Algorithms for Data Science
Exercise 4

Each (sub)subtask gives 1 activity point—if not otherwise noted—for presenting the solution during the class. Recall that during a class session, you can earn at most 2 points for presenting. Each task is labeled with the total number of activity points that you can gain for writing down a solution—which is split among all group members.

The bonus tasks at the end are optional. Try to solve them when you have finished solving the current task!

Task 4.1: *Dynamic Programming* (5 points)

You are given a $(k \times n)$-grid consisting of n columns and k rows. Each grid cell has some non-negative value. Your task is to choose at most one grid cell from each of the n columns such that no two grid cells are touching each other (touching with corners is also forbidden) and the total value is maximized. It suffices that you output the total value.

0	1	2	1	0
1	1	1	1	0
0	1	2	1	2
0	1	2	1	2

An example grid ($n = 5$ and $k = 4$). The chosen cells are shaded in gray. The total value is 6.

Input: array $val[1..n][1..k]$.

Output: total value of the chosen grid cells.

Solve the problem for any $k \geq 4$ via the following methods. What is the running time?

(a) (2 points) Backtracking

Hint: First solve the problem for the restricted case where the input specifies some $j \in \{1, \ldots, k\}$ and, within the *last* column, we are allowed to pick only the grid cell of the j-th row.

Solution:

Running time: Exponential in n (in more detail: $k^{\Theta(n)}$) as for each column we have $2k$ recursive calls.

```
/* We call our function below trying all combinations for the last column.  */
maxVal = 0
for j = 1 to k do
    maxVal = max(maxVal, AlgoBack(val,n,j))
return maxVal
```

```
/* The output is the maximum value attainable in val[1..i][1..k] under the condition
   that from the i-th column we may take only the element val[i][k].              */
AlgoBack(val,i,j)
    if i == 0 then
        return 0
    maxVal = 0
    // The case that we don't take val[i][j].
    for t = 1 to k do
        maxVal = max(maxVal, AlgoBack(val,i-1,t))
    // The case that we take val[i][j].
    for t = 1 to k do
        // The cell val[i − 1][t] touches val[i][j] if t ∈ {j − 1, j, j + 1}.  As k > 3,
           there is at least one non-touching cell.
        if t < j − 1 or t > j + 1 then
            maxVal = max(maxVal, val[i][j] + AlgoBack(val,i-1,t))
    return maxVal
```

(b) (2 points) Recursion with Memoization

Solution: We just add a dictionary *memo* to the last solution. The modifications are in blue.

```
memo = {}
AlgoMem(val,i,j)
    if i == 0 then
        return 0
    if (i, j) ∉ memo then
        maxVal = 0
        for t = 1 to k do
            maxVal = max(maxVal, AlgoMem(val,i-1,t))
        for t = 1 to k do
            if t < j − 1 or t > j + 1 then
                maxVal = max(maxVal, val[i][j] + AlgoMem(val,i-1,t))
        memo[i][j] = maxVal
    return memo[i][j]
```

For this problem, we can assume that the dictionary is a two-dimensional array $memo[1..n][1..k]$: initially, we fill the array with a value that never appears as a solution, for instance with -1. Thus, if we get the query *Is* $(i, j) \notin memo$*?*, we answer with true if and only if $memo[i][j] == -1$. Thus, the access and modification times for *memo* are constant in such a solution.

Observe that the running time is proportional to the total number of recursive calls. Note that within `AlgoMem(val,i,j)` $\Theta(k)$ recursive calls are made but only if $i \geq 1$ and only for the first time `AlgoMem(val,i,j)` is run for the parameters i and j. For any subsequent runs (or if $i = 0$), there are no recursive calls at all within `AlgoMem(val,i,j)`. Therefore the total number of recursive calls is asymptotically equal to $\Theta(k)$ times the number of all combinations of $i \geq 1$ and k. Thus, the resulting running time is $\Theta(nk^2)$.

