# Grokking Algorithms second edition
# Exercises Solutions

Solutions by:
Amirhossein Foorjanizadeh

E-mail:
amir.frjn.84@gmail.com

GitHub Repository:
www.github.com/amir-frjn

·•·≺·∘☽.̇.̇☾∘·≻·•·

Dedicated to the immortals of
January 1404 Iran

·•·≺·∘☽.̇.̇☾∘·≻·•·

# Table of Contents

# 1. Introduction to algorithms

## 1.1

The worst case is that target doesn't exist in our list so we must search $log_2(128)$ + 1 = 8 times, checking the last one is the cause of plus 1 .

Example: suppose our list contains 1 to 128 and we are looking for 129 so steps are:

1. (1 + 128) // 2 = 64 also 129 > 64 so new range: 65 to 128
2. (65 + 128) // 2 = 96 also 129 > 64 so new range: 97 to 128
3. (97 + 128) // 2 = 112 also 129 > 112 so new range: 113 to 128
4. (113 + 128) // 2 = 120 also 129 > 120 so new range: 121 to 128
5. (121 + 128) // 2 = 124 also 129 > 124 so new range: 125 to 128
6. (125 + 128) // 2 = 126 also 129 > 126 so new range: 127 to 128
7. (127 + 128) // 2 = 127 also 129 > 127 so new range: 128 to 128
8. (128 + 128) //2 = 128 also 129 > 128 and this is the end of the list

Note: "//" means integer part of a fraction.

## 1.2

New steps are: $log_2(2 \times 128)$ + 1 = $log_2(256)$ + 1 = 9

## 1.3

In a phone book names are sorted so we can use something like binary search (instead of comparing numbers, we compare names alphabetically) thus O(log n).

## 1.4

As we don't have phone numbers sorted so we must check all items to find person's name so O(n).

## 1.5

As question says, we must check all n items in that list so O(n).

## 1.6

To find the first name that starts with "As" we can use solution in problem 1.3 so O(log n) but wait we want to read all of names that starts with that therefor by increasing amount of them our search time increases linearly so here we have O(n) and answer is O( O(log n) + O(n) ), Given that O(log n) is less than O(n), we can simplify and state this problem's time complexity as O(n).

# 2. Selection sort

## 2.1

Given more emphasis on insertions over readings, a data structure with lower insertion costs is preferred. Linked lists are chosen because their Big-O notation for insertion is O(1), which contrasts with the higher cost of reading, O(n).

## 2.2

We can use both arrays and linked lists as a queue (read appendix 1 to see how). Arrays have fixed sizes, whereas we can insert infinite data into linked lists without any limitations. So, we choose linked lists to create our queue.

## 2.3

Despite being able to access elements in arrays directly, linked lists do not allow this due to their sequential structure (requiring iteration through all previous nodes to reach a target node). Therefore, binary search is typically applied only to arrays.

## 2.4

As I mentioned earlier, arrays have a fixed size. If an array becomes full, we need to use a larger array, which is slow and inefficient. Additionally, after each insertion, the array must be sorted, and we lose O(1) for each insertion. There are two methods you can follow:

1. Insert the new item at the first available free cell and then run a sorting algorithm on the entire array.
2. Use binary search to find the correct place for the new item, shifting all subsequent values one position to the right to create space for the new item.

In both methods, the entire array is iterated once during each insertion operation, leading to more time complexity than O(1) which is O(n).

## 2.5

- This data structure does not face size limitations, offering flexibility in terms of storage capacity.
- Additionally, inserting data can be done in O(1) time by locating the appropriate cell and then updating the pointer to the last item. However, it is slightly slower than arrays or linked lists because it involves two steps (accessing the cell + accessing the linked list's

tail), whereas with arrays and linked lists, you only need to access either the cell or the linked list's tail.

- For searching an item, it takes O(n) time because you must traverse through all items included within a cell in the linked list. In contrast, arrays can use binary search to achieve O(log n), which is more efficient if applicable; otherwise, their search times are like those of linked lists at O(n). Our hybrid approach is marginally faster because it focuses on checking only a section with items that start with a particular letter, by excluding the other 25 sections and reducing the search space.

# 3. Recursion

## 3.1

The program starts by calling the GREET function with the value “MAGGIE”. Inside GREET, another function named GREET2 is called, also using the value “MAGGIE”. Currently, we are inside GREET2. Once this function finishes running, the program will return to GREET and continue with the rest of the code there.

## 3.2

Your PC has a limited amount of memory and cannot allocate more memory than its capacity. Therefore, if you have a recursive function without an endpoint (or base case), it will continue allocating memory on the call stack. Eventually, the call stack will run out of memory, and you will encounter a common runtime error: “Stack Overflow”!

