Algorithms

Homework 1: 2023-12753 EunSu Yeo

1. Environment & Setting

Environment	Setting
Code Editor	Visual Studio Code
Language	Python
Version	3.11.5

2. Program description

   a. Randomized-select

To implement the Randomized selection algorithm in Lecture Note, I used the python random library. To get a simple randomized partition code, I used the code we learned in ch04. Below is the Python code implemented using the pseudo code of partition.

```python
def partition(arr: List[int], low: int, high: int) -> int:
    # partition the array around the pivot
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        # low <= j < high
        if arr[j] <= pivot:
            i += 1
            # swap arr[i] and arr[j]
            arr[i], arr[j] = arr[j], arr[i]
    # swap arr[i+1] and arr[high]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

A randomized partition code selects a random value from the input array. After that, the selected value is swapped with the last value. And by putting this fixed array back into the partition function, the last value will act as pivot.

```python
def randomized_partition(arr: List[int], low: int, high: int) -> int:
    # randomize pivot low <= pivot < high
    rand_index = random.randint(low, high)
    # swap arr[rand_index] and arr[high]
    # swap pivot to the end to work with partition(arr, low, high)
    arr[rand_index], arr[high] = arr[high], arr[rand_index]
    return partition(arr, low, high)
```

Now, I have created a randomized selection algorithm based on the randomized partition function implemented above. Based on ppt ch05 select algorithm, I created the randomized select algorithm by replacing partition to randomized partition. Below is the code I've made.

```
def randomized_select(arr: List[int], left: int, right: int, i: int) ->
int:
    """
    Selects an i-th smallest element using the randomized selection
algorithm.

    This function selects an i-th smallest element in the subarray
arr[left...right]
    using the randomized selection algorithm.

    Parameters:
        arr (List[int]): The array containing the elements.
        left (int): The starting index of the subarray.
        right (int): The ending index of the subarray.
        i (int): The 1-based order of the element to select.

    Returns:
        int: The i-th smallest element in the specified subarray.
    """
    # average-case O(n), worst-case O(n^2)
    # partition is randomized
    if left == right:
        return arr[left]
    pivot_index = randomized_partition(arr, left, right)
    k = pivot_index - left + 1
    if i == k:
        return arr[pivot_index]
    elif i < k:
        # recursive call on the left partition
        return randomized_select(arr, left, pivot_index - 1, i)
    else:
        # recursive call on the right partition
        return randomized_select(arr, pivot_index + 1, right, i - k)
```

b. Deterministic-select

Deterministic select consists of 6 parts in ppt, but the code I wrote consists of 4 parts.

Part 1: if n<=5, find answer by insertion sort and return.

```
def deterministic_select(arr: List[int], left: int, right: int, i: int) -> int:
    # worst-case O(n)
    if right - left + 1 <= 5:
        # if the subarray size is <= 5, sort and return the i-th smallest
        subarray = arr[left:right + 1]
        insertion_sort(subarray, 0, len(subarray) - 1)
        return subarray[i - 1]
```

Part 2: divide n elements into n/5 groups of 5 elements and find the median in each group by insertion sort.

```
    # Divide arr into groups of 5 elements
     medians = []
     for j in range(left, right + 1, 5):
         group_right = min(j + 4, right)
         # sort each group using insertion sort
         insertion_sort(arr, j, group_right)
         # find the median of the groups
         medians.append(arr[j + (group_right - j) // 2])
```

In Python 3.11.2, when we call the When you create an array with indexes outside the array's indexes, the parts outside the indexes are created empty. So, the remainder after the division by 5 also works out well in this code.

Part 3: find the median of medians M.

```
    # Find the median of medians
    median_of_medians = deterministic_select(medians, 0, len(medians) - 1,
(len(medians) + 1) // 2)
```

Part 4: partition n elements using M as pivot and determine which partition contains the i-th element using recursive call.

```
    # Partition arr using the median of medians as pivot
    pivot_index = partition_with_pivot(arr, left, right, median_of_medians)

    k = pivot_index - left + 1
    if i == k:
        return arr[pivot_index]
    elif i < k:
        # recursive call on the left partition
        return deterministic_select(arr, left, pivot_index - 1, i)
    else:
        # recursive call on the right partition
        return deterministic_select(arr, pivot_index + 1, right, i - k)
```

The function 'partition_with_pivot' uses the partition function we defined above.

