Assignment 6

Question 1

You have one supercomputer and n normal computers on which you need to run n jobs. Each job first spends some time on the supercomputer and then some time on the normal computer. A job can only start running on any of the normal computers after it has finished on the supercomputer. However, as soon as any job finishes on the supercomputer, it can immediately start on one of the free normal computers. The goal is to finish running all the jobs as soon as possible. Note that since there is only one supercomputer, you'll always have to wait for the summation of si until the present job time so that the jobs finish running on the supercomputer. However, you can optimize when you run the jobs on normal computers to try to finish running all the jobs as soon as possible.

Show that the following schedule is optimal: execute jobs after sorting them in decreasing order by ni. Note: Include any assumptions about this problem to clarify the justification. But do not allow your assumption(s) to contradict what is in the problem statement.

The optimal schedule is to sort the jobs in decreasing order of *ni*, assuming that the supercomputer takes a lot less time than the normal computers and that no job is dependent on the previous job.

Assumptions:

- The supercomputer takes a lot less time than normal computers.
- No job is dependent on the previous job.

Proof:

Let's say we have a schedule that does not sort the jobs in decreasing order of *ni*. Then, there must be two jobs, *j*1, and *j*2, such that *ni*1>*ni*2, but *si*1*si*2. This means that job *j*1 spends less time on the normal computers but more time on the supercomputer than job *j*2.

We can construct a new schedule by swapping jobs *j*1 and *j*2. In this new schedule, job *j*1 will finish running on the supercomputer sooner, and the overall time to run all the jobs will be less.

Therefore, any schedule that does not sort the jobs in decreasing order of *ni* is not optimal. The only schedule that is guaranteed to be optimal is the one that sorts the jobs in decreasing order of *ni*.

Example:

Let's say we have two jobs, *j*1 and *j*2, with the following characteristics:

- *si*1=1
- *ni*1=10
- *si*2=5
- *ni*2=2

si1 + ni1 = 11The total time to run job *j*1 is *si*1+*ni*1=11. The total time to run job *j*2 is *si*2 + *ni*2 = 7.

If we sort the jobs in decreasing order of *ni*, then job *j*2 will be run first. This means that job *j*1 will have to wait until job *j*2 is finished on the supercomputer before it can start running on the normal computers.

However, if we swap the jobs, then job *j*1 will be run first. This means that job *j*2 will be able to start running on the normal computers sooner, and the overall time to run all the jobs will be less.

Therefore, the schedule that sorts the jobs in decreasing order of *ni* is the optimal schedule.

Example with lists of times:

Let's say we have the following list of times:

- *si* = [1, 5, 3, 2, 4]
- *ni* = [10, 2, 5, 3, 4]

If we sort the jobs in decreasing order of *ni*, then we get the following schedule:

- Job 1
- Job 2
- Job 4

- Job 3
- Job 5

The total time to run all the jobs in this schedule is 1+2+4+3+4=14.

If we sort the jobs in decreasing order of *si*, then we get the following schedule:

- Job 5
- Job 4
- Job 2
- Job 3
- Job 1

The total time to run all the jobs in this schedule is 5+4+2+3+ 1 = 15.

As you can see, the schedule that sorts the jobs in decreasing order of *ni* takes less time to run all the jobs. Therefore, this is the optimal schedule.

```
def optimal_schedule(si, ni):
    jobs = [(si[i], ni[i]) for i in range(len(si))]
    jobs.sort(key=lambda x: x[1], reverse=True)  # Sort in decreasing order of ni

    total_time = 0
    for job in jobs:
        total_time = max(total_time, job[0]) + job[1]

    return total_time

# Example usage
si = [1, 5, 3, 2, 4]
ni = [10, 2, 5, 3, 4]

time = optimal_schedule(si, ni)
print("Total time for the optimal schedule:", time)
```

The time complexity of the `optimal_schedule` function in the provided code is determined by the sorting step and the subsequent iteration over the sorted list.

1. Sorting Step: The sorting step has `O(n log n)` time complexity, where `n` is the number of jobs. This is because the sorting algorithm used in Python's `list.sort()`

method (e.g., Timsort) has an average and worst-case time complexity of `O(n log n)`. Sorting the list of `n` jobs based on their `ni` values is the primary factor contributing to the time complexity.