In most programming languages, the default stack size ranges from 1 to 10 megabytes. Look at Appendix 2 to see a simple example.

# 4. Divide and conquer

## 4.1

```
def recursive_sum(nums_list):
    if nums_list == []:
        return 0
    return nums_list[-1] + recursive_sum(nums_list[:-1])

# To avoid an index out-of-range error, I started the index at -1. This allows
# me to add the last item to the sum of all items from the first up to the
# second-to-last.
```

## 4.2

```
def recursive_counter(items_list):
    if items_list == []:
        return 0
    return 1 + recursive_counter(items_list[:-1])
```

## 4.3

```
def max_recursive(nums_list):
    if len(nums_list) == 0:
        raise Exception("empty list was given")

    current = nums_list[0]
    if len(nums_list) == 1:
        return current

    next_max = max_recursive(nums_list[1:])
    return current if next_max < current else next_max
# If you replace ‘<’ in above line with ‘>’ the result is minimum number
```

## 4.4

```python
def binary_search(target, sorted_list, start, end):
    mid_index = (start + end) // 2
    mid_val = sorted_list[mid_index]

    if mid_val == target:
        # This is the base case if the target is found
        return mid_index
    elif mid_val < target:
        start = mid_index + 1
    else:
        end = mid_index - 1

    if start > end:
        # This is the base case if target isn't found
        return None

    # This one is recursive case
    return binary_search(target, sorted_list, start, end)
```

## 4.5

O(n): Because we visit all the items to print them.

## 4.6

O(n): As in the previous one, we must go through all the items to double them.

## 4.7

O(1): It is always done by accessing the first element; the other items do not need to be considered.

## 4.8

$O(n^2)$: Since each of the n elements is multiplied by every other element, this results in $n^2$ multiplications.

# 5. Hash tables

## 5.1

For every input, it returns 1. So, it produces the same output for the same inputs, but it is not suitable because it makes no distinction between entries. (Think of it as giving us 1 dollar as the price for every fruit.)

## 5.2

Each time we call this function, it produces a random key. Although it creates a distinction between entries, it does not follow the consistency rule because it generates a new random key for the same entry each time. (Think of it as giving us a random price every time we call it with 'apple'.)

## 5.3

This function fails the consistency requirement for hash functions because its output depends on the current state of the hash table, not just on the input x.

Imagine this array: [sth1, sth2, empty]. Here, f(sth3) returns 2. Then we move sth1 from the first index to the last index, resulting in: [empty, sth2, sth1]. Now, f(sth3) returns 0, thus violating the consistency rule.

## 5.4

It returns the same number for the same entry, but it makes only a slight distinction between different entries, which causes too many collisions. (For example, it returns 3 for 'cat', 'dog', 'bee', and so on.)

## 5.5

As you can see from the table, the best hash function is 4.

| Names / Hashes | Esther | Ben | Bob | Dan |
|---|---|---|---|---|
| Hashing 1 | 1 | 1 | 1 | 1 |
| Hashing 2 | 6 | 3 | 3 | 3 |
| Hashing 3 | E | B | B | D |
| Hashing 4[1] | 0 | 7 | 3 | 2 |

[1] See appendix 3 to see how it implements

## 5.6

As you can see from the table, the best hashes are 2 and 4. We prefer 2 because in modern programming languages, calculating length is fast (O(1)), whereas hashing with algorithm 4 is slower (O(n)).

| Hashes \ Names | A | AA | AAA | AAAA |
|---|---|---|---|---|
| Hashing 1 | 1 | 1 | 1 | 1 |
| Hashing 2 | 1 | 2 | 3 | 4 |
| Hashing 3 | A | A | A | A |
| Hashing 4 | 2 | 4 | 6 | 8 |

