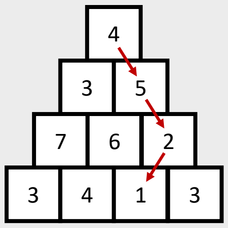
Algorithms Example Sheet 2: Problems Solution Notes

These notes have not be fully proofread. So if you find any typos/mistakes or have any suggestions (even if very small), please do let me know.

1 Dynamic programming

Exercise 2.P.1 [Pyramid of numbers] You are given a pyramid of boxes each of which contains a number. You start at one at the top and at each step you move to the level below in one of the adjacent boxes. Your goal is to find the path with minimum sum of entries. For example, in the image below the largest sum is achieved by the path shown:



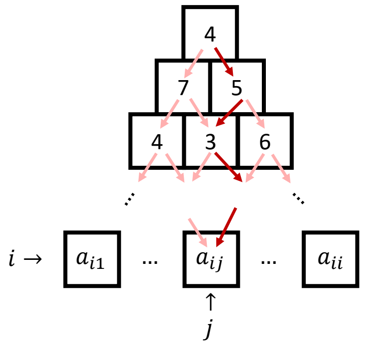
(a) Design an algorithm to efficiently solve this problem.

(b) What is the time and space complexity of your approach?

(c) Write pseudocode for the top-down and the bottom-up approach.

(d) **(I)** Implement this algorithm. (You may want to submit your solution to **[LeetCode 120]**)

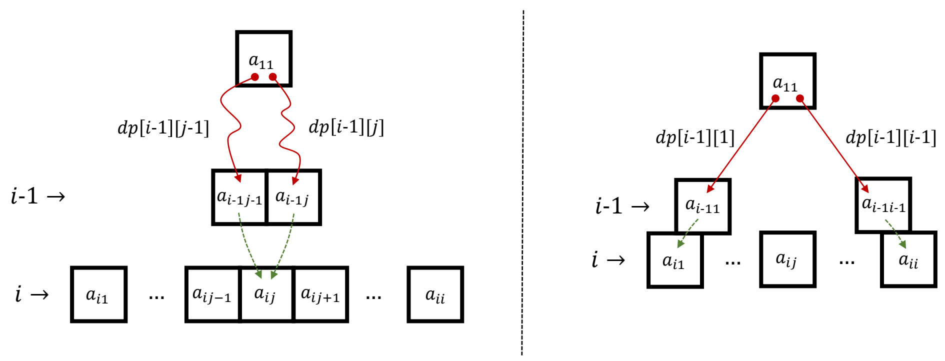
(a) The main idea for solving this problem is to find the minimum sum path to each box. In particular, we define $dp[i][j]$ to be the minimum sum path from the top to the j-th box on level i (1-based).



Then, the table dp satisfies the following properties:

- Base case: $dp[1][1] = a[1][1]$
- $dp[i][1] = a[i][1] + dp[i-1][1]$ for $i > 1$, since the box on the left boundary can be reached only through one path.
- $dp[i][i] = a[i][i] + dp[i-1][i-1]$ for $i > 1$, since the box on the right boundary can be reached only through one path.
- $dp[i][j] = a[i][j] + \min(dp[i-1][j-1], dp[i-1][j])$, since any box in the middle of the level can be reached from either parent.

Then the answer is the minimum over all $dp[N][\cdot]$ values.



Note 1: As with most DP problems, (we will see when we study graphs) that this is an instance of a shortest path problem on a DAG. However, this representation allows us to use less memory.

Note 2: We could also start from the bottom and try to reach the top box.

(b) There are $n \cdot (n-1)/2$ states and each recurrence relation equation takes $\mathcal{O}(1)$ time. Hence, this is an $\mathcal{O}(n^2)$ time algorithm. By noticing that when computing the $dp[i][\cdot]$ values, we only need $dp[i-1][\cdot]$ we can only store the latest row of dp, leading to $\mathcal{O}(n)$ memory.

(c) See the implementations below.

(d) The following is a recursive (top-down) implementation of the DP recurrence relation:

```java
class Solution {
    int dp[][];
    boolean isComputed[][];
    public int minimumTotal(List<List<Integer>> triangle) {
        dp = new int[triangle.size()][triangle.size()];
        isComputed = new boolean[triangle.size()][triangle.size()];

        int mn = Integer.MAX_VALUE;
        for (int i = 0; i < triangle.size(); ++i) {
            mn = Math.min(mn, minSum(triangle, triangle.size() - 1, i));
        }
        return mn;
    }

    public int minSum(List<List<Integer>> triangle, int i, int j) {
        if (isComputed[i][j]) return dp[i][j];
        dp[i][j] = triangle.get(i).get(j);
        if (i == 0 && j == 0) return dp[i][j];
        if (j == 0) dp[i][j] += minSum(triangle, i-1, j);
        else if (j == i) dp[i][j] += minSum(triangle, i-1, j-1);
        else dp[i][j] += Math.min(minSum(triangle, i-1, j), minSum(triangle,
           i-1, j-1));
        isComputed[i][j] = true;
        return dp[i][j];
    }

}
```

The following is an iterative (bottom-up) implementation of the DP recurrence relation which has linear memory using memoisation:

```java
class Solution {
    public int minimumTotal(List<List<Integer>> triangle) {
        int dp[][] = new int[triangle.size()][2];
        int cur = 0;
        dp[0][1] = triangle.get(0).get(0);
        for (int i = 1; i < triangle.size(); ++i) {
            int prev = 1 - cur;
            for (int j = 0; j <= i; ++j) {
                    dp[j][cur] = triangle.get(i).get(j);
                    if (j == 0) dp[j][cur] += dp[j][prev];
                    else if (j == i) dp[j][cur] += dp[j - 1][prev];
                    else dp[j][cur] += Math.min(dp[j][prev], dp[j-1][prev]);
                }
                cur = 1 - cur;
            }
            int mn = dp[0][1 - cur];
            for (int i = 0; i < triangle.size(); ++i) {
                mn = Math.min(mn, dp[i][1 - cur]);
            }
            return mn;
        }
}
```

And a shorter implementation..

```
/* Code by : zmith_nh */
public int minimumTotal(List<List<Integer>> triangle) {
   int n = triangle.size();

   if(n == 0) return 0;
   int[] dp = new int[n+1];

   while(n > 0){
      int[] curr = new int[n];
      for(int i = 0; i < curr.length; i++){
         curr[i] = Math.min(dp[i], dp[i+1]) + triangle.get(n-1).get(i);
      }

      dp = curr;
      n--;
   }

   return dp[0];
 }
```

Question: *Find a counterexample for the following greedy algorithms:*

- Start from the root and always follow the smallest value.
- Start from the bottom and always follow the smallest value.
- Start from the root and always follow the smallest sum two steps ahead.

Exercise 2.P.2 [Longest Common Substring] The *longest common substring* between two strings is defined as the longest (continuous) string s that appears in both strings. For example, the longest common substring of `abcarexample` and `simplexactcat` is $s = $ mple. Design an algorithm to retrieve the longest common substring. What is the time and space complexity of your algorithm?

