Problem 1

(a)

Iteration #	$\pi = (\pi[A], \pi[B], \pi[C], \pi[D], \pi[E])$	$par = (par[A], par[B], par[C], par[D], par[E])$
1	[∞,0,∞,∞,∞]	[null,null,null,null,null]
2	[2,0,5,∞,4]	[B,null,B,null,B]
3	[2,0,5,8,4]	[B,null,B,A,B]
4	[2,0,5,6,4]	[B,null,B,E,B]
5	[2,0,5,6,4]	[B,null,B,E,B]

The values displayed are equal to the minimum distances from the source vertex. This algorithm works because all of the edges have positive weights.

(b)

Iteration #	$\pi = (\pi[A], \pi[B], \pi[C], \pi[D], \pi[E])$	$par = (par[A], par[B], par[C], par[D], par[E])$
1	[∞,0,∞,∞,∞]	[null,null,null,null,null]
2	[∞,0,3,∞,1]	[null,null,B,null,B]
3	[∞,0,3,∞,1]	[null,null,B,null,B]
4	[∞,0,3,4,1]	[null,null,B,C,B]
5	[-4,0,3,4,1]	[D,null,B,C,B]
6	[-4,-2,3,4,1]	[D,A,B,C,B]

The values are not equal to the shortest paths this time because Dijkstra's algorithm assumes that once a vertex has been visited (i.e., its shortest path is determined), its shortest path will never change. This assumption holds true only when all edge weights are non-negative. In this graph, due to the existence of a negative loop ($ABCD$), there is no shortest path to any of the vertices, as we can loop through $ABCD$ repeatedly, continuously decreasing the shortest path to any node.

Problem 2

Approach 1: Set the weight of each edge (u, v) to be $w_{u\to v} = -\log(p(u, v))$. Since $p(u, v) \in (0, 1]$, we have $-\log(p(u, v)) \geq 0$. This transformation allows us to apply Dijkstra's algorithm, as it ensures that all edge weights are non-negative.

Using Dijkstra's algorithm with edge weights $\{w_{u\to v}\}_{u\to v}$, we can find the shortest path from the source node s (Panucci's Pizzeria) to each destination node d. The shortest path from s to d will be a path $P = (s \to v_1, v_1 \to v_2, \ldots, v_k \to d)$ that minimizes the total sum of edge weights along the path:

$$\sum_{(u\to v)\in P} w_{u\to v} = \sum_{(u\to v)\in P} -\log(p(u, v)) = -\log\left(\prod_{(u\to v)\in P} p(u, v)\right).$$

Thus, minimizing $\sum_{(u\to v)\in P} w_{u\to v}$ effectively maximizes the probability of safe passage $\prod_{(u\to v)\in P} p(u, v)$.

Approach 2: Let $w_{u\to v} = p(u, v)$ denote the weight associated with the edge $u \to v$. We can modify Dijkstra's algorithm to maximize the probability of safe passage, i.e., the product of the edge weights, rather than the sum of the edge weights. As discussed in the lecture, the key changes are in the initialization of Dijkstra's algorithm and the edge weight update condition. Specifically, we initialize $\pi(v)$ for all $v \neq s$ to be the worst possible value, i.e., 0, and we initialize $\pi(s)$ to the best possible value, i.e., 1. At each step, we select the vertex in the priority queue that offers the safest passage from the source node. This is achieved by popping the vertex u with the maximum value of $\pi(u)$ and then updating the safety of its neighbors by checking the value of $\pi(v)$ and comparing it with $\pi(u) \times p(u, v)$. The pseudocode for this approach is as follows:

Algorithm 1 Dijkstra's Algorithm for Maximum Safety Probability

```
function DIJKSTRA(starting vertex $s$)
    for all $v \in V$ do
        $\pi[v] \leftarrow 0$                         ▷ Worst possible value for all vertices $v \neq s$
        $par[v] \leftarrow$ NULL
    end foreach
    $\pi[s] \leftarrow 1$                             ▷ Best value for $s$
    $pq \leftarrow$ empty binary heap priority queue
    for all $v \in V$ do
        INSERT($pq, v, \pi[v]$)
    end foreach
    while $pq$ is not empty do
        $u \leftarrow$ DEL-MAX($pq$)                  ▷ Pop the safest city
        for all neighbor $v$ of $u$ do
            if $\pi[v] < \pi[u] \times p(u, v)$ then    ▷ Update the probability of safe passage if necessary
                INCREASE-KEY($pq, v, \pi[u] \times p(u, v)$)
                $\pi[v] \leftarrow \pi[u] \times p(u, v)$
                $par[v] \leftarrow u$
            end if
        end foreach
    end while
end function
```