## 5.7

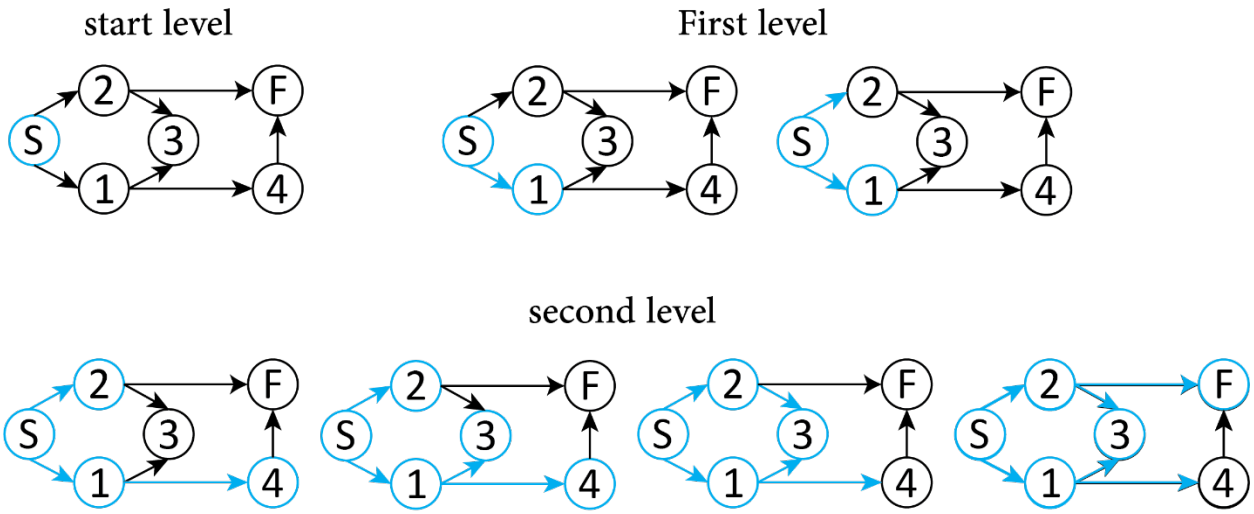
4 and 3 are suitable. 3 is done in O(1) meanwhile 4 is done in O(n). so we prefer Hashing 3

| Hashes \ Names | Maus | Fun | Home | Watchmen |
|---|---|---|---|---|
| Hashing 1 | 1 | 1 | 1 | 1 |
| Hashing 2 | 4 | 3 | 4 | 8 |
| Hashing 3 | M | F | H | W |
| Hashing 4 | 3 | 9 | 8 | 5 |

# 6. Breadth first search

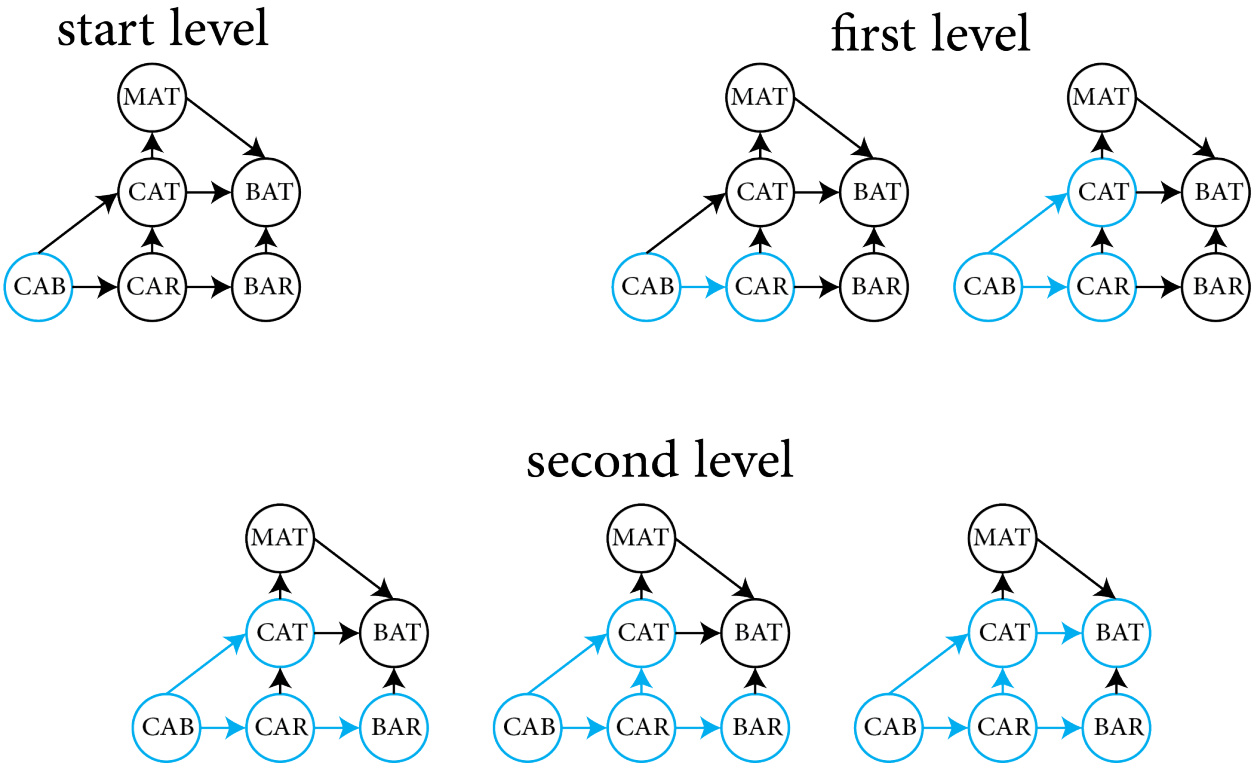
## 6.1

The length of the shortest path is 2 as shown in the following graphs:




## 6.2

The length of the shortest path is 2 as shown in the following graphs:



## 6.3

**A:** Eating breakfast before brushing your teeth is impossible because the latter depends on the former.
**B:** This one is possible.
**C:** You must be awake to do other tasks, so this is impossible.