We can define $dp[i][j]$ to be the length of the longest common substring ending on the i-th character (using indexing at 1) of the first string and the j-th character of the second string. Then, we make the following observations:

1. If $A[i] \neq B[j]$, then $dp[i][j] = 0$.
2. If $A[i] = B[j]$, then $dp[i][j] = dp[i-1][j-1] + 1$.
3. The base case $dp[0][x] = dp[y][0] = 0$ for all x and y.

More concisely, the recurrence relation can be written as

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1 & \text{if } A[i] = B[j] \\ 0 & \text{otherwise} \end{cases}.$$

The longest common substring will have to end at some indices i and j, hence we have to find the maximum of these values. By noticing that we always need the values of the previous row (row $i-1$), we can implement this with linear memory using memoisation.
The following solution implements the iterative longest common substring algorithm for **[LeetCode 718]**.

```java
class Solution {
    public int findLength(int[] A, int[] B) {
        int dp[][] = new int[B.length + 1][2];
        int cur = 0;
        int mx = 0;
        for (int i = 0; i < A.length; ++i) {
            for (int j = 0; j < B.length; ++j) {
                if (A[i] == B[j]) dp[j+1][cur] = dp[j][1-cur] + 1;
                else dp[j+1][cur] = 0;
                mx = Math.max(mx, dp[j+1][cur]);
            }
            cur = 1-cur;
        }
        return mx;
    }
}
```

The following code implements the recursive version,

```java
class Solution {
    int dp[][];
    public int findLength(int[] A, int[] B) {
        dp = new int[A.length + 1][B.length + 1];
        for (int i = 0; i <= A.length; ++i) {
            for (int j = 0; j <= B.length; ++j) {
                dp[i][j] = -1;
            }
        }
        int mx = 0;
        for (int i = 0; i <= A.length; ++i) {
            for (int j = 0; j <= B.length; ++j) {
                mx = Math.max(mx, lcs(A, B, i, j));
            }
        }
        return mx;
    }

    public int lcs(int[] A, int[] B, int i, int j) {
        if (dp[i][j] != -1) return dp[i][j];
        if (i == 0 || j == 0) return 0;
        if (A[i-1] == B[j-1]) dp[i][j] = lcs(A, B, i - 1, j - 1) + 1;
        else dp[i][j] = 0;
        return dp[i][j];
    }
}
```

Note: This is a slightly simpler problem than the longest common subsequence problem, in the sense that there are only $\Theta(n^2)$ possible substrings in each string, while there are $\Theta(2^n)$ possible subsequences in each string.
Further reading: There is actually a more efficient way to find the longest common substring in $\mathcal{O}(n \log n)$ time (using suffix arrays) or in $\mathcal{O}(n)$ time (using suffix trees).

Exercise 2.P.3 [Bounded knapsack] In the bounded knapsack problem each item can be used at most c_i times.

(a) How would you solve the knapsack problem by reducing it to the 0/1 knapsack? What is the time complexity for this?

(b) (optional +) Read this article about speeding this up.

(a) For the i-th item, we can create c_i copies of it and then run the 0/1 knapsack algorithm. There will be a total of $\sum_{i=1}^n c_i$ items, so the time complexity of the approach in Exercise **??** will take $\mathcal{O}\left(\left(\sum_{i=1}^n c_i\right) \cdot C\right)$ time.

(b) One of the approaches in the article, instead of creating c_i copies of the i-th item, it creates $\lfloor \log_2 c_i \rfloor \leq \log_2 C$ items with weights $w_i, 2 \cdot w_i, 2^2 \cdot w_i, 2^3 \cdot w_i, \ldots$ and values $v_i, 2 \cdot v_i, 2^2 \cdot v_i, 2^3 \cdot v_i, \ldots$. These correspond to grouping the elements in groups of size a power of 2. Then, if the optimal solution uses x instances of item i, then this can be achieved by picking the groups whose index appears in the binary representation of x. For example, if $x = 101011_2$, then we should take the groups with weight $w_i, w_i \cdot 2^1, w_i \cdot 2^3, w_i \cdot 2^5$. Hence, this takes $\Theta((n \log_2 C) \cdot C)$.

Exercise 2.P.4 [Unbounded knapsack]

(a) Extend your solution for 0/1 knapsack to the case where you have an infinite supply of each item.

(b) What is the time and space complexity of your approach?

(c) **(I)** Implement the algorithm and test your solution on **[GeeksForGeeks Unbounded knapsack]**.

(d) Explain how you can modify your algorithm to count the number of optimal subsets.

(a) One way would be to add $\lfloor C/w_i \rfloor$ instances of item i and run the 0/1 knapsack problem (or the bounded knapsack problem) and get a solution with running time $\Theta((\sum_{i=1}^n \lfloor C/w_i \rfloor) \cdot C)$ (or $\Theta((n \cdot \log_2 C) \cdot C)$).

Another way that is faster is to notice that if we fill in the $dp[i][\cdot]$ array from bottom to top, then we effectively allow re-using an item many times. For example if the dp array before processing item with $w_i = 3$ ($v_i = 20$), was the following Then one update, gives the following The second update does not

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	$-\infty$	2	$-\infty$	6	$-\infty$	8	$-\infty$	12	$-\infty$	14

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	$-\infty$	2	20	6	$-\infty$	8	$-\infty$	12	$-\infty$	14

update the array, the third update gives: And the interesting part is that the fourth update, uses the

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	$-\infty$	2	20	6	25	8	$-\infty$	12	$-\infty$	14

entry where the i-th item was used (namely 3): More formally, the recurrence relation is given by (the underlined part indicates the difference from the 0/1 knapsack):

$$dp[i][w] = \begin{cases} 0 & \text{if } i = 0 \land w = 0 \\ -\infty & \text{if } i = 0 \land w > 0 \\ \max(dp[i-1][w], v_i + \underline{dp[i][w - w_i]}) & \text{otherwise} \end{cases}$$

and the following code implements this approach:

```java
class Solution{
    static int knapSack(int N, int W, int val[], int wt[]) {
        int dp[] = new int[W + 1];
        for (int i = 1; i <= W; ++i) dp[i] = Integer.MIN_VALUE;
        int mx = 0;
        for (int i = 0; i < N; ++i) {
            for (int j = wt[i]; j <= W; ++j) {
                if (dp[j - wt[i]] != Integer.MIN_VALUE)
                    dp[j] = Math.max(dp[j], val[i] + dp[j - wt[i]]);
                mx = Math.max(mx, dp[j]);
            }
        }
        return mx;
    }
}
```

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	$-\infty$	2	20	6	25	40	$-\infty$	12	$-\infty$	14

(b) If two (or more) items have the same weight, then we can keep only the one with smallest value. We keep $c[i][w]$ the count of configurations that lead to the value $dp[i][w]$. Whenever we find a way to make w using the i-th item leading to a new best (i.e. $dp[i-1][w] < v_i + dp[i][w-w_i]$), we set the counter at $c[i][w] = c[i][w-w_i]$. If $dp[i-1][w] = v_i + dp[i][w-w_i]$, then we set the counter at $c[i][w] = c[i-1][w] + c[i][w-w_i]$. The base case $c[0][0] = 1$.