Problem 3

The idea is to use the `Merge-and-Count` algorithm twice with different input arrays. Recall that the function `Merge-and-Count` takes as input two sorted arrays A and B, and merges them while counting the number of inversions. The pseudocode can be described as follows:

Algorithm 2 Merge-and-Count

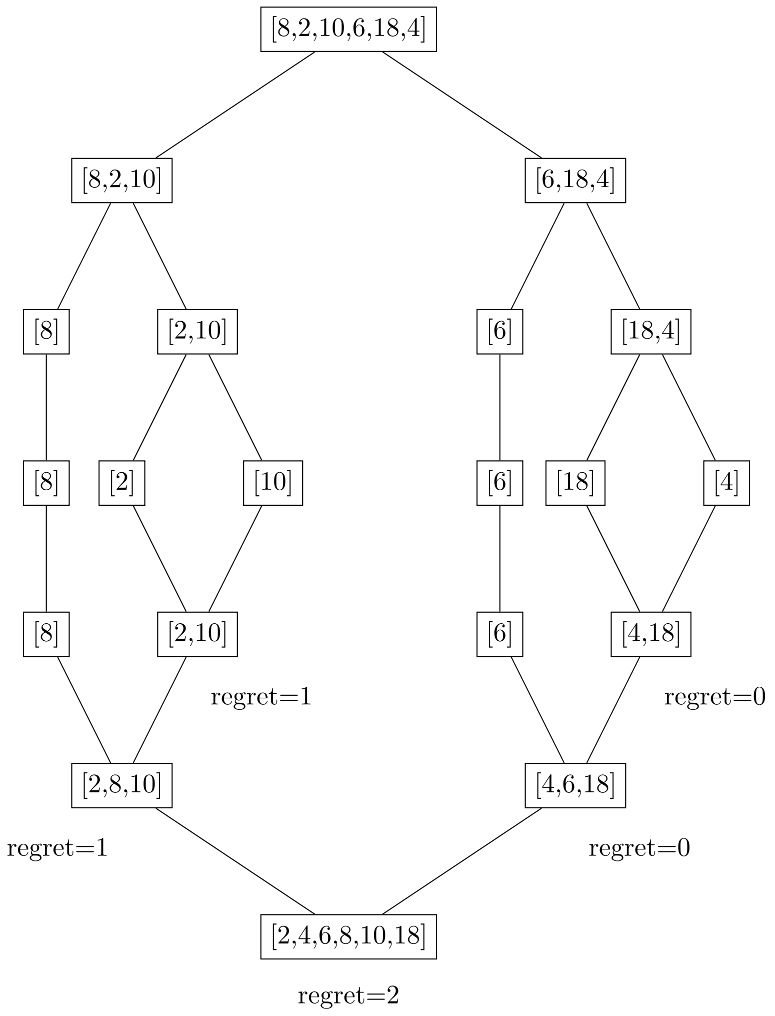
```
function MERGE-AND-COUNT($A, B$)                                ▷ Algorithm 1: Merge-and-Count
    Input: Sorted arrays $A = \{a_i\}$ and $B = \{b_j\}$
    Output: Merged sorted array $C = A \cup B$ and count $r_{AB} = \#\{(i, j) \mid a_i > b_j\}$
    Initialize $r_{AB} \leftarrow 0$ and $C \leftarrow$ empty array
    Let $i \leftarrow 1$, $j \leftarrow 1$
    while $i \leq |A|$ and $j \leq |B|$ do
        if $A[i] > B[j]$ then
            $r_{AB} \leftarrow r_{AB} + (|A| - i + 1)$               ▷ All remaining $a_k$ in $A$ satisfy $a_k > b_j$
            Append $B[j]$ to $C$
            $j \leftarrow j + 1$
        else
            Append $A[i]$ to $C$
            $i \leftarrow i + 1$
        end if
    end while
    while $i \leq |A|$ do
        Append $A[i]$ to $C$
        $i \leftarrow i + 1$
    end while
    while $j \leq |B|$ do
        Append $B[j]$ to $C$
        $j \leftarrow j + 1$
    end while
    return $(r_{AB}, C)$
end function
```

