**Algorithms**

Homework 1: 2023-12753 EunSu Yeo

1. Environment & Setting

|  |  |
| --- | --- |
| **Environment** | **Setting** |
| OS | Window 11 |
| Code Editor | Visual Studio Code (Linux VM) |
| Language | Python |
| Version | 3.11.5 |

1. Program description
   1. 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.

def partition(arr, low, high):

pivot = arr[high]

i = low - 1

for j in range(low, high):

# low <= j < high

if arr[j] <= pivot:

i = 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.

def randomized\_partition(arr, low, high):

# 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 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, low, high, i):

# average-case O(n)

# worst-case O(n^2)

# partition is randomized form the ch05 ppt

# find i-th smallest element in arr[low:high]

if low == high:

return arr[low]

q = randomized\_partition(arr, low, high)

k = q - low + 1

if i == k:

return arr[q]

elif i < k:

return randomized\_select(arr, low, q - 1, i)

else:

return randomized\_select(arr, q + 1, high, i - k)

* 1. 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, i):

# worst-case O(n)

# if n <= 5, find answer by insertion sort and return

if len(arr) <= 5:

return sorted(arr)[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

# in ppt, the sorted() part is insertion sort.

medians = []

for j in range(0, len(arr), 5):

group = arr[j:j+5]

# find median in each group by insertion sort

# the empty part is filled with empty elements

# use the lower median if the group has even number of elements

medians.append(sorted(group)[(len(group) - 1) // 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.

...

median\_of\_medians = deterministic\_select(medians, (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 into left, mid, and right

# mid is the elements equal to median\_of\_medians

# mid is needed to cheek the same elements as median\_of\_medians in arr

left, mid, right = [], [], []

for x in arr:

if x < median\_of\_medians:

left.append(x)

elif x == median\_of\_medians:

mid.append(x)

else:

right.append(x)

# Determine which partition contains the i-th smallest element

# due to the sudo code in the ch05 ppt use the recursive call

if i <= len(left):

arr = left

return deterministic\_select(arr, i)

elif i <= len(left) + len(mid):

return median\_of\_medians

else:

arr = right

i -= len(left) + len(mid)

return deterministic\_select(arr, i)

Below is the total code for deterministic selection algorithm.

def deterministic\_select(arr, i):

# worst-case O(n)

# if n <= 5, find answer by insertion sort and return

if len(arr) <= 5:

return sorted(arr)[i - 1]

# Divide arr into groups of 5 elements

# in ppt, the sorted() part is insertion sort.

# But, I used sorted() for better time complexity

# https://www.wild-inter.net/publications/munro-wild-2018

medians = []

for j in range(0, len(arr), 5):

group = arr[j:j+5]

# find median in each group by insertion sort

# the empty part is filled with empty elements

# use the lower median if the group has even number of elements

medians.append(sorted(group)[(len(group) - 1) // 2])

# Find the median of medians

median\_of\_medians = deterministic\_select(medians, (len(medians) + 1)//2)

# partition arr into left, mid, and right

# mid is the elements equal to median\_of\_medians

# mid is needed to cheek the same elements as median\_of\_medians in arr

left, mid, right = [], [], []

for x in arr:

if x < median\_of\_medians:

left.append(x)

elif x == median\_of\_medians:

mid.append(x)

else:

right.append(x)

# Determine which partition contains the i-th smallest element

# due to the sudo code in the ch05 ppt use the recursive call

if i <= len(left):

arr = left

return deterministic\_select(arr, i)

elif i <= len(left) + len(mid):

return median\_of\_medians

else:

arr = right

i -= len(left) + len(mid)

return deterministic\_select(arr, i)

* 1. 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 checker(arr, i, result):

# check if the result is the i-th smallest element

# need to act in O(n) time

# problem: if there are same elements in arr

smaller = 0

equal = 0

for j in range(len(arr)):

if arr[j] < result:

smaller += 1

if arr[j] == result:

equal += 1

return smaller < i <= smaller + equal

Summary of How to run:

1. Run the Programs.py on VS code with option ‘Run Python File in Dedicated Terminal’
2. Answer the question “Do you want to process the new files? (yes/no):”

Answering ‘yes’ may make new random examples. Else use the given examples.

1. Answer the question “Do you want to input `i` manually? (yes/no):”

Answering ‘yes’ you can select the ‘i’ in the given range. Else ‘i’ will be selected randomly in the range of input size.

1. Wait for the result
2. 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 examples as running time. In the below example, I used random example of size ‘n’ and random number ‘i’ for selection algorithm.

Let’s get the running time in 3 Cases to calculate the hidden constants.

Case 1: n=100000, All units are seconds

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Rand\_Sel\_input1 | Deter\_Sel\_input1 | Rand\_Sel\_input2 | Deter\_Sel\_input2 |
| attempt1 | 0.016798 | 0.040395 | 0.014766 | 0.035832 |
| attempt2 | 0.015359 | 0.043211 | 0.028937 | 0.04693 |
| attempt3 | 0.018562 | 0.038881 | 0.012553 | 0.035054 |
| attempt4 | 0.02015 | 0.051382 | 0.020859 | 0.049119 |
| attempt5 | 0.011 | 0.040919 | 0.016868 | 0.033628 |
| Average | 0.016374 | 0.042958 | 0.018797 | 0.040113 |

Case 2: n=200000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Rand\_Sel\_input1 | Deter\_Sel\_input1 | Rand\_Sel\_input2 | Deter\_Sel\_input2 |
| attempt1 | 0.020225 | 0.074760 | 0.043211 | 0.072782 |
| attempt2 | 0.038934 | 0.075678 | 0.032589 | 0.081716 |
| attempt3 | 0.052809 | 0.093217 | 0.038791 | 0.085327 |
| attempt4 | 0.046390 | 0.075032 | 0.024999 | 0.074583 |
| attempt5 | 0.053265 | 0.076644 | 0.032855 | 0.084589 |
| Average | 0.042325 | 0.079066 | 0.034489 | 0.079799 |

Case 3: n=300000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Rand\_Sel\_input1 | Deter\_Sel\_input1 | Rand\_Sel\_input2 | Deter\_Sel\_input2 |
| attempt1 | 0.048136 | 0.082149 | 0.047316 | 0.093295 |
| attempt2 | 0.052996 | 0.094009 | 0.039533 | 0.094336 |
| attempt3 | 0.054649 | 0.091068 | 0.094398 | 0.0905 |
| attempt4 | 0.041244 | 0.089158 | 0.04523 | 0.088608 |
| attempt5 | 0.051153 | 0.089363 | 0.04493 | 0.09593 |
| Average | 0.049636 | 0.089149 | 0.054281 | 0.092534 |

The x-axis unit is 100000

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

Constant hidden in time complexity is 0.0166 and 0.0231 each.

Let’s calculate the ratio: 0.0231/0.0166=1.39

So, in the case where there are a lot of equal values, the ratio is 1.39

And Deterministic Selection is 1.39 times longer to get the output than the Randomized Selection.

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

Constant hidden in time complexity is 0.0177 and 0.0262 each.

Let’s calculate the ratio: 0.0262/0.0177=1.48

So, in the case where there are a small number of equal values, the ratio is 1.48

And the Deterministic Selection is 1.48 times higher than the Randomized Selection.

As we got the two ratios 1.39 and 1.48, we can know that deterministic selection takes longer time. Because the deterministic select is made for worst case O(n), in comparison of average case it doesn’t seem to show out the difference, Moreover It takes longer time than randomized version to cover worst case.