Exercise 2.P.5 [Knapsack variants]

(a) Given an array of positive integers $\leq U$, determine if it is possible to partition the array into two sets so that they have the same sum. **[LeetCode 416]**

(b) You are given an array of binary strings `strs` and two integers m and n. Return the size of the largest subset of `strs` such that there are at most m 0's and n 1's in the subset. **[LeetCode 474]**

(c) You are given an integer n and you are asked to break it into the sum of k positive integers, where $k \geq 2$, so as to maximise the product of those integers. **[LeetCode 343]**

(a)
```
class Solution {
    public boolean canPartition(int[] nums) {
        int sum = 0;
        for (int v : nums) sum += v;
        if (sum % 2 == 1) return false;
        boolean dp[] = new boolean[sum + 1];
        dp[0] = true;
        for (int i = 0; i < nums.length; ++i) {
            for (int j = sum; j >= nums[i]; --j) {
                dp[j] |= dp[j - nums[i]];
            }
        }
        return dp[sum / 2];
    }
}
```

(b) The idea is to keep track of the number of 0's and 1's in the same way that classical knapsack keeps track of the weights, i.e. by adding one more dimension. The following implementation takes $\mathcal{O}\left(n \cdot C^2\right)$ time and ΘC^2 space.

```
class Solution {
    public int countZeros(String s) {
        int zeros = 0;
        for (int i = 0; i < s.length(); ++i) {
            if (s.charAt(i) == '0') ++zeros;
        }
        return zeros;
    }
    public int findMaxForm(String[] strs, int m, int n) {
        int[][] dp = new int[m+1][n+1];
        dp[0][0] = 1;
        int mx = 1;
        for (int i = 0; i < strs.length; ++i) {
            int zeros = countZeros(strs[i]);
            int ones = strs[i].length() - zeros;
            for (int j = m; j >= zeros; --j) {
                for (int k = n; k >= ones; --k) {
                    if (dp[j-zeros][k-ones] >= 0) dp[j][k] = Math.max(dp[j][k],
                        dp[j-zeros][k-ones] + 1);
                    mx = Math.max(mx, dp[j][k]);
                }
            }
        }
        return mx - 1;
    }
}
```

(c) Let $dp[i]$ be the maximum product that can be made by breaking i into parts. Then,

$$dp[i] = \max(i - \mathbf{1}_{i=n}, \max_{1<j<i} j \cdot dp[i-j]).$$

```
class Solution {
    public int integerBreak(int n) {
        int dp[] = new int[n + 1];
        dp[1] = 1;
        for (int i = 2; i <= n; ++i) {
            dp[i] = i - 1;
            // If i is not the target, then we can make a product of i.
            if (i < n) dp[i] += 1;
            for (int j = 1; j < i; ++j) {
                dp[i] = Math.max(dp[i], j * dp[i - j]);
            }
        }
        return dp[n];
    }
}
```

Exercise 2.P.6 [Change-making]

(a) Recall the change-making problem (define in FoCS). Design a DP algorithm to determine if it is possible to make change for value C. How does the running time compare to that of the FoCS solution?

(b) **(I)** Implement the algorithm for finding the

(c) How would you find the way to make a change w using the fewest number of coins?

(d) Design an algorithm for counting the number of ways of making change.

(e) **(I)** Implement the algorithm for counting the ways of making change. You can test your implementation on **[LeetCode 518]**.

(a) We change the recurrence relation of the unbounded knapsack problem such that $dp[w]$ indicates whether or not we can make a change of value w. Then,

$$dp[i][w] = \begin{cases} \mathbf{T} & \text{if } i = 0 \land w = 0 \\ \mathbf{F} & \text{if } i = 0 \land w > 0 \\ dp[i-1][w] \lor dp[i][w - w_i] & \text{otherwise} \end{cases}$$

(b) The following code implements this approach:

```
public int coinChange(int[] coins, int amount) {
    boolean dp[] = new boolean[amount + 1];
    for (int coin : coins) {
        for (int j = coin; j < amount; ++j) {
            dp[j] |= dp[j - coin];
        }
    }
    return dp[amount];
}
```

(c) We can just convert the problem to the unbounded knapsack problem where the value of each item is 1.

(d) Instead of using a boolean for the value we can use a counter:

$$count[i][w] = \begin{cases} 1 & \text{if } i = 0 \land w = 0 \\ 0 & \text{if } i = 0 \land w > 0 \\ count[i-1][w] + count[i][w - w_i] & \text{otherwise} \end{cases}$$

Again, we can implement this using ΘC memory.

(e) The following code implements this approach:

```
class Solution {
    public int change(int amount, int[] coins) {
        int count[] = new int[amount + 1];
        count[0] = 1;
        for (int coin : coins) {
            for (int j = coin; j <= amount; ++j) {
                count[j] += count[j - coin];
            }
        }
        return count[amount];
    }
}
```

Note: There exist some more efficient ways of solving this problem. These are described in [?] and [?] (see here and here).

Exercise 2.P.7 [Seam Carving]

(a) Watch this video of seam carving in action (or if you are very interested, this extended one).

(b) Attempt Problem 15.8 in CLRS to see how it is implemented.

The solution is similar to the pyramid problem. We need to find the shortest path from one parallel edge to the other.

Exercise 2.P.8 Attempt **[2019P1Q7]**.

See official solution notes

Exercise 2.P.9 [Max-cost independent set on trees] (optional) Attempt problem 15.6 in CLRS.

You are given a (rooted) tree and you need to select some of the vertices so that you do not select two adjacent ones.
We can solve this using the following dynamic programming algorithm. Let

- $dp[v][0]$ be the max-cost independent set for the subtree of v without using v in the solution.
- $dp[v][1]$ be the max-cost independent set for the subtree of v including v in the solution.

The motivation for this definition is that in order to find the max-cost independent set for the subtree of v, we need to know whether its children were chosen or not. If any of the children was chosen, then we cannot choose v. Otherwise, we can choose the best of the two. This gives rise to the following recurrence relations:

$$dp[v][0] = \begin{cases} 0 & \text{if children}(v) = \emptyset, \\ \sum_{c \in \text{children}(v)} \max(dp[c][0], dp[c][1]) & \text{otherwise} \end{cases}$$

$$dp[v][1] = \begin{cases} 0 & \text{if children}(v) = \emptyset, \\ p[v] + \sum_{c \in \text{children}(v)} dp[c][0] & \text{otherwise} \end{cases}$$

where $p[v]$ is the price of v.
Questions: *Find counterexamples for the following greedy algorithms:*

- In each step choose the vertex with most fewest neighbours (and none of them being in the cover) and include it to the cover.
- Choose either the vertices in the odd levels or the even levels to be in the min-cover.

Exercise 2.P.10 [Longest Increasing Subsequence] The *longest increasing subsequence* of sequence $a_1, \ldots, a_n$ is the longest subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ such that $i_1 \leq i_2 \leq \ldots \leq i_k$ and $a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_k}$. For example, in the sequence $[4; 2; 3; 7; 5; 1; 2; 9]$ the longest increasing subsequence is $[2; 3; 5; 9]$.

(a) Design an algorithm to solve this problem.

(b) **(I)** Implement the LIS algorithm. You may want to test your implementation on **[LeetCode 300]**.

(c) Explain how you could retrieve a longest increasing subsequence.