## 6.4

A simple preorder navigation over it gives us a valid list:

Wake up
Pack lunch
Brush teeth
Eat breakfast
Exercise
Shower
Get dressed

## 6.5

A: Is a tree
B: Isn't a tree
C: Is a tree

# 7. Trees

No exercise exists :)

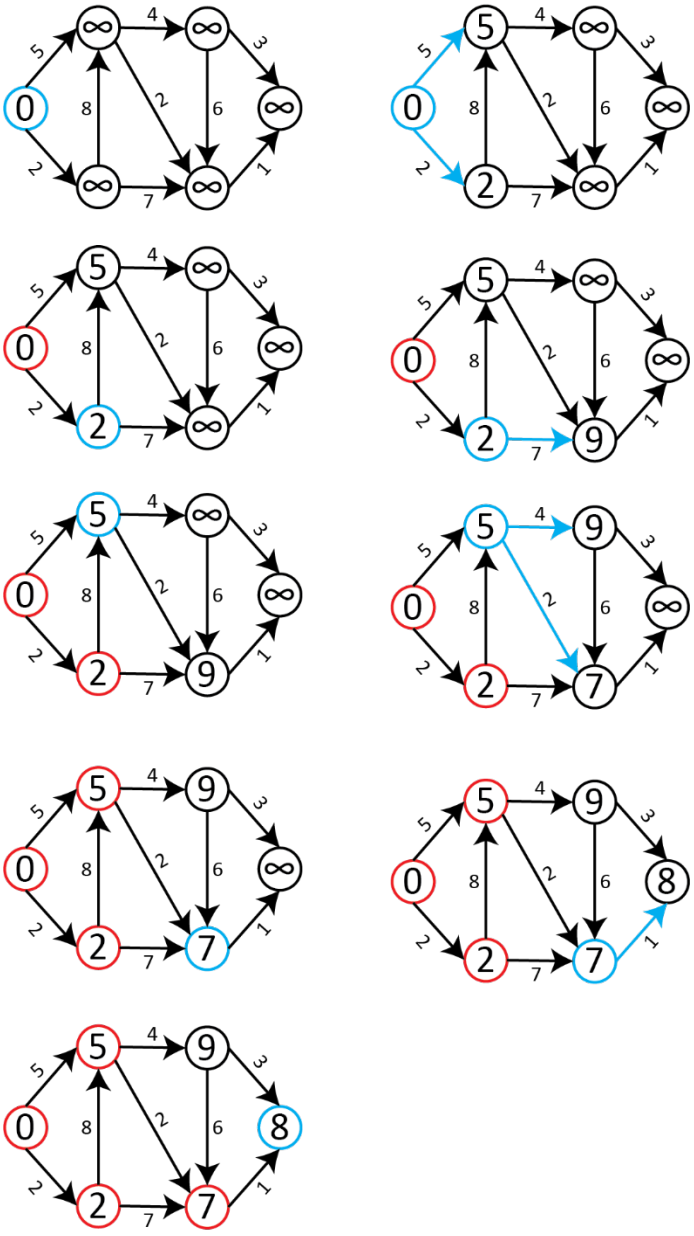
# 8. Balanced trees

Still no exercise :)

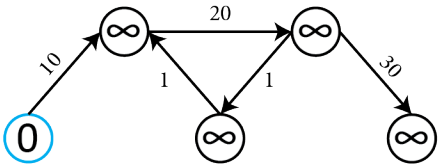
# 9. Dijkstra's algorithm

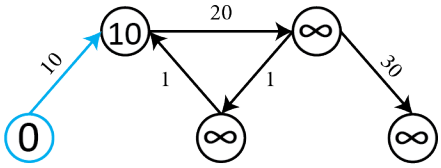
## 9.1

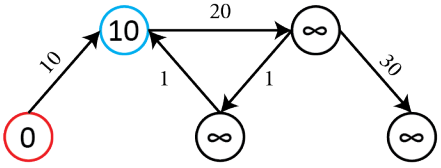
We run Dijkstra's algorithm to find the shortest path.
Blue nodes represent the current lowest-cost nodes to visit.
Red nodes are those that have already been visited.
Blue arrows indicate the proper and possible ways to travel to neighboring nodes.