```
def partition_with_pivot(arr: List[int], low: int, high: int, pivot: int) ->
int:
    # partition the array using a specific pivot
    pivot_index = arr.index(pivot, low, high + 1)
    # swap pivot to the end
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
    return partition(arr, low, high)
```

Below is the total code for deterministic selection algorithm.

```python
def deterministic_select(arr: List[int], left: int, right: int, i: int) ->
int:
    """
    Finds an i-th smallest element using the deterministic median-of-medians
algorithm.

    This function selects an i-th smallest element in the subarray
arr[left...right]
    using the deterministic median-of-medians selection algorithm.

    Parameters:
        arr (List[int]): The array containing the elements.
        left (int): The starting index of the subarray.
        right (int): The ending index of the subarray.
        i (int): The 1-based order of the element to select.

    Returns:
        int: The i-th smallest element in the specified subarray.
    """
    # worst-case O(n)
    if right - left + 1 <= 5:
        # if the subarray size is <= 5, sort and return the i-th smallest
        subarray = arr[left:right + 1]
        insertion_sort(subarray, 0, len(subarray) - 1)
        return subarray[i - 1]

    # Divide arr into groups of 5 elements
    medians = []
    for j in range(left, right + 1, 5):
        group_right = min(j + 4, right)
        # sort each group using insertion sort
        insertion_sort(arr, j, group_right)
        # find the median of the groups
        medians.append(arr[j + (group_right - j) // 2])

    # Find the median of medians
    median_of_medians = deterministic_select(medians, 0, len(medians) - 1,
(len(medians) + 1) // 2)

    # Partition arr using the median of medians as pivot
    pivot_index = partition_with_pivot(arr, left, right, median_of_medians)

    k = pivot_index - left + 1
    if i == k:
        return arr[pivot_index]
    elif i < k:
        # recursive call on the left partition
        return deterministic_select(arr, left, pivot_index - 1, i)
    else:
        # recursive call on the right partition
        return deterministic_select(arr, pivot_index + 1, right, i - k)
```

c. Checker

The given condition for checker program is that the time complexity must be O(n). To satisfy the given condition, I only used 1 for loop statement. By counting the elements that is smaller than the given result (which will be gotten by selection algorithm) and counting the elements that is equal to the given result, we can easily found out that the result is correct or not. If 'i' is in between the number of smaller elements and the sum of the number of smaller elements and equal elements, the result will be true.

```
def check_selection(arr: List[int], i: int, result: int) -> bool:
    """
    Verifies the correctness of the selected element.

    This function checks in linear time whether result is indeed an i-th
smallest element
    in the array.

    Parameters:
        arr (List[int]): The array containing the elements.
        i (int): The 1-based order of the element to verify.
        result (int): The element that has been selected.

    Returns:
        bool: True if result is an i-th smallest element in arr, False
otherwise.
    """
    # check if the result is the i-th smallest element
    # need to act in O(n) time
    smaller = 0
    equal = 0
    for num in arr:
        if num < result:
            smaller += 1
        elif num == result:
            equal += 1
    return smaller < i <= smaller + equal
```

3. Ratio of the constants hidden in the asymptotic time complexities

As there may be differences in running time when randomized selection algorithm works, let's use the average of 5 attempts of same input as running time.

Let's get the running time in 4 Cases (which n=100m, 200m, 300m, 400m, 500m) to calculate the hidden constants.

To make the examples for size ‘n’, I made additional code for generating examples.

Below is the code

```
import random
import os
import sys
sys.setrecursionlimit(10**6)


def read_input(filepath):
    # read data from the input file
    with open(filepath, 'r') as f:
        arr = [int(line.strip()) for line in f]
    return arr

def generate_example_input1(filepath, n):
    """Generate an example input file with random numbers."""
    # Generate a list of random numbers between 1 and n/2
    with open(filepath, 'w') as f:
        for _ in range(n):
            f.write(f"{random.randint(1, n // 2)}\n")

def generate_example_input2(filepath, n):
    """Generate an example input file with random numbers."""
    # Generate a list of random numbers between 1 and n*2
    with open(filepath, 'w') as f:
        for _ in range(n):
            f.write(f"{random.randint(1, n * 2)}\n")

def write_output_file(filepath, n, i, arr):
    """Write the output file with n, i, and array values."""
    with open(filepath, 'w') as f:
        f.write(f"{n} {i}\n")  # Write n and i separated by a space
        f.write(' '.join(map(str, arr)) + '\n')  # Write array values
separated by spaces