(d) (+) Think how you could use binary search to improve the time complexity of your algorithm to $\mathcal{O}(n \log n)$.

(a) Let $dp[i]$ be the longest increasing subsequence terminating at i. In order to determine this, we can look at all possible previous elements $a[j]$ in the sequence such that $a[j] < a[i]$, and we could extend their LIS by one. If there are none, then $dp[i] = 0$, otherwise it is $dp[i] = 1$. Hence, the recurrence equation is given by

$$dp[i] = \max(1, \max_{j<i, a[j]<a[i]} dp[j] + 1).$$

(b)
```java
class Solution {
    public int lengthOfLIS(int[] nums) {
        int dp[] = new int[nums.length];
        int mx = 1;
        for (int i = 0; i < nums.length; ++i) {
            dp[i] = 1; // Sequence with {nums[i]}
            for (int j = 0; j < i; ++j) {
                if (nums[j] < nums[i] && dp[j] + 1 > dp[i]) {
                    dp[i] = dp[j] + 1;
                }
            }
            mx = Math.max(mx, dp[i]);
        }
        return mx;
    }
}
```

(c) For each element i we keep the element that lead to the best sequence. The following code implements this:

```java
public class LongestIncreasingSubsequence {
    public static int[] findALIS(int[] nums) {
        int dp[] = new int[nums.length];
        int par[] = new int[nums.length];
        int mx = 1, maxValue = 0;
        for (int i = 0; i < nums.length; ++i) {
            dp[i] = 1; // Sequence with {nums[i]}
            par[i] = -1;
            for (int j = 0; j < i; ++j) {
                if (nums[j] < nums[i] && dp[j] + 1 > dp[i]) {
                    dp[i] = dp[j] + 1;
                    par[i] = j;
                }
            }
            if (dp[i] > mx) {
               mx = dp[i];
               maxValue = i;
            }
        }
        // Reconstruct the LIS.
        int cur = maxValue;
        int lis[] = new int[mx];
        while (cur != -1) {
         lis[--mx] = nums[cur];
         cur = par[cur];
        }
        return lis;
    }

    public static void main(String[] args) {
        int[] ans = findALIS(new int[] {10,9,2,5,3,7,101,18});
        for (int v : ans) System.out.println(v);
    }
}
```

(d) We make the following observation:

Observation: If $i < j$ and $a[i] < a[j]$, then $dp[i] < dp[j]$.

Proof. Assume otherwise, i.e. that $dp[i] \geq dp[j]$, then we could extend the LIS at i by appending $a[j]$ to obtain a sequence of length $dp[i] + 1 > dp[j]$. This contradicts the optimality of $dp[j]$. □

Hence, if we keep $sorted[i][j]$ to be the smallest element for a LIS of length j using the first i elements, then $sorted[i][\cdot]$ will be sorted in increasing order. For item $a[i]$ we can find the longest sequence ending at this item by binary searching the largest index k such that $sorted[i-1][k] < a[i]$. Then $sorted[i][k+1] = \min(sorted[i-1][k+1], a[i])$. Since we are only updating one entry, we do not need a new array for each i.

This takes $\mathcal{O}(n \log n)$ time and a sample implementation is shown below:

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        int sorted[] = new int[nums.length + 1];
        sorted[0] = Integer.MIN_VALUE;
        int right = 0;
        for (int i = 0; i < nums.length; ++i) {
            int v = 1; // Sequence with {nums[i]}
            int st = 0, en = right;
            while (st < en) {
                // Note the +1.
                int mn = (st + en + 1) / 2;
                if (sorted[mn] < nums[i]) st = mn;
                else en = mn - 1;
            }
            v = st + 1;
            if (v > right) {
               sorted[v] = nums[i];
               ++right;
            } else sorted[v] = Math.min(sorted[v], nums[i]);
        }
        return right;
    }
}
```

Further reading 1: The van Embde Boas data structure (which used to be part of the course) is a data structure that allows finding the predecessor/successor and do insertion for integers in $[1, U]$ in $\mathcal{O}(\log \log U)$ time. Hence, the binary search part of the LIS algorithm can be implemented in $\mathcal{O}(\log \log n)$ time, giving a total complexity of $\mathcal{O}(n \log \log n)$.

Further reading 2: Read the paper "On computing the length of longest increasing subsequences" (1974) by Michael L. Fredman (from example from here) and describe the $\Omega(n \log n)$ lower bound for the longest increasing subsequence problem in the comparison model.

Exercise 2.P.11 [Longest Palindromic Subsequence] A palindrome is a string that reads the same way forwards and backwards, e.g. `abcba`. The *longest palindromic subsequence* of $a_1, \ldots, a_n$ is the longest subsequence that is a palindrome. For example, $s =$ `anotherexample` then LPS is `aerea`. Design an algorithm that efficiently solves this problem. (If you prefer you can attempt **[2013P1Q6]**)

(Solution 1) Let $dp[i][j]$ be the longest palindromic subsequence in $s_1, \dots, s_j$. Then for $j = i$, $dp[i][i] = 1$ and $dp[i][i+1] = 2$ if $s_i = s_{i+1}$, otherwise it is 1. In the general case the longest palindromic subsequence for $[i,j]$ is either thee longest palindromic subsequence for $[i+1,j]$ or for $[i,j-1]$ or $dp[i+1][j-1]+1$ is $s_i = s_j$. Hence, we get the following recurrence relation,

$$dp[i][j] = \begin{cases} 1 & \text{if } j = i \\ 1 & \text{if } j = i+1 \wedge s_i \neq s_{i+1} \\ 2 & \text{if } j = i+1 \wedge s_i = s_{i+1} \\ \max(dp[i+1][j], dp[i][j-1]) & j > i+1 \wedge s_i = s_{i+1} \\ \max(dp[i+1][j], dp[i][j-1], 1 + dp[i+1][j-1]) & \text{otherwise} \end{cases}$$

This gives an $\Theta(n^2)$ algorithm. We can implement this using $\Theta(n^2)$ memory:

```
class Solution {
    public int longestPalindromeSubseq(String s) {
        int dp[][] = new int[s.length()][s.length()];
        for (int len = 0; len < s.length(); ++len) {
            for (int i = 0; i + len < s.length(); ++i) {
                if (len == 0) dp[i][i] = 1;
                else if (len == 1) dp[i][i+1] = 1 + (s.charAt(i) == s.charAt(i+1) ?
                    1 : 0);
                else {
                    dp[i][i + len] = Math.max(dp[i][i + len - 1], dp[i+1][i + len]);
                    if (s.charAt(i) == s.charAt(i+len)) {
                        dp[i][i + len] = Math.max(dp[i][i + len], 2 +
                            dp[i+1][i+len-1]);
                    }
                }
            }
        }
        return dp[0][s.length()-1];
    }
}
```

or using $\Theta(n)$ space by observing that when computing $dp[i][i+\ell]$ we only need the entries $dp[i][i+\ell-1]$ and $dp[i][i+\ell-2]$.

