to count the calls to `mergesort` on arrays of size two or more.

You can try this out by hand. For instance, suppose $n = 8$. Below is the tree of all of the calls to `mergesort`, where `ms(start, stop)` is the call that sorts `arr[start:stop]`:



The 8 calls at the bottom are on arrays of size one, so they don't call `merge`. Each of the other calls makes one call to `merge`: 1 on the top level, 2 on the next, and 4 on the one after that. In total, that's $1 + 2 + 4 = 7$ calls to `merge`. If you try this again with $n = 4$ or $n = 16$, you'll get 3 or 15 calls. In general, mergesort makes $n - 1$ calls to `merge`, which is $\Theta(n)$. This is true even when n isn't a power of two: mergesort still makes $n - 1 = \Theta(n)$ calls to `merge`.

Note that this is the number of *calls* to `merge`, not the total time they take. Each level of the recursion does $\Theta(n)$ work merging, and there are $\Theta(\log n)$ levels, which is where mergesort's $\Theta(n \log n)$ time comes from.

This problem was Problem #129 from the Quiz 03 practice problems.

Problem 10.

In this problem, assume that A and B are **sorted** lists, each containing n numbers.

Consider the problem of finding the **biggest** difference between a number in A and a number in B . That is, we want the largest value of $|a - b|$, where a is any number in A and b is any number in B . For example, if $A = [1, 4, 9]$ and $B = [2, 3, 12]$, the biggest difference is $|1 - 12| = 11$.

What is the best possible (worst case) time complexity of any algorithm for this problem?

Hint: the lists are sorted. Try to think of the fastest algorithm you can that takes advantage of that fact.

- ☒ $\Theta(1)$
- ☐ $\Theta(\log n)$
- ☐ $\Theta(n)$
- ☐ $\Theta(n \log n)$
- ☐ $\Theta(n^2)$

Solution: $\Theta(1)$.

Think about how we found the answer in the example. The biggest difference, $|1 - 12| = 11$, came from the smallest number in A and the largest number in B . That's no accident: to make $|a - b|$ as big as possible, we want one number to be as small as possible and the other to be as large as possible. In general, the biggest difference is either the largest number in B minus the smallest number in A (as in the example), or the largest number in A minus the smallest number in B , so we compute both and take the larger.

Because the lists are sorted, finding the smallest number in a list takes constant time (it's the first element), and so does finding the largest number (it's the last element). So there's an algorithm that takes $\Theta(1)$ time, and we can't do better than constant time, so $\Theta(1)$ is the answer.

This problem was part (b) of Problem #146 from the Quiz 03 practice problems.