Contents

1 Introduction

The main objective of this assignment is to learn about dynamic programming.

2 Exercise 1: Dragon vault

For this problem, we write the following code:

```
import time
import functools
import sys

def read_vault(file_path):
    with open(file_path, 'r') as file:
        vault = [list(map(int, line.split(','))) for line in file]
    return vault

@functools.lru_cache(maxsize=50000)
def find_max_coins(vault, row, col):
    if row == 0 and col == 0:
        return vault[row][col], ''
    if row < 0 or col < 0:
        return 0, ''

    north_coins, north_path = find_max_coins(vault, row - 1, col)
    west_coins, west_path = find_max_coins(vault, row, col - 1)

    if north_coins > west_coins:
        return north_coins + vault[row][col], 'N' + north_path
    else:
        return west_coins + vault[row][col], 'W' + west_path

if __name__ == '__main__':
    vault = read_vault(sys.argv[1])
    start_time = time.perf_counter_ns()
    total_coins, path = find_max_coins(tuple(map(tuple, vault)), len(vault) -
        1, len(vault[0]) - 1)
    end_time = time.perf_counter_ns()
    print(path[::-1])
    print(total_coins)
    print(end_time - start_time)
```

We note that **@functools.lru_cache(maxsize=50000)** is extremely important here as it marks the usage of memoization of dynamic programming.

We test it's performance on 5 test sets:

- Vault of size 4 x 5 (4 rows, 5 columns)
- Vault of size 10 x 10 (10 rows, 10 columns)
- Vault of size 15 x 15 (15 rows, 15 columns)
- Vault of size 40 x 40 (40 rows, 40 columns)
- Vault of size 100 x 100 (100 rows, 100 columns)

We run the tests on our local machine, and here's the result in table:

size of vault	Without DP (ns)	With DP (ns)
N=4, M=5	39600	39800
N=10, M=10	54973300	112200
N=15, M=15	44557778200	344700
N=40, M=40	too big...	10842400
N=100, M=100	too big...	377020500

Figure 1: Runtime table for different input size

By the way, we use the line with **@cache** to toggle the option of adopting dynamic programming or not.

Note that we did not manage to have the runtime value for the last two test cases, as it took too much time to run. Also, we have not passed the testcase given vault3.txt even with dynamic programming, as it says it exceeds the maximum recursion depth:

```
Daenerys@yuyubaobao MINGW64 ~/Desktop/hwk6-23
$ python vault.py vault3.txt
Traceback (most recent call last):
  File "C:\Users\Daenerys\Desktop\hwk6-23\vault.py", line 28, in <module>
    total_coins, path = find_max_coins(tuple(map(tuple, vault)), len(vault) -
        1, len(vault[0]) - 1)
  File "C:\Users\Daenerys\Desktop\hwk6-23\vault.py", line 16, in
      find_max_coins
    north_coins, north_path = find_max_coins(vault, row - 1, col)
  File "C:\Users\Daenerys\Desktop\hwk6-23\vault.py", line 16, in
      find_max_coins
    north_coins, north_path = find_max_coins(vault, row - 1, col)
  File "C:\Users\Daenerys\Desktop\hwk6-23\vault.py", line 16, in
      find_max_coins
    north_coins, north_path = find_max_coins(vault, row - 1, col)
  [Previous line repeated 496 more times]
RecursionError: maximum recursion depth exceeded
```

To solve this, we made a reasonable guess that given better equipments, our code might still work.

To help have a better intuition of the benefits of dynamic programming, we visualized the above runtime table using the following code:

```
import matplotlib.pyplot as plt

A = [4, 10, 15, 40, 100]
B = [39600, 54973300, 44557778200]
```

```
C = [39800, 112200, 344700, 10842400, 377020500]

A_short = A[:len(B)]

plt.figure(figsize=(10, 6))
plt.plot(A_short, B, label='Without DP (ns)', color='blue')
plt.plot(A, C, label='With DP (ns)', color='red')

plt.title('Runtime with and without DP')
plt.xlabel('Value N for size NxN vault')
plt.ylabel('Runtime (ns)')
plt.legend()

plt.show()
```