def main():
    # Ask the user if they want to process the new files
    proceed = input("Do you want to process the new files? (yes/no):
").strip().lower()
    if proceed != "yes":
        print("Skipping file generation. Using existing files if
available.")
        base_dir = os.path.dirname(__file__)
        input_filepath1 = os.path.join(base_dir, "example_input1.txt")
        input_filepath2 = os.path.join(base_dir, "example_input2.txt")
    else:
        # Prompt user for the value of n
        try:
            n = int(input("Enter the value of n (number of random numbers to
generate): "))
            print(f"Received input: {n}")  # 디버깅용 출력
```

```
            if n <= 0:
                print("Please enter a positive integer.")
                return
        except ValueError:
            print("Invalid input. Please enter an integer.")
            return

        # Generate example input files
        print("Generating example input files...\n")
        base_dir = os.path.dirname(__file__)

        # Filepath for generate_example_input1
        input_filepath1 = os.path.join(base_dir, "example_input1.txt")
        generate_example_input1(input_filepath1, n)
        print(f"Example input file 1 generated at: {input_filepath1}")

        # Filepath for generate_example_input2
        input_filepath2 = os.path.join(base_dir, "example_input2.txt")
        generate_example_input2(input_filepath2, n)
        print(f"Example input file 2 generated at: {input_filepath2}")

    for input_filepath in [input_filepath1, input_filepath2]:
        print(f"\nProcessing file: {input_filepath}")

        if not os.path.exists(input_filepath):
            print(f"Input file not found: {input_filepath}")
            continue

        arr = read_input(input_filepath)

        i_choice = input("Do you want to input `i` manually? (yes/no): ").strip().lower()
        if i_choice == "yes":
            try:
                i = int(input(f"Enter the value of i (1 <= i <= {len(arr)}): "))
                if i < 1 or i > len(arr):
                    print(f"Invalid input. Please enter a value between 1 and {len(arr)}.")
                    continue
            except ValueError:
                print("Invalid input. Please enter an integer.")
                continue
        else:
            i = random.randint(1, len(arr))  # Select a random i within the range of the array
            print(f"Randomly selected i: {i}")

        output_filepath1 = os.path.join(base_dir, "example_input1.in")
        write_output_file(output_filepath1, n, i, arr)
        print(f"Output file generated at: {output_filepath1}")

        arr2 = [random.randint(1, n * 2) for _ in range(n)]
        output_filepath2 = os.path.join(base_dir, "example_input2.in")
        write_output_file(output_filepath2, n, i, arr2)
        print(f"Output file generated at: {output_filepath2}")

