Solutions and Notes to the Problems in: grokking algorithms by Aditay Bhargava

John L. Weatherwax*

June 19, 2022

Chapter 1: Introduction to Algorithms

Exercise 1.1

This would take $\log_2(128) = 7$ steps at maximum.

Exercise 1.2

From the previous problem this would now take $7 + 1 = 8$ steps at maximum.

Exercise 1.3

Using binary search this would be $O(\log_2(n))$.

Exercise 1.4

The phone numbers are unsorted so no search algorithm will help here. This would take $O(n)$ time.

*wax@alum.mit.edu

Exercise 1.5

You have to read every entry so $O(n)$ time.

Exercise 1.6

There would be $O\left(\frac{n}{26}\right) = O(n)$.

Chapter 2: Selection Sort

Why is insertion into an array $O(n)$

If we need to insert at the *beginning* of an array we might have to create memory for all elements, place the item of interest at this new memory location, and then copy all $O(n)$ elements "over" before we are done.

Exercise 2.1

With many inserts it is better to use a linked list to avoid the copying required by inserting into an array.

Exercise 2.2

As the chef will be reading the first item in the queue and then "deleting it" a linked list is the preferable data structure here. The chef has no need to "randomly access" orders.

Exercise 2.3

The lookup must be done in an array (sorted) since binary search's speed is directly dependent on random access.

Exercise 2.4

Inserting a new user will take $O(n)$ time and would need to be done with every insert.

Exercise 2.5

For searching, this hybrid structure would be *slower* than an array only implementation since it does not have complete random access to its elements but *faster* than a fully linked list implementation.

For inserting, this hybrid structure would be *faster* than an array only implementation since inserting into the linked lists is fast but slower (same time) than a fully linked list implementation.

Chapter 3: Recursion

Exercise 3.1

We can tell that we were in the function `greet` with the `name` variable set to `maggie`. We then called the `greet2` function with its `name` variable also set to `maggie`.

Exercise 3.2

Your stack will get grow forever as your code will keep pushing new functions onto it. Eventually the stack for this program will run out of memory and the code will.exit with a stack overflow error.

Chapter 4: quicksort

Exercise 4.1

This would be something like

```
def recursive_sum(x):
    if len(x)==0:
        return 0
    elif len(x)==1:
        return x[0]
    else:
        return x[0] + recursive_sum(x[1:])

assert recursive_sum([1,2,3,4])==10
```

Exercise 4.2

This would be something like

```
def recursive_len(x):
    if x==[]:
        return 0
    else:
        return (1 + recursive_len(x[1:]))

assert recursive_len([1,2,3,4])==4
```

Exercise 4.3

This would be something like

```
def recursive_max(x):
    if len(x)==1:
        return x[0]
    else:
        rm = recursive_max(x[1:])
        if x[0] > rm:
            return x[0]
        else:
            return rm

assert recursive_max([1,2,3,4])==4
```

Exercise 4.4

This would be something like

```
def recursive_binary_search(x, t):
    """
    Returns the index of 't' in the array 'x' or None if 't' is not found
    """
    if len(x)==1:
        # One base case:
        #
        if x[0]==t:
            return 0
```

```
        else:
            return None
    elif len(x)==2:
        # Another base case (for ease of coding the general case below):
        #
        if x[0]==t:
            return 0
        elif x[1]==t:
            return 1
        else:
            return None
    else:
        mid_index = len(x)//2
        if t == x[mid_index]:
            return mid_index
        elif t < x[mid_index]:
            # search the lower 1/2 of the list:
            #
            return recursive_binary_search(x[:mid_index], t)
        else:
            # search the upper 1/2 of the list:
            #
            indx = recursive_binary_search(x[(mid_index+1):], t)
            if indx is not None: indx += (mid_index+1)
            return indx

# Test our code:
#
assert recursive_binary_search([1], 1)==0
assert recursive_binary_search([1], 10)==None
assert recursive_binary_search([1,2,3,4], 2)==1
assert recursive_binary_search([1,2,3,4], 3)==2
assert recursive_binary_search([1,2,3,4], 4)==3
assert recursive_binary_search([1,2,3,4], 5)==None

assert recursive_binary_search([1,2,3,4,5], 2)==1
assert recursive_binary_search([1,2,3,4,5], 3)==2
assert recursive_binary_search([1,2,3,4,5], 4)==3
assert recursive_binary_search([1,2,3,4,5], 5)==4
assert recursive_binary_search([1,2,3,4,5,9], 10)==None
```

Exercise 4.5

We have to access each element so this would be $O(n)$.