And here is the visualized result:

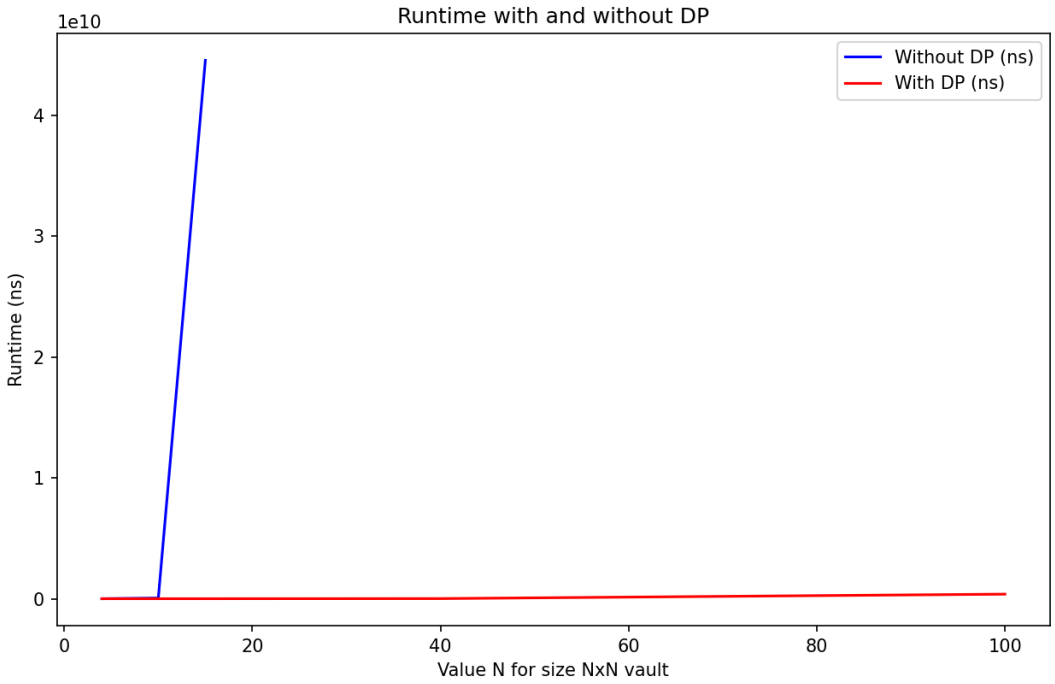

Figure 2: Visualized runtime difference to demonstrate benefits of DP

This matches our expectation, because we expect the time complexity with dynamic programming to be $O(NM)$ and the one without dynamic programming to be $O(2^{N+M})$, where N and M are the dimensions of dragon vault. Specifically, without memoization, the **find-max-coins** function calls itself recursively for each cell in the vault, so that it would need to explore all possible paths to each cell. This will result in an exponential number of calls, leading to the exponential time complexity. However, with memoization, the time cost for this function call will be $O(1)$ because all it needs to do is to look up an already stored value. For this vault, the function will be called $O(NM)$ times. Moreover, we note that the time complexity for **read_vault** will be $O(NM)$ no matter we use memoization or not, as it's only done once, and the time cost for this time is $O(NM)$.