(c) (2 points) Dynamic Programming

Solution: We do everything bottom up. Time $\Theta(nk^2)$, Space $\Theta(nk)$.

```
DPtable[0..n][1..k] = new array filled with 0s
for i = 1 to n do
    for j = 1 to k do
        maxVal = 0
        for t = 1 to k do
            maxVal = max(maxVal, DPtable[i − 1][t])
        for t = 1 to k do
            if t < j − 1 or t > j + 1 then
                maxVal = max(maxVal, val[i][j] + DPtable[i − 1][t])
        DPtable[i][j] = maxVal
maxVal = 0
for t = 1 to k do
    maxVal = max(maxVal, DPtable[n][t])
return maxVal
```

(d) Dynamic Programming with $O(k)$ extra memory.

Solution: Just record the current and the last column (in total two at any moment). Changes are in blue. If i is odd, than we use the column 1 in the DP table to store values for it, otherwise the column 0; hence, $i\%2$ gives us the column for i (where $i\%2$ means i modulo 2).

```
DPtable[0..1][1..k] = new array filled with 0s
for i = 1 to n do
    for j = 1 to k do
        maxVal = 0
        for t = 1 to k do
            maxVal = max(maxVal, DPtable[(i − 1)%2][t])
        for t = 1 to k do
            if t < j − 1 or t > j + 1 then
                maxVal = max(maxVal, val[i][j] + DPtable[(i − 1)%2][t])
        DPtable[i%2][j] = maxVal
maxVal = 0
for t = 1 to k do
    maxVal = max(maxVal, DPtable[n%2][t])
return maxVal
```

(e) Modify the algorithm of task (c) such that it outputs the chosen grid cells as a binary array $chosen[1..n][1..k]$ where $chosen[i][k] = true$ if the grid cell in the i-th column and j-th row has been chosen.

Solution: When finished computing the DP table go back through the table and write into *chosen* which element you choose.

Input: $maxVal$ and $DPtable[0..n][1..k]$ as computed by our dynamic program.
Output: $chosen[1..n][1..k]$

```
// To make our algorithm simpler, we assume that we have a dummy element in the
   non-existing column n+1 at the non-existing row k+1 (that does not touch
   any other cells of the actual input table.  Starting from this dummy element,
   the current candidate has coordinates (i,j) and we want to find out whether to
   add it or not.
chosen[1..n+1][1..k] = new boolean array; initially all entries set to false; // for
   technical reasons n+1 columns
i = n + 1
j = k + 1
valij = 0// Later, when i ≤ n, it will be equal to val[i][j].
while i ≥ 1 do
    t = 1
    notFound = true     // Is set to true as long as we don't know from which cell
     and in what case we reached (i,j)

    // Case 1:  We don't take element (i,j)
    while t ≤ k and notFound do
        if maxVal == DPtable[i − 1][t] then
            notFound = false
            j = t
        t = t + 1
    t = 1
    // Case 2:  We take element (i,j).  This case must hold if notFound is still
       set to false.
    while notFound do
        if (t < j − 1 or t > j + 1) and maxVal == valij + DPtable[i − 1][t] then
            notFound = false
            chosen[i][j] = true
            j = t
        t = t + 1
    valij = val[i][j]
    maxVal = DPtable[i − 1][j]
    i = i − 1
Remove last (technical) column from chosen
return chosen
```

If you finished all subtasks, take a look Task 4.4!

Task 4.2: *Bonus Task - Heap Sort* (2 points)

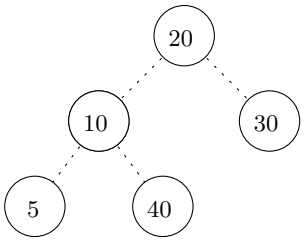
(2 activity points for presenting or writing.)

Run HeapSort on the array $a = \langle 20, 10, 30, 5, 40 \rangle$. Draw the array a after each single change as a binary complete tree.