2. Iteration Step: The subsequent iteration over the sorted list has a linear time complexity of `O(n)`. In each iteration, a constant-time operation is performed to calculate the maximum time between the current job's start time (`job [0]`) and the previous total time. This is done for each job in the sorted list.

Therefore, the overall time complexity of the `optimal_schedule` function is dominated by the sorting step, resulting in a time complexity of `O(n log n)`, where `n` is the number of jobs.

Question 2

A ski rental agency has n pairs of skis, where the height of the nth pair of skis is si. There are n skiers who wish to rent skis, where the height of the nth skier is hi Ideally, each skier should obtain a pair of skis whose height matches her or his own as closely as possible. We would like to assign skis to skiers so that the sum of the absolute differences between the heights of each skier and her or his skis is minimized. Design a greedy algorithm for the problem. Prove the correctness of your algorithm. Note: Include any assumptions about this problem to clarify your proof. But do not allow your assumption(s) to contradict what is in the problem statement.

(Hint: Start with two skis and two skiers.) How would you match them? Continue to three skis and three skiers, and identify a strategy.)

```
def ski_rental(skis, skiers):
  """
  Assigns skis to skiers so that the sum of the absolute differences between
  the heights of each skier and her/his skis is minimized.

  Args:
    skis: A list of the heights of the skis.
    skiers: A list of the heights of the skiers.

  Returns:
    A list of the skiers, each with their assigned skis.
  """

  # Sort the skis and skiers in increasing order.
  skis.sort()
```

```
    skiers.sort()

    # Initialize the assignment of skis to skiers.
    assignments = []
    for i in range(len(skis)):
      assignments.append((skis[i], skiers[i]))

    # Iterate over the assignments and swap the skis of two skiers if it
    # reduces the sum of the absolute differences.
    for i in range(len(skis)):
      for j in range(i + 1, len(skis)):
        if abs(skis[i] - skiers[j]) < abs(skis[j] - skiers[i]):
          assignments[i], assignments[j] = assignments[j], assignments[i]

    return assignments
```

The time complexity of the `ski_rental` function can be analyzed as follows:

1. Sorting Step: Sorting the `skis` and `skiers` lists using a `list Sort()` takes `O(n log n)` time complexity, where `n` is the length of the lists. This is because of the sorting algorithm used in Python's `list.The sort()` method has an average and worst-case time complexity of `O(n log n)`.
2. Assignment Initialization: The initialization of the `assignment` list takes `O(n)` time complexity, where `n` is the length of the lists. This is because it involves iterating over the `ski` list and creating tuples with corresponding skis and skiers.
3. Nested Loop: The nested loops that iterate over the `assignment` list have a time complexity of `O(n2)`. This is because, in the worst case, each assignment is compared with every other assignment to determine if swapping skis would reduce the sum of absolute differences.

Overall, the dominant factor contributing to the time complexity is the nested loop, resulting in a time complexity of `O(n2)`, where `n` is the length of the lists. The sorting step and assignment initialization have lower time complexities in comparison.

To prove the correctness of the algorithm, we can use a proof by induction.

Base case: The algorithm is correct for the case of two skis and two skiers.

In this case, there is only one way to assign the skis to the skiers, and the algorithm will find this assignment.

Inductive step: Assume that the algorithm is correct for the case of k skis and k skiers.

Then, the algorithm is also correct for the case of k + 1 skis and k + 1 skiers.

To see this, let's consider the first-k skiers. The algorithm will assign the first k skis to these skiers in such a way that the sum of the absolute differences between the heights of the skiers and their skis is minimized.

The (k + 1)th skier will then be assigned the (k + 1)th ski, which will minimize the sum of the absolute differences for all skiers.

Therefore, the algorithm is correct for all values of n.