3 Exercise 2: Matrix multiplication

For this problem, we write the following code:

```
import time
import functools
import sys

def read_matrices(file_path):
    with open(file_path, 'r') as file:
        matrices = [(line.split(',')[0], int(line.split(',')[1]), int(line.
            split(',')[2])) for line in file.readlines()]
    return matrices

@functools.lru_cache(maxsize=1000000)
def matrix_chain_order(p, i, j):
    if i == j:
        return 0
    min_ops = float('inf')
    k_best = None
    for k in range(i, j):
        ops_left = matrix_chain_order(p, i, k)
        ops_right = matrix_chain_order(p, k + 1, j)
        ops_mult = p[i][1] * p[k][2] * p[j][2]
        total_ops = ops_left + ops_right + ops_mult
        if total_ops < min_ops:
            min_ops = total_ops
            k_best = k
    return min_ops

def construct_optimal_solution(p, i, j):
    if i == j:
        return p[i][0]
    for k in range(i, j):
        if matrix_chain_order(p, i, k) + matrix_chain_order(p, k + 1, j) + p[i
            ][1] * p[k][2] * p[j][2] == matrix_chain_order(p, i, j):
            return (construct_optimal_solution(p, i, k),
                construct_optimal_solution(p, k + 1, j))
    return None


if __name__ == '__main__':
    matrices = read_matrices(sys.argv[1])
    start_time = time.perf_counter_ns()
    p = tuple((name, rows, cols) for name, rows, cols in matrices)
    min_ops = matrix_chain_order(p, 0, len(p) - 1)
    optimal_solution = construct_optimal_solution(p, 0, len(p) - 1)
    end_time = time.perf_counter_ns()
    print(optimal_solution)
    print(min_ops)
    print(end_time - start_time)
```

We note that **@functools.lru_cache(maxsize=100000)** is extremely important here as it marks the usage of memoization of dynamic programming.

We test it's performance on 13 test sets:

- 3 lines of input (multiplication of 3 matrices)

- 4 lines of input (multiplication of 4 matrices)
- 5 lines of input (multiplication of 5 matrices)
- 15 lines of input (multiplication of 15 matrices)
- 50 lines of input (multiplication of 50 matrices)
- 100 lines of input (multiplication of 100 matrices)
- 150 lines of input (multiplication of 150 matrices)
- 200 lines of input (multiplication of 200 matrices)
- 250 lines of input (multiplication of 250 matrices)
- 300 lines of input (multiplication of 300 matrices)
- 350 lines of input (multiplication of 350 matrices)
- 400 lines of input (multiplication of 400 matrices)
- 450 lines of input (multiplication of 450 matrices)

We run the tests on our local machine, and here's the result in table in Figure 3.

By the way, we use the line with **@cache** to toggle the option of adopting dynamic programming or not.

To help have a better intuition of the benefits of dynamic programming, we visualized the above runtime table using the following code:

```
import matplotlib.pyplot as plt

A = [3, 4, 5, 15, 50, 100, 150, 200, 250, 300, 350, 400, 450]
B = [136400, 136300, 166700, 405900, 332736928700]
C = [152500, 162500, 201500, 6939100, 54739200, 173062100, 451419700,
    893762300, 1676732800, 3022311600, 4895044900, 7162956100, 10880560100]

A_short = A[:len(B)]

plt.figure(figsize=(10, 6))
plt.plot(A_short, B, label='Without DP (ns)', color='blue')
plt.plot(A, C, label='With DP (ns)', color='red')

plt.title('Runtime with and without DP')
plt.xlabel('Value N for N lines of input')
plt.ylabel('Runtime (ns)')
plt.legend()

plt.show()
```

The visualized result is shown in Figure 4.

This matches our expectation, because we expect the time complexity with dynamic programming to be $O(N^3)$ and the one without dynamic programming to be $O(N * 2^N)$, where N is the number of lines of input. Specifically, with memoization, for the maximum cost, **matrix-chain-order** will be called $O(N^2)$ times (because there are N matrices,

size of input	Without DP (ns)	With DP (ns)
3	136400	152500
4	136300	162500
5	166700	201500
15	405900	6939100
50	332736928700	54739200
100	too big...	173062100
150	too big...	451419700
200	too big...	893762300
250	too big...	1676732800
300	too big...	3022311600
350	too big...	4895044900
400	too big...	7162956100
450	too big...	10880560100