if __name__ == "__main__":
    main()
```

By generating examples, let's check the time of each case.

Input1 means the input of having a lot of equal values, and Input2 means the input of having a small number of equal values.

Case 1: n=100000, All units are millisecond

	Input1		Input2	
Selection / Attempt	randomized	deterministic	randomized	deterministic
1	39.259195	237.794161	43.642759	239.539146
2	29.283524	232.327700	13.853073	237.668991
3	16.227722	236.250401	16.073227	227.275133
4	17.848015	230.637789	18.857718	235.623598
5	22.039652	235.921860	24.853945	236.611843
Average	24.93162	234.5864	23.45614	235.3437

Case 2: n=200000

	Input1		Input2	
Selection / Attempt	randomized	deterministic	randomized	deterministic
1	78.848600	493.520737	68.714619	497.433901
2	40.330172	474.701166	89.327335	485.786438
3	59.701920	492.808104	64.766407	484.725237
4	70.328236	479.033232	86.546659	503.512621
5	56.84638	479.157209	84.029198	498.952866
Average	61.21106	483.8441	78.67684	494.0822

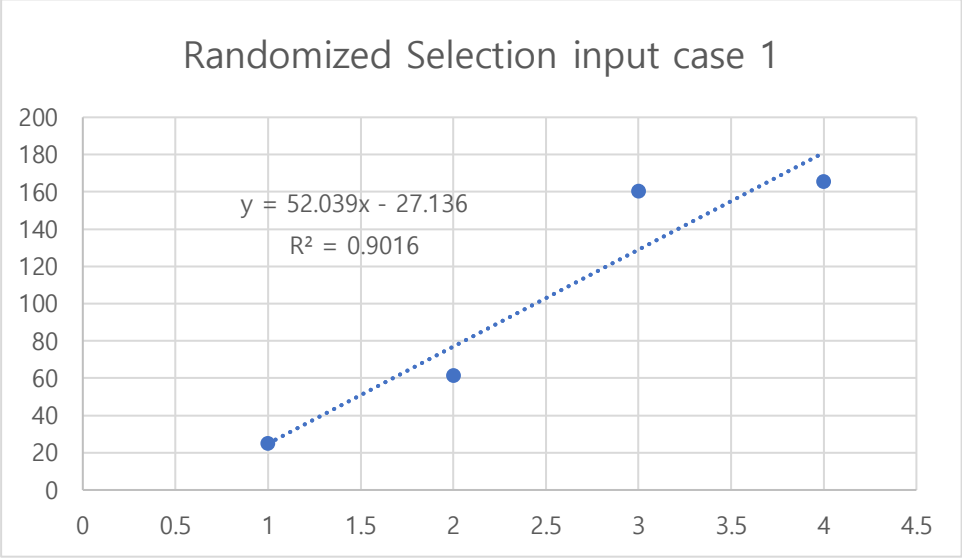
Case 3: n=300000

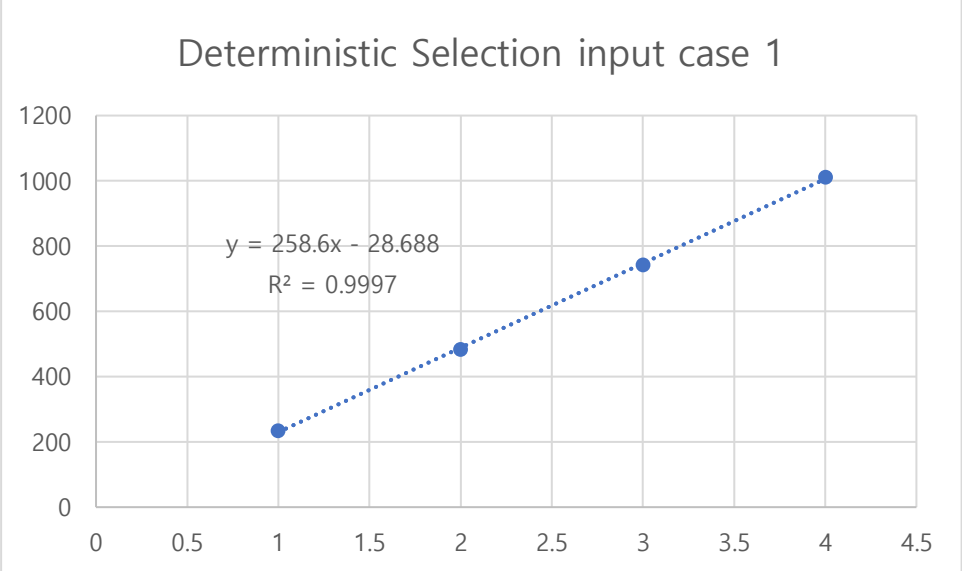
	Input1		Input2	
Selection / Attempt	randomized	deterministic	randomized	deterministic
1	117.088318	735.688210	98.075151	762.289524
2	231.843233	731.224298	93.776464	727.780342
3	159.566641	732.396841	145.112276	720.116138
4	163.301468	737.349272	143.323421	711.148739
5	129.995346	775.307894	122.066736	719.832659
Average	160.359	742.3933	120.4708	728.2335

Case 4: n=400000

	Input1		Input2	
Selection / Attempt	randomized	deterministic	randomized	deterministic
1	238.696814	983.428001	218.641281	997.479439