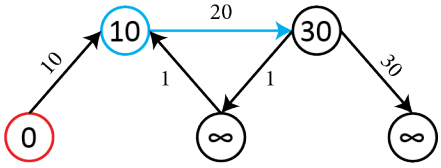


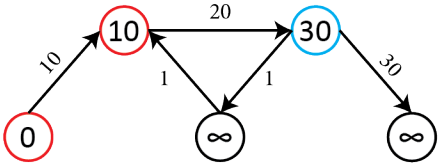
A: Thus, the shortest path length is 8.

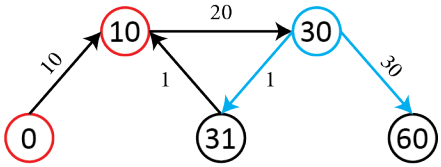


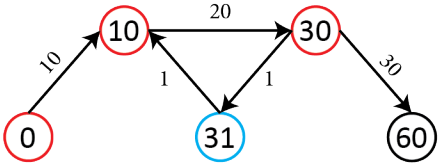



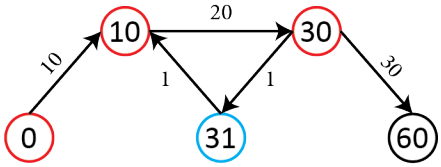



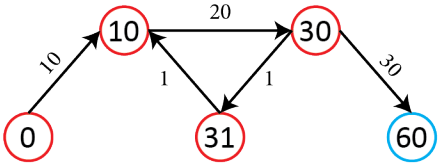



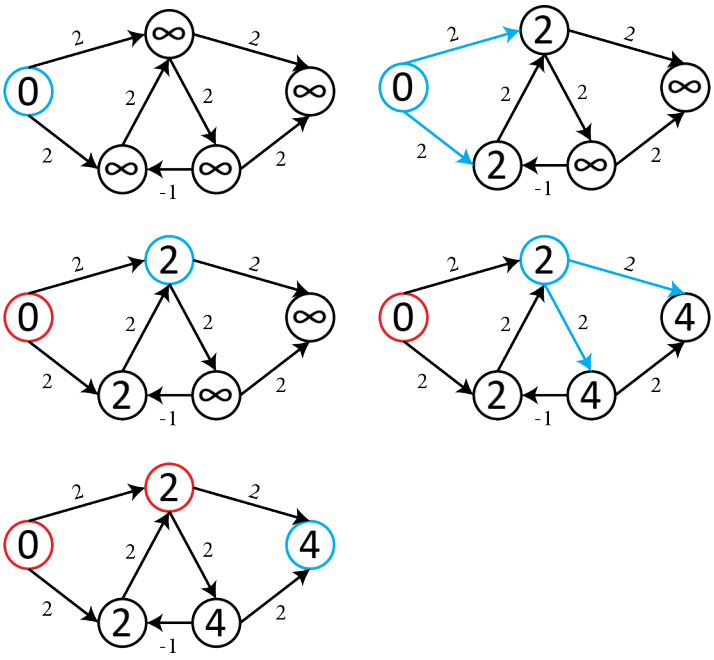




B: Thus, the shortest path <u>is 60</u>




C: This one is tricky. However, in this case, Dijkstra gives the correct answer 4; But in graphs with negative weights, we should use the Bellman–Ford[2] algorithm.

[2] See appendix 4 for Bellman-Ford implementation.

# 10. Greedy algorithms

## 10.1

In each step, the greedy algorithm collects the largest box that can still fit. However, this approach does not always yield the optimal selection. Suppose our truck has a capacity of 9, and we have boxes with weights 2, 2, 3, and 5. At the first step, the greedy algorithm picks the box of weight 5. Next, it picks the box of weight 3, since 5 + 3 = 8, which fits within the capacity. This seems reasonable, but a better combination is 2 + 2 + 5 = 9, which fills the truck completely. Two common ways to solve this kind of problem are brute-force and dynamic programming (covered in later chapter).

## 10.2

Just like in the knapsack problem, we can think of each point as the 'price' and the days spent there as the 'weight.' A greedy algorithm will always choose the option that fits within the remaining days and offers the greatest number of points. However, as usual, this approach is not optimal (just like the counterexample in the book that shows the algorithm's inefficiency). But we can improve this algorithm by introducing a **"worth factor"**: **we divide the points by the days**. Then, instead of choosing the item with the highest points, we choose the one with the greater worth factor. If several items have the same worth factor, we pick the one with the lower weight. This approach is better, but it still won't give us the optimal choice. For example, in the knapsack problem:

| item | price | weight | Worth factor |
|---|---|---|---|
| Stereo | 3000 | 30 | 3000 / 30 = 100 |
| Laptop | 2000 | 20 | 2000 / 20 = 100 |
| Guitar | 1500 | 15 | 1500 / 15 = 100 |

Here we have equal factors, so we choose the lightest items first. Thus, we pick the Guitar, then the Laptop and we find the correct answer.