```
class Solution {
    public int longestPalindromeSubseq(String s) {
        int dp[][] = new int[s.length()][3];
        for (int len = 0; len < s.length(); ++len) {
            for (int i = 0; i + len < s.length(); ++i) {
                if (len == 0) dp[i][0] = 1;
                else if (len == 1) dp[i][1] = 1 + (s.charAt(i) == s.charAt(i+1) ? 1
                    : 0);
                else {
                    dp[i][len%3] = Math.max(dp[i][(len-1)%3], dp[i+1][(len-1)%3]);
                    if (s.charAt(i) == s.charAt(i+len)) {
                        dp[i][len%3] = Math.max(dp[i][len%3], 2 +
                            dp[i+1][(len-2)%3]);
                    }
                }
            }
        }
        return dp[0][(s.length()-1)%3];
    }
}
```

(Solution 2) An alternative solution is to find the LCS between the string s and its reverse $\overleftarrow{s}$. This gives an $\Theta(n^2)$ time and $\Theta(n)$ algorithm.
The reason why it works is not obvious. Of course every palindromic subsequence corresponds to a common subsequence of s and $\overleftarrow{s}$. It remains to show the reverse direction, i.e. that if there is an LCS of length ℓ, then there is a common subsequence of the same length that is palindromic.

Assume that the LCS consists of indices $a_1 < \ldots < a_\ell$ and $b_1 > \ldots > b_\ell$ (in the original string) and let $m = \lfloor \ell/2 \rfloor$. If $a_m \leq b_m$, then we can construct the index sequences $a_1 < \ldots < a_m \leq b_m < b_{m-1} < \ldots < b_1$ and $b_1 > b_2 > \ldots > b_m \geq a_m > \ldots > a_1$, which is palindromic and has length at least ℓ.
Otherwise, construct the sequence $b_\ell < b_{\ell-1} < \ldots < b_m < a_m < \ldots < a_\ell$ and $a_\ell > a_{\ell-1} > \ldots > a_m > b_m > \ldots > b_\ell$.

Fact 7.2 in this paper

Exercise 2.P.12 [Bring your own problem I] Find at least one problem solvable using dynamic programming and attempt to solve it (or just bring it to the supervision and be ready to discuss it). (One possible source is the end of Chapter 15 in CLRS)

Greedy algorithms

Exercise 2.P.13 [Minimising waiting time at the queue] There are n customers waiting at a supermarket queue. You know that the i-th person will take t_i minutes to be served. You want to find the order to serve the n customers, which minimises the total waiting time. For example, if $t_1 = 5$, $t_2 = 7$ and $t_3 = 2$, then serving customer 2 (waits 0 minutes), then 3 (waits 7 minutes) and 1 (waits $7+2$ minutes) gives a total of 16 minutes (which is not optimal).

(a) Design an algorithm to find the optimal order to serve the customers.

(b) Prove that your algorithm is optimal.

(a) The time is minimised when serving the customers in decreasing order of t_i. So, an algorithm for this would be sort the serving times and then serve the one that is first, then the one that is second, and so on.

Consider the serving sequence $i_1, \ldots, i_n$, then the total waiting time is

$$0 + t_{1_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_3}) + \ldots = \sum_{j=1}^{n-1} \sum_{k=1}^{j} t_{i_k}.$$

By grouping terms, this is equal to

$$t_1 \cdot (n-1) + t_2 \cdot (n-2) + t_3 \cdot (n-3) + \ldots = \sum_{j=1}^{n-1} (n-j) t_{i_j}.$$

Now, we are ready to prove that the algorithm produces the minimal total waiting time. Assume that the optimal sequence is not sorted. Then, $t_{i_j} < t_{i_k}$ but $j > k$. Hence, the cost contributed by these two terms is $t_{i_j} \cdot (n-j) + t_{i_k}(n-k)$. If we swapped them around, then the cost would be $t_{i_j} \cdot (n-k) + t_{i_k}(n-j)$. Let's compare the two by taking their difference,

$$t_{i_j} \cdot (n-j) + t_{i_k}(n-k) - t_{i_j} \cdot (n-k) + t_{i_k}(n-j) = t_{i_j}(k-j) + t_{i_k}(j-k) = (j-k)(t_{i_k} - t_{i_j}),$$

which is > 0, since both $j - k > 0$ and $t_{i_k} - t_{i_j} > 0$. This means that by making this swap we are reducing the cost for the optimal solution (contradiction).

Note: As we will see in Exercise 14 is an instance of the re-arrangement inequality.

Exercise 2.P.14 [Rearrangement inequality] You are given positive values $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$. Design an algorithm that pairs them up (say using an bijection $f : [n] \to [n]$) so that $\sum_{i=1}^n a_i b_{f(i)}$ is minimised. Prove that your algorithm is optimal.

The idea is to sort the two sequences and obtain $a'_1, \ldots, a'_n$ and $b'_1, \ldots, b'_n$. Then, their inner product is minimised when multiplying the terms of a' with terms of b' in reverse order:

$$\sum_{i=1}^{n} a'_i b'_{n-i+1}.$$

This requires $\mathcal{O}(n \log n)$ time to find by sorting the two sequences.
To prove that this is optimal, we follow an exchange argument as in the previous exercise. Assume that the matching is not as above, then there exist $j < k$ such that $b'_{f(j)} < b'_{f(k)}$. Then the contribution of this to the value is $a'_j \cdot b'_{f(j)} + a'_k \cdot b'_{f(k)}$. If we swap these matchings, then the contribution is $a'_j \cdot b'_{f(k)} + a'_k \cdot b'_{f(j)}$. Let's compare these by taking their difference,

$$a'_j \cdot b'_{f(j)} + a'_k \cdot b'_{f(k)} - a'_j \cdot b'_{f(k)} + a'_k \cdot b'_{f(j)} = a'_j(b'_{f(j)} - b'_{f(k)}) + a'_k(b'_{f(k)} - b'_{f(j)} = (a'_j - a'_k)(b'_{f(j)} - b'_{f(k)}),$$

which is > 0 since $a'_j < a'_k$ and $b'_{f(j)} < b'_{f(k)}$. Hence, by swapping $j \leftrightarrow i$ we get a smaller value than the optimal (contradiction).

Exercise 2.P.15 [Fractional knapsack] You are given a knapsack of capacity W. There are n different types of cheese with weight x_i and total value c_i. You can choose to include a fractional amount of cheese in your knapsack. Design an algorithm to maximise the value of cheese in your knapsack. (You can submit your solution **[GeeksForGeeks FractionalKnapsack]**).

The solution is to sort the cheese by value per weight and include as much of the cheese at the highest value as their is available and fits in the knapsack, then continue with the second and so on. This takes $\mathcal{O}(n \log n)$ time.
To prove this is correct assume that we included $x > 0$ units of weight from cheese i, when there existed some cheese j with higher value to weight ratio than i, that we did not fully use (say y left). Then by replacing these $z = \min(x, y) > 0$ units of cheese i with z units of cheese j we increase the value that can be added to the knapsack.