Figure 3: Runtime table for different input size

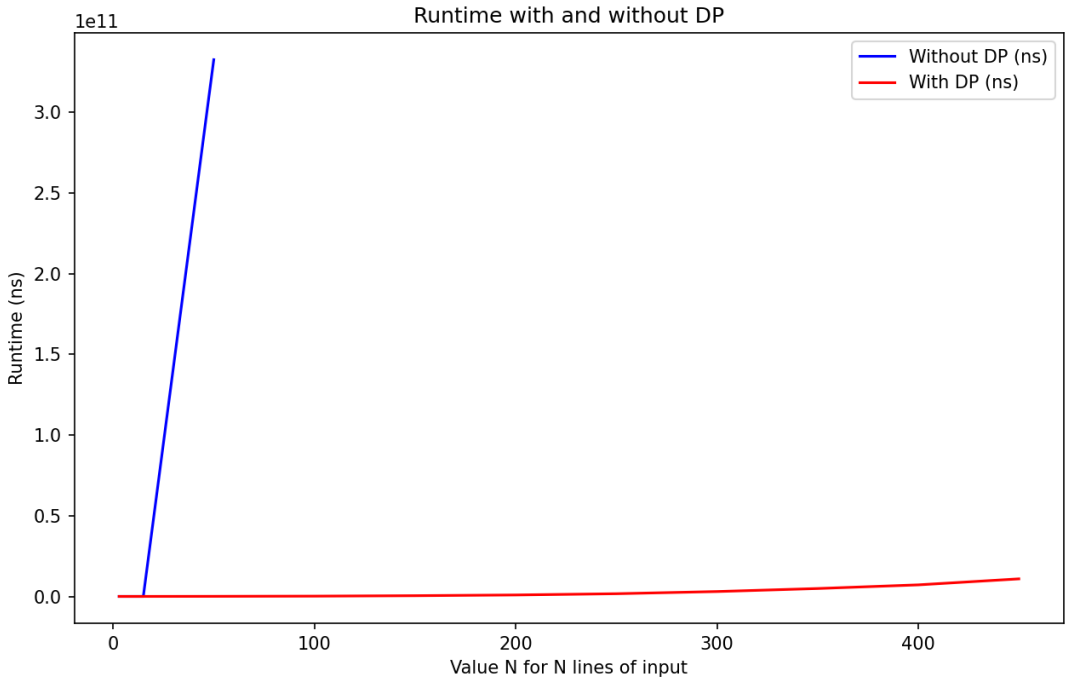

Figure 4: Visualized runtime difference to demonstrate benefits of DP

each of them executes a for loop ranging from i to j), and cost per time will be $O(N)$, leading to the $O(N^3)$ complexity of the whole code. Without memoization, as the dominant cost, **matrix-chain-order** calls itself exponentially (for each matrix, there are two options for either include it in the current multiplication or not), therefore results in $O(2^N)$ times of calls, and $O(N)$ time cost for each time. That's why we say the complexity will be $O(N * 2^N)$ in this case.

4 Exercise 3: Biased binary search tree

For this problem, we write the following code:

```
import time
import functools
import sys

class TreeNode:
    def __init__(self, val, freq, left=None, right=None):
        self.val = val
        self.freq = freq
        self.left = left
        self.right = right
        self.cost = None

    def __str__(self):
        if self.left is None and self.right is None:
            return str(self.val)
        left = str(self.left) if self.left is not None else '()'
        right = str(self.right) if self.right is not None else '()'
        return '({} {} {})'.format(self.val, left, right)

    def computeCost(self):
```

```
        if self.cost is not None:
            return self.cost
        def helper(n, depth):
            if n is None:
                return 0
            return depth * n.freq + helper(n.left, depth+1) + helper(n.right,
                depth +1)
        self.cost = helper(self, 1)
        return self.cost

def read_input(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()
    return [(int(line.split(':')[0]), int(line.split(':')[1])) for line in
        lines]

def construct_optimal_bst(keys, freqs):
    n = len(keys)
    cost = [[0 for _ in range(n)] for _ in range(n)]
    root = [[0 for _ in range(n)] for _ in range(n)]
    sum_freqs = [0] * (n + 1)
    for i in range(n):
        sum_freqs[i + 1] = sum_freqs[i] + freqs[i]
    for i in range(n):
        cost[i][i] = freqs[i]
        root[i][i] = i
    for L in range(2, n + 1):
        for i in range(n - L + 1):
            j = i + L - 1
            cost[i][j] = float('inf')
            for r in range(i, j + 1):
                c = (cost[i][r - 1] if r > i else 0) + (cost[r + 1][j] if r <
                    j else 0) + sum_freqs[j+1] - sum_freqs[i]
                if c < cost[i][j]:
                    cost[i][j] = c
                    root[i][j] = r
    @functools.lru_cache(maxsize=1000000)
    def tree_from_root(i, j):
        if i > j:
            return None
        r = root[i][j]
        return TreeNode(keys[r], freqs[r], tree_from_root(i, r - 1),
            tree_from_root(r + 1, j))
    return tree_from_root(0, n - 1)

if __name__ == '__main__':
    items = read_input(sys.argv[1])
    keys, freqs = zip(*items)
    start_time = time.perf_counter_ns()
    root = construct_optimal_bst(keys, freqs)
    end_time = time.perf_counter_ns()
    print(root)
    print(root.computeCost())
    print(end_time - start_time)
```