Exercise 4.6

We have to access each element so this would be $O(n)$.

Exercise 4.7

We have to access only the first element so this would be $O(1)$.

Exercise 4.8

To form this "grid" we will have to perform

$$n + (n-1) + (n-2) + \cdots + 2 + 1 = O(n^2)\,,$$

multiplications.

Chapter 5: hash tables

Exercise 5.1

This function is consistent but it gives the same value for each key and thus would produce many hash collisions.

Exercise 5.2

This function would give different values each time and would not be consistent.

Exercise 5.3

Depending on the order called (if our hash function was called before to insert or retrieve values) we might not get the same value each time for the same input x. Thus this function is not consistent.

Exercise 5.4

This would give the same value for the same input x each time and is thus consistent.

Exercise 5.5-5.7

In the `python` code `chap_5_exercise.py` we implement the hash function in **D**.

Exercise 5.5

The constant function will always have collisions. Many of these names have the same length so the length function is not a good hash function. Many names have the same initial letter so the first letter is not a good hash function. The fourth hash function maps to

```
Exercise 5.5 items hashed to the indices:
[0, 7, 3, 2]
```

and thus provides a good distribution.

Exercise 5.6

The constant function will always have collisions. Each of these names have a different length so the length function is a reasonable hash function. All of these names have the same initial letter so the first letter is not a good hash function. The fourth hash function maps to

```
Exercise 5.6 items hashed to the indices:
[2, 4, 6, 8]
```

and thus provides a good distribution.

Exercise 5.7

The constant function will always have collisions. Several of these names have the same length so the length function is not a reasonable hash function. All of these names have different initial letter so the first letter is a good hash function. The fourth hash function maps to

```
Exercise 5.7 items hashed to the indices:
[9, 5, 8, 5]
```

and thus does have a duplicate.

Chapter 6: breadth-first search

Exercise 6.1

Label the nodes from left-to-right bottom-to-top as A, B, C, and D. If we consider a breadth-first-search from node S we would first add nodes A and then D at a "distance" of one. As these are not the goal node we explore from A adding nodes C and B at a "distance" of two. As neither of these are the goal node we explore from D adding node F at a distance of two (node C has already been added). As F is the goal the shortest path has a length of two

Exercise 6.2

From "cab" we would explore "car" and "cat" (each at a distance of one). From "car" we would explore "bar" and "cat" (already added). From "cat" we would explore "bat" (at a distance of two) which is the goal state and we can stop. The shortest distance is of length two.

Exercise 6.3

I think the arrows on this graph are drawn in the wrong order or at least in a non-intuitive order.

Part (A): Not valid.

Part (B): Valid.

Part (C): Not valid. I must "wake up" before anything else.

Exercise 6.4

A valid list might be

- wake up
- brush teeth
- eat breakfast
- pack lunch

- exercise
- shower
- get dressed

Exercise 6.5

Part (A): Tree.

Part (B): Not a tree.

Part (C): Tree.

Chapter 7: Dijkstra's algorithm

Exercise 7.1

For these graphs we could execute Dijkstra's algorithm "by hand" (much like the book did for some of its examples) or implement it in python and then run our implementation on the given graphs. As the book has already presented good examples at executing Dijkstra's algorithm "by hand" I'll follow the second approach here. See the `python` code `dijkstra.py` and `chap_7_exercises.py`.

Part (A): Here I'll label the two nodes at the "bottom" of this graph from left-to-right as A and B. I'll label the two nodes at the "top" of this graph from left-to-right as C and D. With these definitions when I run the above code I find

```
Minimum cost to "finish"= 8
Path taken to get this minimum: ['start', 'C', 'B', 'finish']
```

Part (B): Here I'll label the one node at the "bottom" of this graph as A. I'll label the two nodes at the "top" of this graph from left-to-right as B and C. With these definitions when I run the above code I find

```
Minimum cost to "finish"= 60
Path taken to get this minimum: ['start', 'B', 'C', 'finish']
```

Part (C): We cannot use Dijkstra's algorithm on a graph with negative weights.