As I said, sometimes it is inefficient, look at this example:

| item | price | weight | Worth factor |
|---|---|---|---|
| Stereo | 3000 | 30 | 3000 / 30 = 100 |
| Laptop | 2000 | 20 | 2000 / 20 = 100 |
| Guitar | 1500 | 15 | 1500 / 15 = 100 |
| Washer | 3392 | 32 | 3392 / 32 = 106 |

In this case, we added a washer with a worth factor of 106. Since it has the greatest worth factor, we fill our knapsack with it and can steal 3392 worth of items. However, if we had chosen the laptop and guitar instead, we would have stolen 3500.

# 11. Dynamic programming

## 11.1

Let's update our table to see the answer:

| | | | |
|---|---|---|---|
| 1500 | 1500 | 1500 | 1500 |
| 1500 | 1500 | 1500 | 3000 |
| 1500 | 1500 | 2000 | 3500 |
| 2000 | 3500 | 3500 | 4000 |
| 1000 vs 2000: 2000 | (1000+2000) vs 3500: 3500 | (1000+3500) vs 3500: 4500 | (1000+3500) vs 4000: 4500 |

Thus, we choose this one and our set becomes:
mechanical keyboard: 1 lbs. 1000$
guitar: 1 lbs. 1500$
iPhone: 1 lbs. 2000$

## 11.2

Let's create our table:

| Items \ Weights | 1 | 2 | 3 | 4 | 5 | 6 | selected |
|---|---|---|---|---|---|---|---|
| Water: 3 lb, 10 | 0 | 0 | 10 | 10 | 10 | 10 | W |
| Book: 1 lb, 3 | 3 | 3 | 3 vs 10: 10 | 3+10 vs 10: 13 | 3+10 vs 10: 13 | 3+10 vs 10: 13 | B, W |
| Food: 2 lb, 9 | 3 | 9 vs 3: 9 | 9+3 vs 10: 12 | 9+3 vs 13: 13 | 9+10 vs 13: 19 | 9+13 vs 13: 22 | F, B, W |
| Jacket: 2 lb, 5 | 3 | 5 vs 9: 9 | 5+3 vs 12: 12 | 5+9 vs 13: 14 | 5+12 vs 19: 19 | 5+13 vs 22: 22 | F, B, W |
| Camera: 1 lb, 6 | 6 | 6+3 vs 9: 9 | 6+9 vs 12: 15 | 6+12 vs 14: 18 | 6+14 vs 19: 20 | 6+19 vs 22: 25 | C, J, F, B |

So, we'll pack camera, jacket, food, book and water for our camping.

## 11.3

Let’s do it:

| clues \ blue | B | L | U | E | Common substring |
|---|---|---|---|---|---|
| C | 0 | 0 | 0 | 0 | _ |
| L | 0 | 1 | 0 | 0 | L |
| U | 0 | 0 | 2 | 0 | LU |
| E | 0 | 0 | 0 | 3 | LUE |
| S | 0 | 0 | 0 | 0 | _ |

| clues \ blue | B | L | U | E | Common subsequence |
|---|---|---|---|---|---|
| C | 0 | 0 | 0 | 0 | _ |
| L | 0 | 1 | 1 | 1 | L |
| U | 0 | 1 | 2 | 2 | LU |
| E | 0 | 1 | 2 | 3 | LUE |
| S | 0 | 1 | 2 | 3 | LUE |

Both the common substring and the subsequence are the same string "LUE".

# 12. K-nearest neighbors

## 12.1

We have many statistical ways to solve this problem. Here, I'll explain some of them (for more complex and practical solutions, see Appendix 5).

One simple approach is **binarization**. As we know, Yogi has a binary behavior (like = 5, dislike = 1). We can implement this manner for Pinky by introducing a pivot (e.g. 3) and consider all votes less than it as 1 and the rest as 5. Then, we can compare these sets.

We can improve this algorithm by incorporating the mean. This technique is known as **mean-centering.**

For each user, calculate their average rating across all movies. Then subtract that average from each of their individual ratings. After this adjustment, if a value in the set is positive, it means the user liked that movie; if it is negative, they disliked it. By comparing the like/dislike sets for both users, we can determine their similarity.

## 12.2

Just like in the previous exercise, there are many approaches to handle this problem. Some simple methods are described here.