We note that **@functools.lru_cache(maxsize=100000)** is *supposed to be* extremely important here as it marks the usage of memoization of dynamic programming. However, for this exercise, we found a very strange thing about our design – with or without the @cache decorator, the performance does not seem to vary much. We will analyze the

possible reasons later.

We test it's performance on 10 test sets:

- 15 biased keys
- 20 biased keys
- 50 biased keys
- 100 biased keys
- 200 biased keys
- 300 biased keys
- 400 biased keys
- 500 biased keys
- 600 biased keys
- 700 biased keys

We run the tests on our local machine, and here's the result in table:

size of input	Without DP (ns)	With DP (ns)
15	235300	246500
20	504500	515000
50	5805600	5938100
100	40299500	41844600
200	322923700	320105300
300	1091474400	1102579400
400	2649767000	2627818800
500	5290265900	5310406800
600	9538436900	9563338400
700	16124002900	15776176700

Figure 5: Runtime table for different input size

By the way, we use the line with **@cache** to toggle the option of adopting dynamic programming or not. Once again, we are surprised that there are actually no significant difference whether we use the cache or not.

To help have a better intuition of this problem, we visualized the above runtime table using the following code:

```
import matplotlib.pyplot as plt

A = [15, 20, 50, 100, 200, 300, 400, 500, 600, 700]
B = [235300, 504500, 5805600, 40299500, 322923700, 1091474400, 2649767000,
    5290265900, 9538436900, 16124002900]
C = [246500, 515000, 5938100, 41844600, 320105300, 1102579400, 2627818800,
    5310406800, 9563338400, 15776176700]

plt.figure(figsize=(10, 6))
plt.plot(A, B, label='Without DP (ns)', color='blue')
plt.plot(A, C, label='With DP (ns)', color='red')

plt.title('Runtime with and without DP')
plt.xlabel('Value N for N lines of input')
plt.ylabel('Runtime (ns)')
plt.legend()

plt.show()
```

And here is the visualized result:

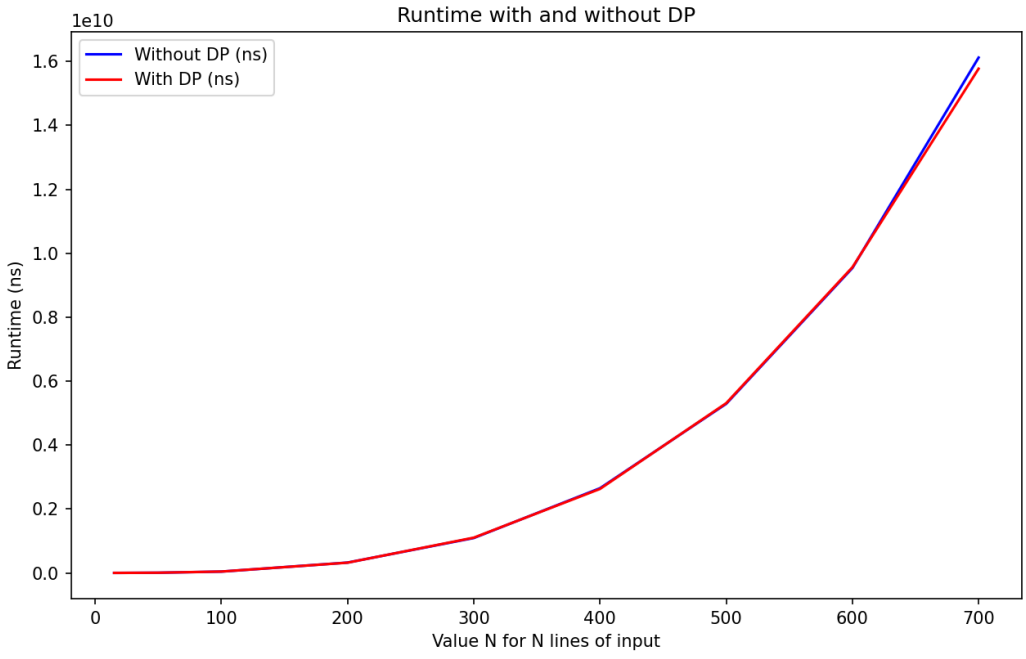

Figure 6: Visualized runtime difference to demonstrate benefits of DP

To demonstrate that our code did work well, we provide the following screenshot on a relatively large test case set, as is shown in Figure 7.