Algorithm 3 Stock-Regret

```
function STOCK-REGRET($S$)                                      ▷ Algorithm 2: Stock-Regret
    Input: Array $S = \{s_1, s_2, \ldots, s_n\}$
    Output: Total regret count $r$ and sorted array $L$
    if $n = 1$ then
        return $(0, S)$                                         ▷ Base case: single element
    end if
    $A \leftarrow S[1 : \lfloor n/2 \rfloor]$                   ▷ First half of $S$
    $B \leftarrow S[\lfloor n/2 \rfloor + 1 : n]$               ▷ Second half of $S$
    $(r_A, A) \leftarrow$ Stock-Regret$(A)$                     ▷ Recursive call on $A$
    $(r_B, B) \leftarrow$ Stock-Regret$(B)$                     ▷ Recursive call on $B$
    $(\tilde{r}_{AB}, L) \leftarrow$ Merge-and-Count$(A, B)$    ▷ Merge $A$ and $B$ to produce sorted array $L$
    $(r_{AB}, \tilde{L}) \leftarrow$ Merge-and-Count$(B, 3A)$   ▷ Count pairs $(i, j)$ where $3a_i < b_j$
    return $(r_A + r_B + r_{AB}, L)$
end function
```

Problem 3 continued on next page...

In the pseudocode above, the line `Merge-and-Count`(A, B) ensures that the array L is a sorted array by properly merging A and B, and the line `Merge-and-Count`(B, 3A) counts the number of elements $s \in A$ and $s' \in B$ for which $3s < s'$. Note that $\tilde{r}_{AB}$ and $\tilde{L}$ are not used in the recursive calls. One can also modify the `Merge-and-Count` function directly, which will eliminate the need for additional memory. As an example, running the above algorithm will generate the following recursive calls of the `Stock-regret` function.



The overall complexity of the algorithm can be described by the recurrence relation:

$$T(n) = 2T(n/2) + O(n)$$

where $T(n)$ is the total time for an array of size n, $2T(n/2)$ accounts for the recursive calls on the two halves, and $O(n)$ accounts for the time taken to count cross-pairs and merge the two halves, assuming n is a power of 2. This results in a total time complexity of $T(n) = O(n \log n)$.

Problem 4

(a)

	Input Array	**Output Array**
1st Call	$\boxed{-1}\boxed{5}\boxed{-7}\boxed{0}$	$\boxed{-7}\boxed{-1}\boxed{0}\boxed{5}$
2nd Call	$\boxed{-1}\boxed{5}$	$\boxed{-1}\boxed{5}$
3rd Call	$\boxed{-1}$	$\boxed{-1}$
4th Call	$\boxed{5}$	$\boxed{5}$
5th Call	$\boxed{-7}\boxed{0}$	$\boxed{-7}\boxed{0}$
6th Call	$\boxed{-7}$	$\boxed{-7}$
7th Call	$\boxed{0}$	$\boxed{0}$

(b)

	Input Array	**Output Array**
1st Call	$\boxed{5}\boxed{3}\boxed{-1}\boxed{4}$	$\left(4, \boxed{-1}\boxed{3}\boxed{4}\boxed{5}\right)$
2nd Call	$\boxed{5}\boxed{3}$	$\left(1, \boxed{3}\boxed{5}\right)$
3rd Call	$\boxed{5}$	$\left(0, \boxed{5}\right)$
4th Call	$\boxed{3}$	$\left(0, \boxed{3}\right)$
5th Call	$\boxed{-1}\boxed{4}$	$\left(0, \boxed{-1}\boxed{4}\right)$
6th Call	$\boxed{-1}$	$\left(0, \boxed{-1}\right)$
7th Call	$\boxed{4}$	$\left(0, \boxed{4}\right)$