Examples:

Here are some examples of how the algorithm works:

Examples:

1. Skis: [160, 170, 180]
   Skiers: [165, 175, 185]

   Greedy Assignment:
   Skier 1 - Ski Pair 2 (170)
   Skier 2 - Ski Pair 2 (170)
   Skier 3 - Ski Pair 3 (180)

   Sum of absolute differences: |165 - 160| + |175 - 170| + |185 - 180| = 5 + 5 + 5 = 15

2. Skis: [150, 160, 170, 180]
   Skiers: [155, 162, 165, 180]

   Greedy Assignment:
   Skier 1 - Ski Pair 2 (160)
   Skier 2 - Ski Pair 3 (170)
   Skier 3 - Ski Pair 3 (170)
   Skier 4 - Ski Pair 4 (180)

   Sum of absolute differences: |155 - 160| + |162 - 170| + |165 - 170| + |180 - 180| = 5 + 8 + 5 + 0 = 18

In both examples, the greedy algorithm provides assignments that minimize the sum of absolute differences between the heights of the skiers and their assigned skis.

Question 3:

Consider a directed graph in which the only negative edges are those that leave s; all other edges are positive. Can Dijkstra’s algorithm, started at s, fail on such a graph? Prove your answer.

In a directed graph where the only negative edges are those that leave the source vertex, Dijkstra's algorithm will not fail to produce correct shortest path results, assuming there are no negative cycles and that the only negative edges are those that leave s; all other edges are positive. then

code

```
import heapq

def dijkstra(graph, source):
    # Initialize distances to infinity for all vertices except the source
    distances = {vertex: float('inf') for vertex in graph}
    distances[source] = 0

    # Priority queue to store vertices and their tentative distances
    queue = [(0, source)]

    while queue:
        # Get the vertex with the minimum tentative distance from the priority queue
        current_distance, current_vertex = heapq.heappop(queue)

        # Skip if the current distance is greater than the known distance
        if current_distance > distances[current_vertex]:
            continue

        # Traverse neighbors and update distances
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            # If a shorter path is found, update the distance
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))

    return distances

# Example graph represented as an adjacency dictionary (directed graph)
graph = {
    'A': {'B': -4, 'C': -2},
    'B': {'D': 3},
    'C': {'D': 3},
    'D': {'A': 1, 'C': 3}
}

source_vertex = 'A'
```

```
distances = dijkstra(graph, source_vertex)

# Print the shortest distances from the source vertex to all other vertices
for vertex, distance in distances.items():
    print(f"Shortest distance from {source_vertex} to {vertex}: {distance}")
```

The input that we have received is

'A': {'B': -4, 'C': -2},
'B': {'D': 3},
'C': {'D': 3},
'D': {'A': 1, 'C': 3}

we get the outputs as

Shortest distance from A to A: 0
Shortest distance from A to B: -4
Shortest distance from A to C: -2
Shortest distance from A to D: -1

we get the correct shortest distances from all proving this algorithm works.

After the first iteration choosing the minimum in negative edges and then as every thing is positive it take cares of the shortest path as after some time every comparison will be positive and we also knew that there not gonna be any negative cycles in prior.

Algorithm

Dijkstra's algorithm is a greedy algorithm that works by iteratively finding the shortest path from the source vertex to all other vertices in the graph. It does this by maintaining a priority queue of vertices that have not yet been visited, and each vertex in the queue is associated with its distance from the source vertex. At each iteration, the vertex with the shortest distance is removed from the queue, and its neighbors are updated with the new shortest distances.

The algorithm works as follows:

1. Initialize a priority queue of all vertices in the graph.
2. Set the distance of the source vertex to 0 and the distance of all other vertices to infinity.
3. Add the source vertex to the priority queue.

4. While the priority queue is not empty,
    - Remove the vertex with the shortest distance from the priority queue.
    - For each neighbor of the removed vertex:
        - If the distance to the neighbor is greater than the distance to the removed vertex plus the weight of the edge between them, then update the distance to the neighbor.
        - Add the neighbor to the priority queue.
5. The algorithm terminates when the priority queue is empty.

The correctness of Dijkstra's algorithm can be proved by induction. The base case is the source vertex, which is guaranteed to have a distance of 0. The inductive step is that if the distance to a vertex is correct, then the distances to all of its neighbors are also correct. This is because the algorithm only updates the neighbors' distances if the new distance is shorter than the old distance.

Dijkstra's algorithm is a very efficient algorithm for finding the shortest paths in graphs. It is typically O(E log V), where E is the number of edges in the graph and V is the number of vertices in the graph. However, the algorithm can be slower for graphs with negative edge weights, because it has to check for negative cycles.