Solution: First, the array is transformed into a heap. Here, we do it by adding the elements one after the other. One could also call the method `BuildHeap()` from the literature which would result in a different heap tree. After obtained the heap, we remove the maximum element until the heap is empty.

For clarity, changes are marked in yellow and edges that do not belong to the heap are dashed.

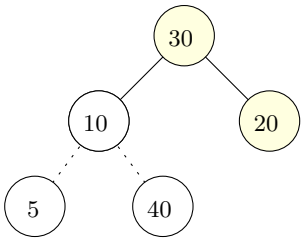
Beginning:



(1)

No change after `Add(20)` and `Add(10)`

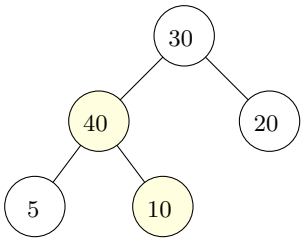
Change after `Add(30)`



(2)

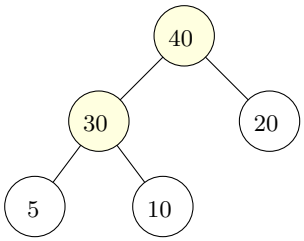
No change after `Add(5)`

First change during `Add(40)`



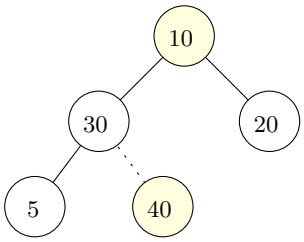
(3)

Secong (last) change during `Add(40)`



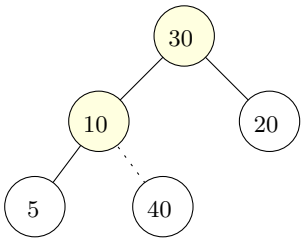
(4)

First change during first `RemoveMax()`



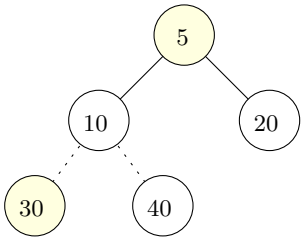
(5)

Second (last) change during first `RemoveMax()`



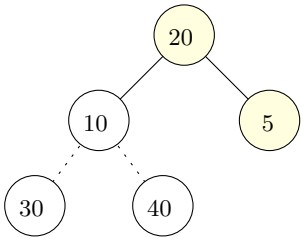
(6)

First change during second `RemoveMax()`



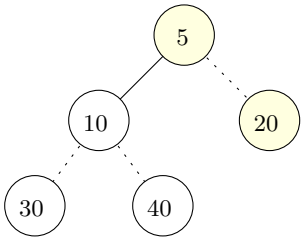
(7)

Second (last) change during second `RemoveMax()`



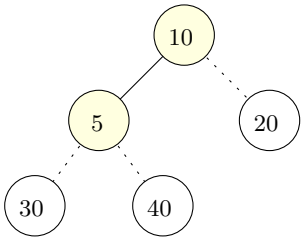
(8)

First change during third `RemoveMax()`



(9)

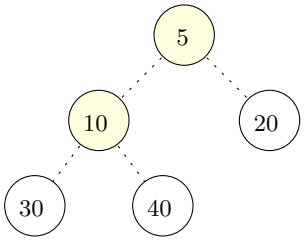
Second (last) change during third `RemoveMax()`



(10)

Change after forth `RemoveMax()`

(No change after fifth `RemoveMax()`)



(11)

Task 4.3: *Bonus Task - QuickSort: Polish Flag* (3 points)

Second May is Polish National Flag Day. For this occasion, you want to program a robot to arrange red and white balls to form a Polish flag. In particular, you are given $2n$ bins numbered from 1 to $2n$ where each bin contains exactly one ball. In total there are n red and n white balls. Your goal is that the first n bins contain red balls, the remaining ones white balls. Your robot can only do the following operations:

- isRed(i) : the robot looks at the i-th bin and returns true if it contains a red ball, and false otherwise.
- swap(i,j): the robot exchanges the balls of the i-th and j-th bin.