```java
/*
class Item {
    int value, weight;
    Item(int x, int y){
        this.value = x;
        this.weight = y;
    }
}
*/

class Solution {
    //Function to get the maximum total value in the knapsack.
    double fractionalKnapsack(int W, Item arr[], int n) {
        Arrays.sort(arr, new Comparator<Item>() {
            public int compare(Item o1, Item o2) {
                return Double.compare(o2.value / ((double) o2.weight), o1.value /
                    ((double) o1.weight));
            }
        });
        double value = 0.0;
        for (Item it : arr) {
            if (W >= it.weight) {
                W -= it.weight;
                value += it.value;
            } else {
                value += it.value * W / ( (double) it.weight);
                break;
            }
        }
        return value;
    }
}
```

Further reading: It is possible to implement this in $\mathcal{O}(n)$ expected time using a modification of Quickselect for computing the weighted median.

Exercise 2.P.16 Attempt **[2015P1Q8]**.

Exercise 2.P.17 [Supervision scheduling] I have n supervisions to schedule. Supervision i starts at time a_i and ends at time b_i.

(a) Help me find the minimum number of supervisions that I will have to reschedule so that I do not have any overlapping ones.

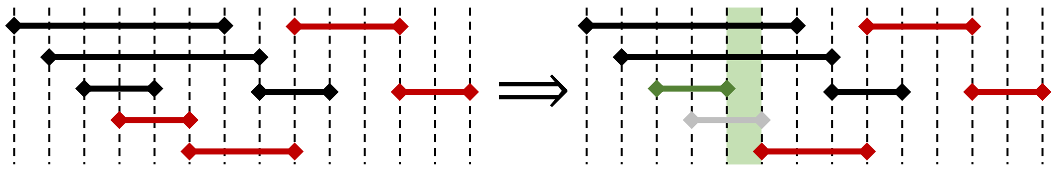
[Source: Hayk Saribekyan]

(b) **(I)** Implement your approach. You may test your implementation on **[LeetCode 435]**.

(c) What if the tasks were not supervisions and they were meetings; and each meeting has a different weight of importance. How do you maximise the total weight of the meetings you attend?

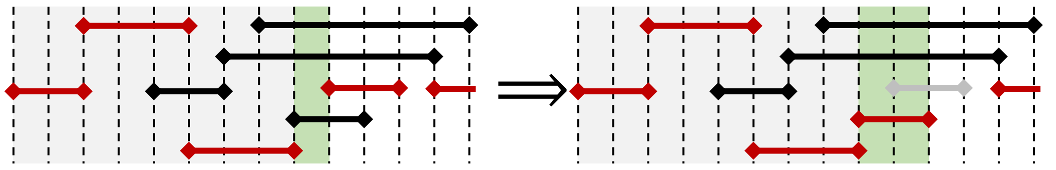
(d) **(I)** Implement your approach. You may test your implementation on **[LeetCode 1235]**.

(a) A key observation is that we can always include the supervision that ends soonest (smallest ending time). Consider an optimal subset of intervals and assume that it does not use the interval that ends soonest, then by replacing the leftmost interval with the interval that ends soonest, we give room for more intervals.



Still optimal and we also have some more spare space

Now, this generalises to an arbitrary position. Assume that we have found the optimal subset of intervals up to position x, then the next interval to be added is going to be the interval that starts after x and ends soonest. The proof for why this works is similar to case of the earliest ending interval. Consider all intervals that start after x and assume that we had chosen a different interval, then by swapping it with the one ending soonest we obtain a valid solution:



Still optimal and we also have some more spare space

Hence, we can formalise this as a proof by induction: The n intervals produced by the algorithm are part of an optimal solution. The base case follows from the first observation and the inductive step follows from the argument above.

This hints to the following algorithm:

1. Sort the intervals by ending position.
2. As iterating through the sorted intervals keep the ending r of the last interval (which is the rightmost ending). If the current interval begins after r then add it to the solution and update r.

This gives an $\mathcal{O}(n \log n)$ algorithm.

Question 1: *Why are we saying that this produces **an** optimal solution?* Because there could be multiple solutions with the same cardinality,

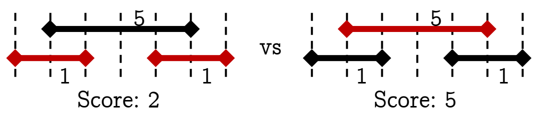
Question 2: *What time would you suggest to a supervisor that is allocating slots using this algorithm?* The earliest possible (maybe a minute earlier than that).

(b) Here is one possible implementation:

```java
class Solution {
    public int eraseOverlapIntervals(int[][] intervals) {
        Arrays.sort(intervals, new Comparator<int[]>(){
            @Override
            public int compare(int[] a, int[] b) {
                return Integer.compare(a[1], b[1]);
            }
        });
        // we should always take the interval that ends soonest.
        int recent_end = intervals[0][1];
        int total = 1;
        for (int i = 1; i < intervals.length; ++i) {
            if (intervals[i][0] >= recent_end) {
                recent_end = intervals[i][1];
                ++total;
            }
        }
        return intervals.length - total;
    }
}
```

Note: If the coordinates are integers and the range is small (for example the supervision times are at :00, :15, :30, :45 of an hour), then we could use counting sort.

(c) If each meeting has a weighting then we can see that the above algorithm does not work. For example, consider the following configuration:



Instead, we formulate a recurrence relation, that we solve using DP. Let $dp[x]$ be the maximum profit until time x. Then, if s_0 is the starting time, s_i and e_j is the starting and ending time of the i-th interval and p_i is the weight,

$$dp[x] = \begin{cases} 0 & \text{if } x = s_0, \\ \max(dp[x], \max_{j:e_j = x} dp[s_j] + p_j) & \text{otherwise.} \end{cases}$$

This requires $\mathcal{O}(n \log n)$ time for sorting (and mapping the coordinates in $[0, 2n-1]$) and $\mathcal{O}(n)$ space.

(d) Here is a sample implementation:

```
class Solution {
    static class Event {
        int x;
        int intervalId;
        boolean isStart;
        Event(int x, int intervalId, boolean isStart) {
            this.x = x;
            this.intervalId = intervalId;
            this.isStart = isStart;
        }
    };
    public int jobScheduling(int[] startTime, int[] endTime, int[] profit) {
        Event events[] = new Event[2 * startTime.length];
        for (int i = 0; i < startTime.length; ++i) {
            events[2 * i] = new Event(startTime[i], i, true);
            events[2 * i + 1] = new Event(endTime[i], i, false);
        }
        // Make timestamps unique.
        Arrays.sort(events, new Comparator<Event>() {
            @Override
            public int compare(Event a, Event b) {
                if (a.x == b.x) return a.isStart ? 1 : -1;
                return Integer.compare(a.x, b.x);
                }
            });
            int prev = -1, cur_x = -1;
            for (int i = 0; i < events.length; ++i) {
                if (prev != events[i].x) {
                    ++cur_x;
                    prev = events[i].x;
                }
                events[i].x = cur_x;
                if (events[i].isStart) startTime[events[i].intervalId] = cur_x;
            }