2	161.156893	1025.052547	228.199482	1005.732059
3	176.868677	1006.818771	205.157518	1037.059307
4	141.614437	1021.675348	206.796885	1012.826920
5	108.394384	1014.987469	182.560921	1007.955313
Average	165.3462	1010.392	208.2712	1012.211

Case1) When input 1 which has a lot of equal values.

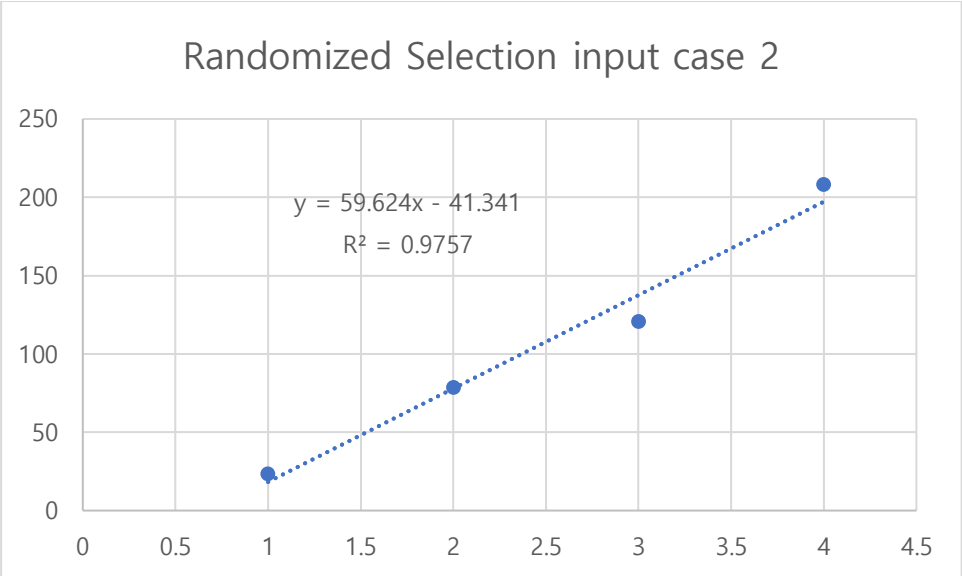


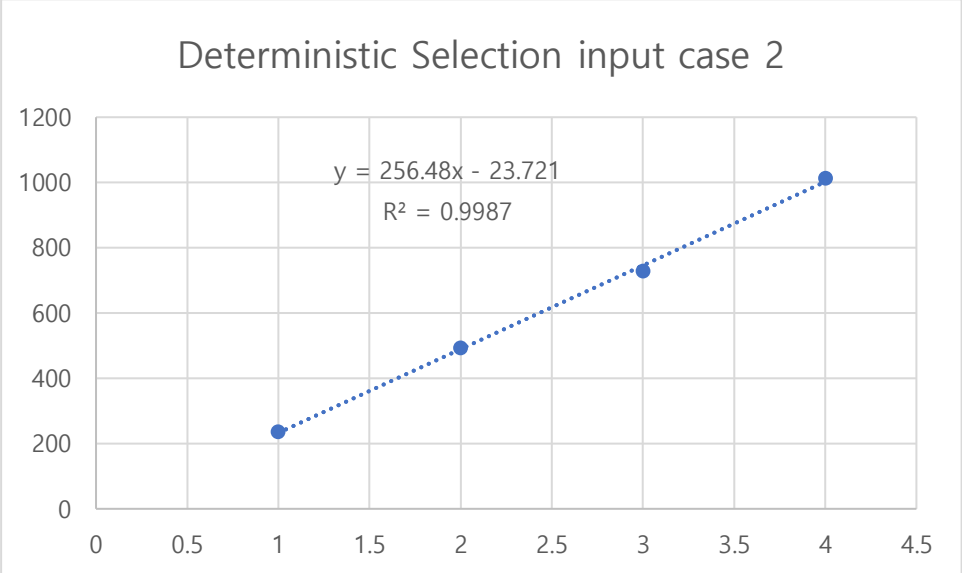


Constant hidden in time complexity is 52.039 and 258.6 each.
Let's calculate the ratio: 258.6/52.039=4.97

So, in the case where there are a lot of equal values, the ratio is 4.97
And Deterministic Selection is 4.97 times longer to get the output than the Randomized Selection in average case.

Case2) When input 2 which has a small number of equal values.





Constant hidden in time complexity is 59.624 and 256.48 each.
Let's calculate the ratio: 256.48/59.624=4.30

So, in the case where there are a small number of equal values, the ratio is 4.30

And the Deterministic Selection is 4.30 times higher than the Randomized Selection in average case.

As we got two ratios, 4.97 and 4.3. Due to these results we can know that the ratio is maybe 4.6 (which is the average of 2 ratios). Also, we can know that deterministic selection takes longer time. Because the deterministic selection is made for worst case O(n), in comparison to the average case it doesn't seem to show the difference, Moreover It takes longer time than randomized version to cover worst case.

The code and examples I used are uploaded to my GitHub.
Below is the link.
https://github.com/Ice-Moca/SNUCSE/tree/main/Algorithm(retake)/HW1