We observed this phenomenon, and after careful consideration, we can give the rea-

```
Daenerys@yuyubaobao MINGW64 ~/Desktop/hwk6-23
$ python tree.py larger_test_case_600.txt
(315 (162 (78 (43 (23 (10 (5 (2 (1 0 ()) (4 3 ())) (8 (7 6 ()) 9)) (18 (15 (11 () (13 12 14)) (17 16 ())) (21 (20 19 ()) 22))) (36 (27 (25 24 26) (32 (29 28 (31 30 ())) (35 (34 33 ())
()))) (41 (38 37 (40 39 ())) 42))) (61 (54 (48 (46 (45 44 ()) 47) (50 49 (53 (51 () 52) ()))) (58 (56 55 57) (60 59 ()))) (71 (66 (63 62 (64 () 65)) (69 (67 () 68) 70)) (75 (72 () (74
73 ())) (77 76 ))))))) (117 (98 (86 (81 (79 () 80) (83 82 (85 84 ()))) (91 (88 87 (89 () 90)) (94 (92 () 93) (97 (96 95 ()) ()))))) (107 (102 (100 99 101) (105 (103 () 104) 106)) (112 (
110 (108 () 109) 111) (115 (114 113 ()) 116)))) (139 (129 (123 (120 (119 118 ()) (122 121 ())) (126 (124 () 125) (127 () 128))) (137 (134 (132 (130 () 131) 133) (136 135 ())) 138)) (15
2 (147 (144 (140 () (143 (142 141 ()) ())) (146 145 ())) (150 (149 148 ()) 151)) (156 (153 () (155 154 ())) (161 (159 (158 157 ()) 160) ))))))) (240 (202 (179 (172 (169 (165 (164 163
()) (166 () (167 () 168))) (170 () 171)) (175 (173 () 174) (178 (176 () 177) ()))) (192 (184 (181 180 (183 182 ())) (189 (187 (185 () 186) 188) (190 () 191))) (199 (195 (193 () 194) (1
96 () (198 197 ())) (201 200 ()))) (222 (210 (206 (203 () (205 204 ())) (208 207 209)) (215 (212 211 (213 () 214)) (220 (218 (216 () 217) 219) 221))) (232 (227 (225 (224 223 ()) 226)
 (230 (228 () 229) 231)) (236 (234 233 235) (238 237 239))))) (277 (257 (248 (243 (242 241 ()) (246 (245 244 ()) 247)) (252 (249 () (250 () 251)) (254 253 (255 () 256)))) (267 (261 (25
9 258 260) (265 (263 262 264) 266)) (273 (271 (269 268 270) 272) (276 (275 274 ()) ))))) (296 (286 (280 (279 278 ()) (284 (281 () (282 () 283)) 285)) (288 287 (293 (291 (290 289 ()) 2
92) (294 () 295)))) (308 (302 (298 297 (300 299 301)) (305 (303 () 304) (306 () 307))) (312 (310 309 311) (314 313 ())))))) (450 (381 (353 (335 (328 (320 (317 316 (319 318 ())) (324 (
322 321 323) (326 325 327))) (333 (331 (329 () 330) 332) 334)) (342 (339 (336 () (337 () 338)) (340 () 341)) (347 (344 343 (346 345 ())) (350 (349 348 ()) (351 () 352)))))) (367 (362 (3
57 (354 () (356 355 ())) (360 (358 () 359) 361)) (365 (363 () 364) 366)) (376 (373 (370 (369 368 ()) (372 371 ())) (374 () 375)) (379 (378 377 ()) 380)))) (416 (400 (391 (387 (383 382
(386 (384 () 385) ())) (389 388 390)) (397 (395 (394 (392 () 393) ()) 396) (398 () 399))) (409 (405 (402 401 (404 403 ())) (407 406 408)) (413 (412 (411 410 ()) ()) (415 414 ()))))) (43
3 (423 (419 (418 417 ()) (422 (421 420 ()) ())) (428 (426 (424 () 425) 427) (430 429 (431 () 432)))) (440 (437 (435 434 436) (438 () 439)) (445 (442 441 (443 () 444)) (447 446 (448 ()
449)))))))) (529 (480 (464 (458 (454 (452 451 453) (455 () (456 () 457))) (460 459 (462 461 463))) (473 (468 (466 465 467) (471 (469 () 470) 472)) (478 (475 474 (476 () 477)) 479))) (50
0 (494 (492 (485 (483 (482 481 ()) 484) (488 (487 486 ()) (491 (490 489 ()) ())))) 493) (497 (495 () 496) (498 () 499))) (515 (507 (503 (502 501 ()) (505 504 506)) (512 (511 (508 () (51
0 509 ())) ()) (513 () 514))) (524 (521 (518 (517 516 ()) (520 519 ())) (522 () 523)) (527 (526 525 ()) 528))))) (565 (544 (535 (534 (531 530 (532 () 533)) ()) (539 (537 536 538) (542
(540 () 541) 543))) (555 (550 (547 (545 () 546) (548 () 549)) (552 551 (553 () 554))) (558 (557 556 ()) (562 (560 559 561) (563 () 564))))) (584 (575 (571 (568 (567 566 ()) (570 569 ()
)) (574 (572 () 573) ())) (577 576 (581 (578 () (579 () 580)) (582 () 583)))) (594 (588 (585 () (587 586 ())) (591 (590 589 ()) (593 592 ()))) (596 595 (598 597 599))))))))
185557
9538436900
(base)
Daenerys@yuyubaobao MINGW64 ~/Desktop/hwk6-23
$
```

Figure 7: Evidence that our code works well even if the @cache does not

sons why @cache does not really work here. Basically, we realize that dynamic programming is most useful when @cache is placed to functions that calls themselves with same parameters many times, however in our design we applied **tree-from-root** for completely different parameter pairs, making them unique recursive calls. That is to say in this design, we did not define the subproblems very well – they are all non-overlapped.

Moreover, after we computed the time complexity, we found that in our design, with the @cache decorator or not, the time complexity will always be $O(N^3)$, where N is the number of biased keys. The dominant $O(N^3)$ term comes from the three-times-nested loop inside **construct-optimal-bst**, and since **tree-from-root** is only $O(N^2)$ with or without @cache decorator (nothing stored in cache is reused due to our design, thus it's recursive call is used $O(N^2)$ times while each time cost $O(1)$ time), the time complexity remains $O(N^3)$. This aligns with our observation.

We have found out the reason why @cache decorator does not work well here – it was due to our design scheme of code this time. We will try to put up with better design where we can re-use the overlapped subproblems effectively, thus largely enhance the performance.

5 Conclusion

In this assignment, we learned about the power of dynamic programming. We learnt that memoization can make the algorithm highly efficient for even largely-sized inputs.