            // Solve the DP recurrence.
            int dp[] = new int[cur_x + 1];
            for (int i = 0; i < events.length; ++i) {
                int x = events[i].x;
             if (x > 0) dp[x] = Math.max(dp[x], dp[x - 1]);
                if (!events[i].isStart) {
                   int id = events[i].intervalId;
                    dp[x] = Math.max(dp[x], dp[startTime[id]] + profit[id]);
                }
                // System.out.println(x + " : " + dp[x]);
            }
            return dp[cur_x];
        }
    }
```

Exercise 2.P.18 [Invigilator scheduling] We want to find invigilators to cover the time interval $[S, T]$. We have received several offers from person i to invigilate in the interval $[a_i, b_i]$. Find the minimum number of invigilators needed to cover the time interval $[S, T]$.
If you implement your algorithm, you can test your implementation on **[LeetCode 1326]**.

The solution is greedy. Assume we have covered the interval $[S, x]$, then the next interval to choose should be the interval i with $s_i \leq x$ and ending point e_i as large as possible. Assuming at some point the optimal solution has a different interval, then by replacing this interval with the one that ends farthest, we can cover at least as much time as the other interval, hence, it produces a solution that is at least as good.
Hence, to find the number of intervals needed to cover $[S, T]$ we sort the intervals and then apply this procedure in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space.
The following implementation solves the problem:

```
class Solution {
    static class Interval {
        int s, e;
        Interval(int s, int e) {
            this.s = s;
            this.e = e;
        }
    };
    public int minTaps(int n, int[] ranges) {
        Interval intervals[] = new Interval[ranges.length];
        for (int i = 0; i < ranges.length; ++i) {
            intervals[i] = new Interval(i - ranges[i], i + ranges[i]);
        }
        Arrays.sort(intervals, new Comparator<Interval>() {
            @Override
            public int compare (Interval a, Interval b) {
                return Integer.compare(a.s, b.s);
            }
        });
        int covered_up_to = 0, cur_rightmost = 0, total = 0;
        for (int i = 0; i < intervals.length && covered_up_to < n; ++i) {
            // System.out.println("(" + intervals[i].s + ", " + intervals[i].e + ")");
            //System.out.println("Rightmost: " + cur_rightmost);
                if (intervals[i].s > cur_rightmost) break;
                if (intervals[i].s > covered_up_to) {
                    covered_up_to = cur_rightmost;
                    // System.out.println("New interval up to : " + cur_rightmost);
                    ++total;
                }
                if (intervals[i].e > cur_rightmost) {
                    cur_rightmost = intervals[i].e;
                    if (cur_rightmost >= n) {
                      covered_up_to = cur_rightmost;
                      ++total;
                    }
                }
            }
            if (covered_up_to < n) return -1;
            return total;
        }
}
```

Exercise 2.P.19 [Necklace] You have a necklace with n beads $b_1, \ldots, b_n$. There are k types of beads (for convenience we number these between 0 and $k-1$). We want to find the smallest (continuous) part of the necklace that contains all k different types (if such exists). (Aim for an $\mathcal{O}(n)$ algorithm)
If you want you can test your implementation on **[LeetCode 76]**.

We can use a hash table to keep track of the beads that we have already encountered. By looping through positions $0, \ldots, x$, we can determine when we have encountered all k beads by keeping a counter `cnt` that is incremented each time we see a bead with 0 count in the hash table. Hence, after this loop we have the smallest position x such that $[0, x]$ contains all beads, or know that there does not exist such position.
Now assuming that we know the smallest interval $[\ell, j]$ ending at j (which must be $\geq x$) that contains all bead types. Then to find the smallest interval ending at $j+1$ containing all bead types, we add b_{j+1} to the hash table and start removing beads from the left, i.e. remove b_ℓ as long as the count in the hash table is at least 2.

```
class Solution {
public:

    int norm(int c) {
        return c + 128;
    }

    vector<int> countChars(const string& s) {
        vector<int> count(256, 0);
        for (char c : s) {
            ++count[norm(c)];
        }
        return count;
    }

    string minWindow(string s, string t) {
        std::vector<int> target = countChars(t);
        std::vector<int> found(256, 0);
        int en = 0;
        int count = 0;
        int min_idx = -1;
        int min_len = -1;
        for (int k = 0; k < s.size(); ++k) {
            // Remove the previous value.
            if (k != 0) {
                int prev = norm(s[k-1]);
                if (found[prev] <= target[prev]) {
```

```
                --count;
            }
            --found[prev];
        }
        // Reach the best place for the current value.
        while (count != t.size() && en < s.size()) {
            int cur = norm(s[en]);
            ++found[cur];
            if (found[cur] <= target[cur]) {
                ++count;
            }
            ++en;
        }

        // If no such value exist, then break.
        if (en == s.size() && count != t.size()) {
            break;
        }

        // Else: update the current best.
        int cur_len = en - k;
        if (min_idx == -1 || min_len > cur_len) {
            min_len = cur_len;
            min_idx = k;
        }
    }
    if (min_idx == -1) {
        return "";
    }
    return s.substr(min_idx, min_len);
  }
};
```

Exercise 2.P.20 [Facility location in 1D] Given n points on the x-axis with coordinates x_i, find the optimal position x^* so as to minimise

$$f(x^*) = \sum_{i=1}^{n} |x^* - x_i|.$$

[Hint: How does $f(x^*)$ change as we move x^* from $-\infty$ to ∞?]

Exercise 2.P.21 [Bring your own problem II] Find at least one problem solvable using a greedy approach and attempt to solve it (or just bring it to the supervision and be ready to discuss it).

Divide and conquer

Exercise 2.P.22 [Counting inversions] Modify merge sort to count the number of pairs $i < j$ such that $A[i] > A[j]$ in $\mathcal{O}(n \log n)$ time. How does this help us count the number of swaps performed by insertion sort?
If you want you can test your implementation on **[GeeksForGeeks Count Inversions]**.

(Solution 1) We can use a modification of mergesort. We count the number of inversions in each half of the array recursively. Then we need to count the number of inversions between the two halves. For each element j of the upper half, we will count the number of elements in the first half that are larger that $A[j]$. At the moment we insert j in the merged array, there are $n_1 - i_1$ remaining elements in the first array and all of these will be larger than $A[j]$. Since we are counting the number of inversions for the smaller item, we do not need to count the number of inversions for the larger elements (otherwise, we would double count).

Note: We need to take special care when dealing with equal elements, so that we do not count them as inversions. In the implementation below, in case of equality we insert the element from the first part of the array, so that it is not counted in the second branch.