- As the question states, influencers carry more weight. Therefore, we consider their votes with a higher weight (this is known as **weighted similarity**). In practice, we assign a coefficient A to a normal user and a coefficient B to an influencer, where A < B. This biases the KNN algorithm accordingly.
  Example:
  Let A = 1 and B = 10.
  You rated: Pulp Fiction 5, Kill Bill 4
  Normal user Bob rated: Pulp Fiction 4, Kill Bill 3 → Similarity score = 0.85 × 1 = 0.85
  Influencer Tarantino rated: Pulp Fiction 5, Kill Bill 5 → Similarity score = 0.90 × 10 = 9.0
  Even though Tarantino is mathematically slightly less like you than Bob, his influence weight makes him count 10 times more in the recommendations!

- A better and more common algorithm is the **hybrid approach**, which is especially useful for **regression**. It combines the opinions of some influencers with those of our similar neighbors in two steps:
  1. First, look for similar neighbors in the set of normal users.

2. Then, look for similar neighbors in the set of influencers.

Afterward, we combine the results from each part using two coefficients, A and B, where **A + B = 1**, to compute the final answer.

**Example:**

**Step 1** - Find your real neighbors:
Similar users to you: [Alice, Bob, Carol, David, Eva] → all action movie fans
**Step 2** - Force-add influencers:
Influencers to include: [Tarantino, Wes Anderson, Scorsese]
**Step 3** - Make prediction:
Action Movie X ratings:
Your neighbors: [5,4,5,4,5] → average = 4.6
Influencers: [5,3,5] → average = 4.3
Final weighted: (4.6 × 0.7) + (4.3 × 0.3) = 4.51

- Why be democratic when you can be a dictator? With **Influencer-First Filtering**, we decide what's good and serve it to the people. First, we seize all movies liked by influencers (or, if we're feeling inclusive, by your friends). Then, we purify the selection based on your personal taste and your friends' opinions (or flip the script and let influencers have the final say).
    1. Get top n movies rated highly by Tarantino
    2. From those n, find the ones that match your personal or your friends' tastes.
    3. Recommend those

## 12.3

It is very low amount, 5 is good for 100 users not for millions of people. The best amount is something near 50-100 parameters, but this seems still low, Netflix would:

1. First cluster users into taste groups
2. Then find k neighbors within your cluster
3. Typically use k between 50-1

# 13. Appendix

## 13.1 Queue using arrays and linked lists

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class Queue:
    def __init__(self):
        self.head = None
        self.tail = None

    def enqueue(self, value):
        new_node = Node(value)
        if self.tail is not None:
            self.tail.next = new_node
        self.tail = new_node
        if self.head is None:
            self.head = new_node

    def dequeue(self):
        if self.head is None:
            raise IndexError("Dequeue from empty queue")

        value = self.head.value
        self.head = self.head.next
        if self.head is None:
            self.tail = None
        return value
# A simple queue implemented in python using linked-lists
```

```python
class Queue:
    def __init__(self, length):
        self.start = 0
        self.end = 0
        self.len = length
        self.items = [None] * length

    def enqueue(self, new_item):
        self.items[self.end] = new_item
        self.end = (self.end + 1) % self.len

    def dequeue(self):
        dequeued = self.items[self.start]
        self.items[self.start] = None
        self.start = (self.start + 1) % self.len
        return dequeued
# A simple queue implemented in python using arrays
# Warning: Data will overwrite if the number of values exceeds the capacity
# of self.item.
```

## 13.2 Simple infinite recursion test

```
def stack_test(function_order):
    print(function_order)
    stack_test(function_order+1)

stack_test(0)
# In my pc stack_test was called 999 times then stop working and said:
# Traceback (most recent call last):#  File "/home/amirf/test.py", line 5, in #<module>
#    stack_test(0)
#  File "/home/amirf/test.py", line 3, in stack_test
#    stack_test(function_order+1)
#  File "/home/amirf/test.py", line 3, in stack_test
#    stack_test(function_order+1)
#  File "/home/amirf/test.py", line 3, in stack_test
#    stack_test(function_order+1)
#  [Previous line repeated 996 more times]
#RecursionError: maximum recursion depth exceeded
# This error says we cannot repeat self-calling more than 998 times
```

## 13.3 Simple Hash function

```
def prime_hasher(key, bound):
    key = key.lower()
    bindings = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
73, 79, 83, 89, 97]
    final_hash = 0

    for c in key:
        bind_index = ord(c) - ord('a')
        bind_code = bindings[bind_index]
        final_hash = (bind_code + final_hash) % bound

    return final_hash
# This algorithm only works for alphabetical letters
# This algorithm isn’t case sensitive (It converts uppercase letters to lowercase then
calculates it).
```

## 13.4 Simple Bellman-Ford implementation

This algorithm supports negative weights also detects negative cycles (when a cycle has a negative cost, we can traverse it many times to reduce the total cost, and this process never ends).