Chapter 8: greedy algorithms

Exercise 8.1

Select the largest box (by volume) that will fit in the remaining space in the truck. Repeat this process until there are no more packages that can be placed in the truck.

This will not be the optimal strategy for the same reason that the greedy approach will not be the optimal strategy for the knapsack problem. The example given in the text with item weight changed to item volume (and a max volume of 35) gives an example where the greedy algorithm is not optimal.

Exercise 8.2

This is exactly the knapsack problem. We want to maximize the sum of the point values subject to the constraint that the total length of time to do all of the items is less than our given seven days The greedy heuristic for the knapsack problem will work here. Keep picking the item with the largest point value such that when added to the items already selected does not violate the time constraint. This will not be optimal.

Exercise 8.3

Not greedy.

Exercise 8.4

Greedy. At each step you look to see if there is a path one longer that gets you to the destination node.

Exercise 8.5

Greedy. At each step you find the cheapest node and update its neighbors if a path through this cheaper node is shorter.

Exercise 8.6

Yes. This is the traveling salesman problem.

Item	1	2	3	4
Guitar (G)	1500 (G)	1500 (G)	1500 (G)	1500 (G)
Stereo (S)	1500 (G)	1500 (G)	1500 (G)	3000 (S)
Laptop (L)	1500 (G)	1500 (G)	2000 (L)	3500 (L, G)
IPhone (I)	2000 (I)	3500 (I, G)	3500 (I, G)	4000 (I, L)
MP3 (M)	2000 (I)	3500 (I, G)	4500 (M, I, G)	4500 (M, I, G)

Table 1: The dynamic programming grid for Exercise 9.1.

Exercise 8.7

This is the "largest clique" problem and is known to be NP-complete.

Exercise 8.8

This is a version of the graph coloring problem which is known to be NP-complete (for $k \geq 3$).

Chapter 9: dynamic programming

Exercise 9.1

To decide this we will add this item as another row in the dynamic programming table already presented. We will then compute the numerical values in each cell. When we do that we get the result in Table 1. The fact that the last row and last column contains the MP3 player means that we should steal it.

Exercise 9.2

We construct the dynamic programming grid for this problem in Table 2. The last row and last column indicates that we should take the Food, the Water, and the Camera on the camping trip.

Exercise 9.3

We construct the dynamic programming grid for the longest common *substring* (and not the longest common subsequence) in Table 3. The largest number in that grid is a three which

Item	1	2	3	4	5	6
Water (W)	0 ()	0 ()	10 (W)	10 (W)	10 (W)	10 (W)
Book (B)	3 (B)	3 (B)	10 (W)	11 (B, W)	11 (B, W)	11 (B, W)
Food (F)	3 (B)	9 (F)	10 (W)	12 (F, B)	19 (F, W)	20 (F, B, W)
Jacket (J)	3 (B)	9 (F)	10 (W)	14 (J, F)	19 (F, W)	20 (F, B, W)
Camera (C)	6 (C)	9 (F) or (C, J)	15 (C, F)	16 (C, W)	20 (C, J, F)	25 (C, F, W)

Table 2: The dynamic programming grid for Exercise 9.2.

	c	l	u	e	s
b	0	0	0	0	0
l	0	1	0	0	0
u	0	0	2	0	0
e	0	0	0	3	0

Table 3: The dynamic programming grid for Exercise 9.3.

is the length of the largest common substring.

Chapter 10: k-nearest neighbors

Exercise 10.1

In this case, it seems the *average* rating each user gives is somewhat different. We could subtract this *user based* average from each movies rating done by that user and then compare these normalized ratings between users.

Exercise 10.2

The k-NN method just averages the "votes" from a samples neighbors. If some neighbors (like influencers) should be allowed larger weights in the voting that can be accomplished by weighting their votes more i.e. using weighted averaging.

Exercise 10.3

With 10^6 users and $k = 5$ you are making recommendations using a very small fraction of the users. I would think this is too small to get very good recommendations and increasing the value of k would probably be a good idea. From the book, a good starting value for k would be $k = O(\sqrt{N})$ with N the number of users (data points).