```java
class Solution {
    // arr[]: Input Array
    // N : Size of the Array arr[]
    //Function to count inversions in the array.
    static long inversionCount(long arr[], long N) {
        return mergeCount(arr, 0, (int) N - 1);
    }

    static long mergeCount(long arr[], int i, int j) {
        if (i == j) return 0;
        int mid = (i + j) / 2;
        long cnt1 = mergeCount(arr, i, mid);
        long cnt2 = mergeCount(arr, mid + 1, j);
        long cntMerge = 0;
        long tmp[] = new long[j - i + 1];
        int out = 0, i1 = i, i2 = mid + 1;
        while (i1 <= mid || i2 <= j) {
            if (i2 == j + 1 || (i1 <= mid && arr[i1] <= arr[i2])) {
                tmp[out] = arr[i1];
                ++i1;
            } else {
                tmp[out] = arr[i2];
                ++i2;
                cntMerge += (mid - i1 + 1);
            }
            ++out;
        }
        for (int k = 0; k < tmp.length; ++k) {
            arr[i + k] = tmp[k];
        }
        return cnt1 + cnt2 + cntMerge;
    }
}
```

(Solution 2) Another way to do this is to count for each (fixed) i the number of items $j < i$ such that $A[i] < A[j]$. This can be done by iterating trough the numbers from $0, \ldots, i-1$ and storing them inside a RBT (with subtree sizes). Then we can count the number of items larger than $A[i]$. Each query takes $\mathcal{O}(\log n)$ time, hence it gives a total of $\mathcal{O}(n \log n)$ time.

Exercise 2.P.23 [Closest pair of points] (+) In the *closest-pair-of-points* problem you are given a collection of N points $\{(x_i, y_i)\}_{i=1}^N$ on the 2D (Euclidean plane). The goal is to find two points with the smallest distance. This exercise guides you through the analysis of the algorithm presented here. It will probably not be clear why scanning finds the smallest distance.

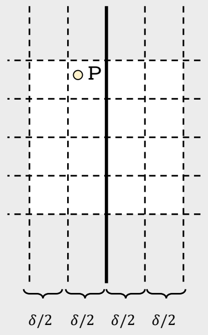
(a) Argue that you can implement the time complexities shown on the right of the algorithm.

(b) Argue that the time complexity of the algorithm is $\mathcal{O}\left(n \log^2 n\right)$.

(c) In the scanning step, why do we consider only points within a distance of δ from the median?

(d) Argue that two points in a square of side length $\delta/2$ have distance less than δ.

(e) (+) By considering a point P within a distance of δ from the median, use the following diagram (or otherwise) to argue that we need to check at most the next 8 points from the other side (in the y-sorted sequence of the points in the strip).



(f) Argue that you can implement sorting by y coordinates in $\mathcal{O}(n)$ time (in each recursive call). Show that the new algorithm takes $\mathcal{O}(n \log n)$ time.

The slides contain the solution for the analysis of the algorithm.
Note 1: We are assuming that distances can be computed in $\mathcal{O}(1)$ time, while in practice this is not realistic.
Note 2: This is actually the optimal time complexity in the comparison-based model.

Exercise 2.P.24 [Range sum queries (I)] In the *range sum query* problem, we are given an array with n elements and operations of the form:

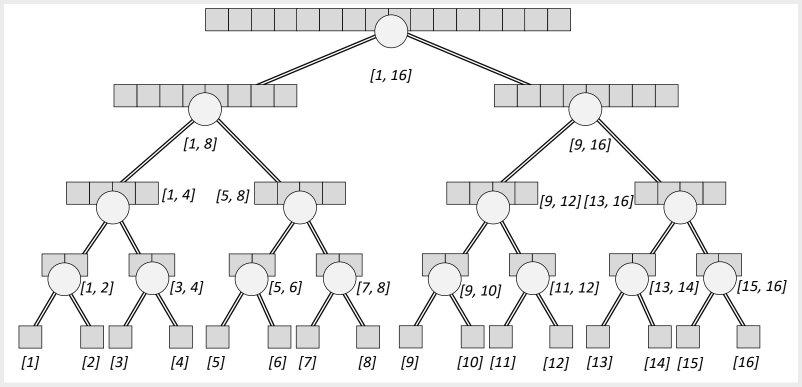
1. Update index i to value v.
2. Query the sum of the entries from index i to index j.

(a) Give a simple algorithm that solves the problem in $\mathcal{O}(1)$ per update and $\mathcal{O}(n)$ per query.

(b) Give an algorithm that solves the problem in $\mathcal{O}(n)$ per update and $\mathcal{O}(1)$ per query.

(c) (+) Split the array into sub-arrays of size roughly $\sqrt{n}$. By keeping the sum for each sub-array separately, argue (in detail) that you can support both operations in $\mathcal{O}(\sqrt{n})$ time.

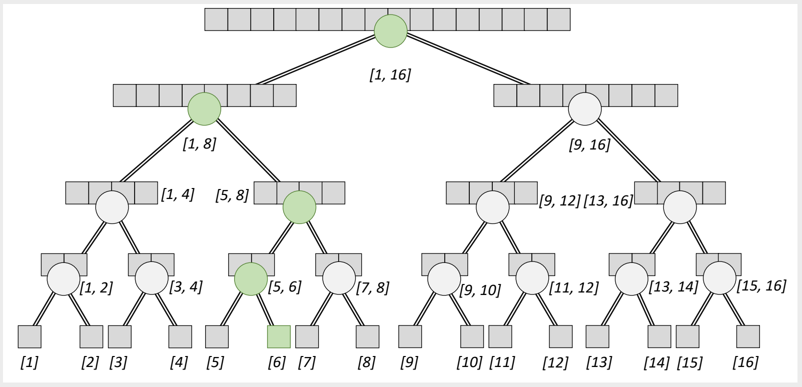
Exercise 2.P.25 [Range sum queries (II)] In this exercise, we will examine a slightly different approach to the range sum query problem than the one we took in Exercise 24. We will store a binary tree whose root maintains the sum of elements in $[1, n]$. This root will have two children the left one storing the sum of elements in $[1, n/2]$ and the right one storing the sum for $[n/2 + 1, n]$. The following diagram shows the case for $n = 16$.



(a) How many nodes are there in this binary tree? What is the height of the tree?

(b) Think how you would implement the update operation.

(c) (+) In order to update element i we start from the root and follow the sub-interval containing i. What is the worst-case time complexity of this operation? (See figure below for updating index 6)

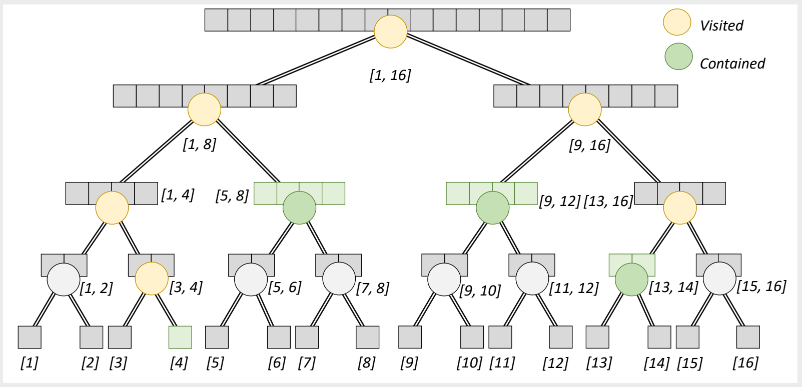


(d) Think how you would implement the query operation.

(e) (++) In order to query the sum from elements $[i, j]$, we start from the root and we recursively consider the following cases (assuming $[u, v]$ is the interval of the current node):

- If $[u, v] \subset [i, j]$, then return the sum of the node without exploring its children.
- If $[u, v] \cap [i, j] = \emptyset$, then return 0.
- Otherwise, recursively call both children and sum their answers.

What is the time complexity for this operation? (See figure below for querying the sum in $[4, 14]$)


Visited
Contained
[1, 16]
[1, 8]
[9, 16]
[1, 4]
[5, 8]
[9, 12]
[13, 16]
[1, 2]
[3, 4]
[5, 6]
[7, 8]
[9, 10]
[11, 12]
[13, 14]
[15, 16]
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16]