```
def bellman_ford(graph, source):
    # Step 1: Initialize distances
    distances = {vertex: float('inf') for vertex in graph}
    distances[source] = 0

    # Step 2: Relax edges |V| - 1 times
    for _ in range(len(graph) - 1):
        for u in graph:
            for v, weight in graph[u].items():
                if distances[u] != float('inf') and distances[u] + weight < distances[v]:
                    distances[v] = distances[u] + weight

    # Step 3: Check for negative weight cycles
    for u in graph:
        for v, weight in graph[u].items():
            if distances[u] != float('inf') and distances[u] + weight < distances[v]:
                raise ValueError("Graph contains negative weight cycle")

    return distances


# Sample graph
graph = {
    'A': {'B': -1, 'C': 4},
    'B': {'C': 3, 'D': 2, 'E': 2},
    'C': {},
    'D': {'B': 1, 'C': 5},
    'E': {'D': -3}
}
source = 'A'

shortest_distances = bellman_ford(graph, source)
print(shortest_distances)
# This code is in geeksforgeeks.org
```

**But why we don't always use Bellman-Ford:** Bellman-Ford has a time complexity of $O(V \times E)$, which on complete graphs becomes $O(V^3)$, making it extremely slow for large graphs. Dijkstra achieves $O(V^2)$ in worst case with simple implementation or $O(E \log V)$ with binary heap, running exponentially faster on most graphs. The trade-off is that we only accept Bellman-Ford's performance penalty when we absolutely need its ability to handle negative weight edges.

## 13.5 Some common dataset normalizing and comparing

- ### Z-Score standardization

**How it works:** First mean-center the ratings, then divide by the user's standard deviation[3] (how spread out their ratings are).

**Logic:** This not only accounts for different average ratings but also for different rating ranges. A user who only uses 1s and 5s has high variance; a user who uses the whole spectrum has moderate variance. Dividing by the standard deviation puts everyone on the scale of how many standard deviations from their mean is this rating?

**What it solves:** Yogi's extreme style (only 1 and 5) gets normalized to approximately -1 and +1. Pinky's accurate ratings get scaled proportionally, so a 4 might become +0.5 while a 5 becomes +1.2.

**Simple Math:** New Rating = (Old Rating − Old Average) ÷ (Old Deviation)

- ### Pearson Correlation

**How it works:** Look at pairs of ratings for movies both users have seen. Calculate how well the patterns match when one goes up relative to their average, does the other also go up relative to their average?

**Logic:** This is essentially mean centering built into a similarity metric. It doesn't care about the actual numbers, only about whether users tend to agree on which movies are better or worse than average.

**What it solves:** When Yogi rates a movie 5 (above his average) and Pinky rates it 4 (above her average), that's positive agreement. When Yogi rates 1 (below average) and Pinky rates 2 (below average), that's also agreement. The correlation captures this perfectly.

**Simple Math:** Measures how your ups and downs match my ups and downs

It might be same as binarization but they're quite different:

Binarization: Only cares if rating is above or below threshold (like/dislike only)

Pearson Correlation: Cares how much above or below each user's average (preserves intensity)

[3] To find deviation: Find the mean; For each data point; find the square of its distance to the mean; Sum the values from previous step; Divide by the number of data points; Take the square root.

So binarization treats all "likes" as equal, while Pearson knows that a 5 means more than a 4 to someone!

- ## Bayesian Personalization

**How it works:** Combine each user's personal rating pattern with the global average pattern, with more weight on personal pattern as they rate more movies.

**Logic:** New users or users with few ratings might not have reliable statistics. By "shrinking" their ratings toward the global average, you get more stable comparisons until they've rated enough movies.

**What it solves:** If Yogi has only rated 5 movies, his calculated mean of 3.8 might not be trustworthy. Bayesian approach would pull his mean toward the global average of 3.2, making comparisons more robust.

**Simple Math:** New Rating = (Your Ratings × Your Weight) + (Global Average × (1 − Your Weight))

- ## Rank-Based Transformation

**How it works:** For each user, order all their rated movies from favorite to least favorite, then replace each rating with its rank position (1st favorite, 2nd favorite, etc.).

**Logic:** This focuses entirely on preference order rather than intensity. If Yogi puts Movie A above Movie B and Pinky does the same, they agree regardless of whether Yogi gave them 5 and 1 while Pinky gave them 4 and 2.

**What it solves:** Yogi's extreme ratings and Pinky's moderate ratings become irrelevant—only the ordering matters. Their top 3 movies will be ranked 1,2,3 for both, making their similarity clear.

**Simple Math:** Replace ratings with 1st, 2nd, 3rd, etc. (based on order)