Furthermore, your robot has only a small memory of constant size.

(a) How is this problem related to QuickSort?

Solution: The Polish flag problem is equivalent to the subproblem of Quicksort where elements smaller than the pivot (let's call them red) have to be moved to the left side of the pivot and elements larger than the pivot (let's call them white) have to be moved to the right side of the pivot. For instance, if we color our elements like this, then any algorithm for the Polish flag problem will move the elements correctly. At the end, we just have to reinsert the pivot element in between them. On the other hand, the QuickSort subroutine would solve the Polish flag problem directly if we define the order *red is smaller than white* (or, alternatively, give all red balls the value 0 and all white balls the value 1).

Hence, if we can solve the Polish flag problem using only *a small memory of constant size* as required, then the Quicksort algorithm for the subproblem, and in fact, for the overall sorting problem, can also be implemented by using only constant additional memory. In the next subtask, we observe that this is in fact possible.

(b) Program the robot to arrange a Polish flag within $O(n)$ operations.

Solution:

$left = 1$
$right = n$
// The invariant: Anything to the left of $left$ is red, and anything to the right of $right$ is right.
while $left < right$ **do**
  // The exclamaftion mark in !isRed($left$) negates the output. That is, the following if statement should be read as:
  // If isRed($left$) == false **and** isRed($right$) == true
  if !isRed($left$) **and** isRed($right$) **then**
    swap($left$,$right$)
  if isRed($left$) **then**
    $left = left + 1$
  if !isRed($right$) **then**
    $right = right - 1$

The algorithm terminates as, in each repetition of the while loop, we increase $left$ or decrease $right$. Thus, we repeat the for loop at most $n-1$ times and therefore the running time is linear in n.

The invariant is correct as we increase or decrease $left$ and $right$ only if they are red or white, respectively. Consequently, the output is correct.

Task 4.4: *Bonus Task - Dynamic Programming* (2 points)

Given an array containing possibly negative values, select a block of maximum value.

Solution: ***The solution itself is simple (see any of the four algorithms below). In the text below, I give you some intuition on how to come up with the final solution.***

Let $a[1..n]$ be the input array. The problem is to find a *heaviest* block, that is, a subarray $a[i..j]$, of maximum total value $\sum_{k=i}^{j} a[k]$.

How to solve such a problem using recursion or dynamic programming? Let us take a look at what choices we have:

- If we choose to take some element $a[j]$ into the block, then (i) either $a[j]$ is the first element of the block—meaning that we cannot take any element $a[k]$ with $k < j$ to the block—, or (ii) $a[j]$ is some middle element meaning that we have to choose $a[j-1]$ also to the block and to again consider the two options (i) and (ii) for the element $a[j-1]$ (hence, we have a recursion here!).

  By this reasoning we can compute the maximum total value of a block within the subarray $a[1..j]$ assuming that the block ends in $a[j]$; let us call this value $DPtake[j]$. In case (i), the block also starts in $a[j]$, hence, consists only of $a[j]$ and has therefore value just $a[j]$. In case (ii), the block consists of $a[j]$ and—this is now the important observation—of a heaviest block that ends in $a[j-1]$, that is, of value $DPtake[j-1]$. Hence, to compute $DPtake[j]$, we have to take the maximum over both cases (i) and (ii):

  $$DPtake[j] = \max(a[j], a[j] + DPtake[j-1]) \ .$$

  Now, the question arises: Couldn't we just write $DPtake[j] = a[j] + DPtake[j-1]$? The answer is no as $DPtake[j-1]$ might be negative—recall that we have negative numbers!

- OK, but what if we don't choose to take the element $a[j]$ into the block? Then we have to find a block to the left and a block to the right of j of total maximum value. Let $DP[k]$ be the value of a heaviest block in the subarray $a[1..k]$ independently of whether $a[k]$ has been taken). Then the heaviest block to the left of j has value $DP[j-1]$. Let us skip the right side for now and ask, how to compute $DP[j]$.

Above, we have seen two choices, to take or not to take $a[j]$. What is the better choice? If we are interested in a heaviest block in the subarray $a[1..j]$ (whose value is defined to be $DP[j]$), then the answer is to take the maximum over both choices:

$$DP[j] = \max(DPtake[j], DP[j-1]) \ .$$

But what about the right side, that is, the subarray $a[j+1..n]$? Well, at the end of the day, we are interested in computing $DP[n]$ which is in fact the answer to the overall problem: the total value of a heaviest block in $a[1..n]$! Therefore, if we are able to compute $DP[n]$ without considering the right sides for particular values for j, we are totally fine!

Since $DP[j]$ depends only on $DP[j–1]$ and $DPtake[j]$, whereas $DPtake[j]$ depends only on $DPtake[j–1]$ and $a[j]$, we can compute $DP[n]$ by computing the respective values from left to right (for increasing j) or in a recursive manner.

Our discussion leads directly to the following dynamic programming algorithm:

```
Block_DP(a)
    DP[0..n], DPtake[0..n] = new arrays filled with zeros
    for j = 1 to n do
        DPtake[j] = max(a[j], a[j] + DPtake[j − 1])
        DP[j] = max(DPtake[j], DP[j − 1])
    return DP[n]
```

Since the for loop repeats only n times and has only operations that take constant time, the total running time is $\Theta(n)$. The space complexity is $\Theta(n)$ due to the two arrays DP and $DPtake$.

However, we can even reduce the space complexity to $\Theta(1)$ in the following way:

```
Block_DP_lessspace(a)
    DP[0..1], DPtake[0..1] = new arrays filled with zeros
    for j = 1 to n do
        DPtake[j%2] = max(a[j], a[j] + DPtake[(j − 1)%2]
        DP[j%2] = max(DPtake[j%2], DP[(j − 1])%2)
    return DP[n%2]
```

To make the picture complete, we also discuss the recursive solutions. We begin by solving the problem recursively via backtracking. We have one method corresponding to DP and another corresponding to $DPtake$. We call the algorithm via `Block_Back(a,n)`.

```
Block_Back(a,j)
    if j == 0 then
        return 0
    return max(Block_Back(a,j-1), Blocktake_Back(a,j))
Blocktake_Back(a,j)
    if j == 0 then
        return 0
    return max(a[j], a[j] + Blocktake_Back(a,j-1))
```

The algorithm above has a high running time but—interestingly—it is not too horrible. Observe that `Block_Back(a,j)` is called exactly once for every value of j (and has constant time per call—not counting the time within the recursive calls). Thus, the total running time alone for all calls of `Block_Back()` is $\Theta(n)$. What about `Blocktake_Back(a,j)`? For each j, it is called $n-j+1$ times: once by `Block_Back(a,j)` and $n - j$ times by `Blocktake_Back(a,j+1)`. The total running time is therefore $\Theta(\sum_{j=1}^{n}(n - j + 1))$ which can be shown to be $\Theta(n^2)$. Hence, not so bad.

But, we can do it better using memoization and improve the running time down to $\Theta(n)$. We use two dictionaries, called DP and $DPtake$.

```
DP, DPtake = {}
Block_Mem(a,j)
    if j == 0 then
        return 0
    if j ∉ DP then
        DP[j] = max(Block_Mem(a,j-1), Blocktake_Mem(a,j))
    return DP[j]
Blocktake_Mem(a,j)
    if j == 0 then
        return 0
    if j ∉ DPtake then
        DPtake[j] = max(a[j], a[j] + Blocktake_Mem(a,j-1))
    return DPtake[j]
```