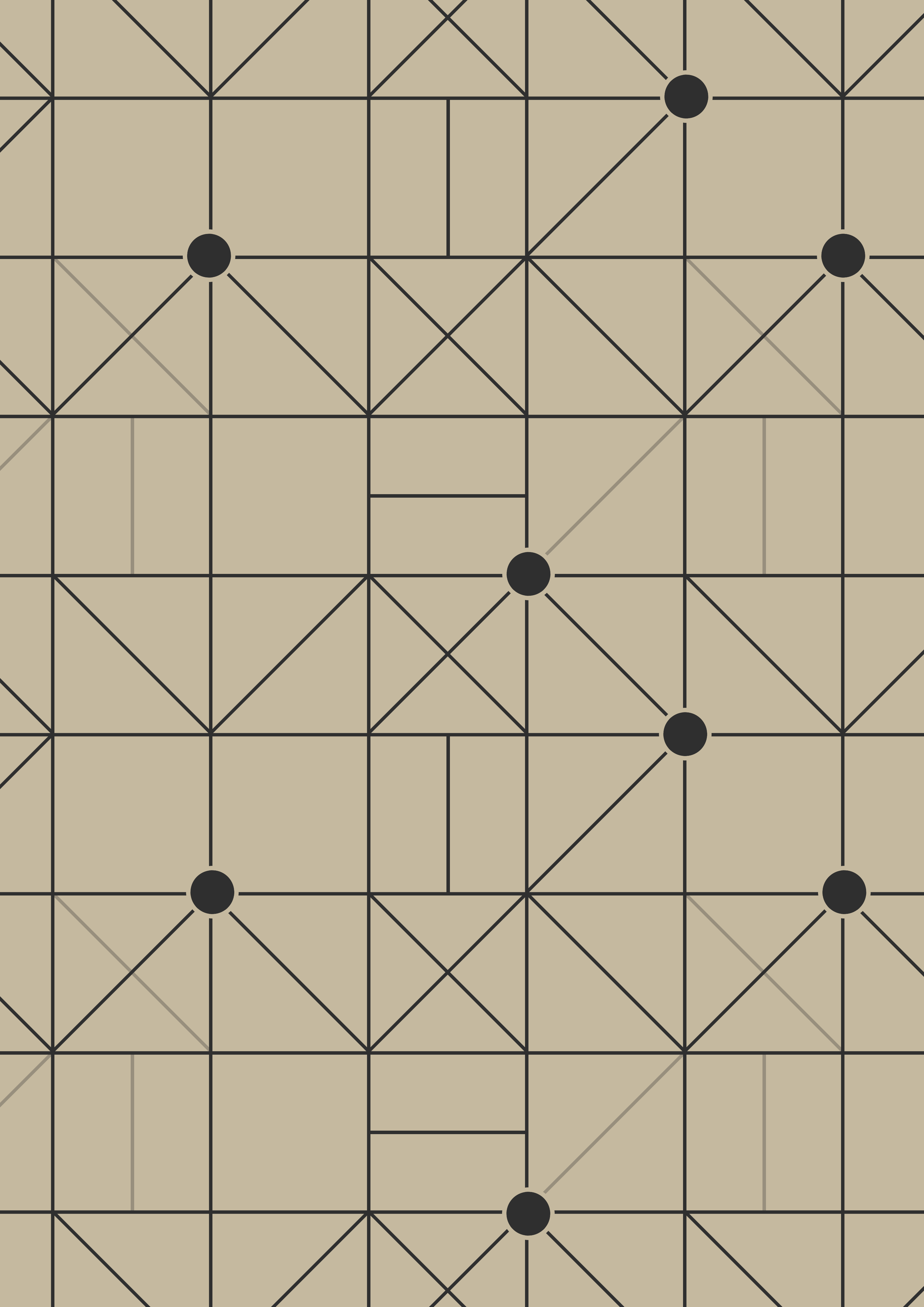
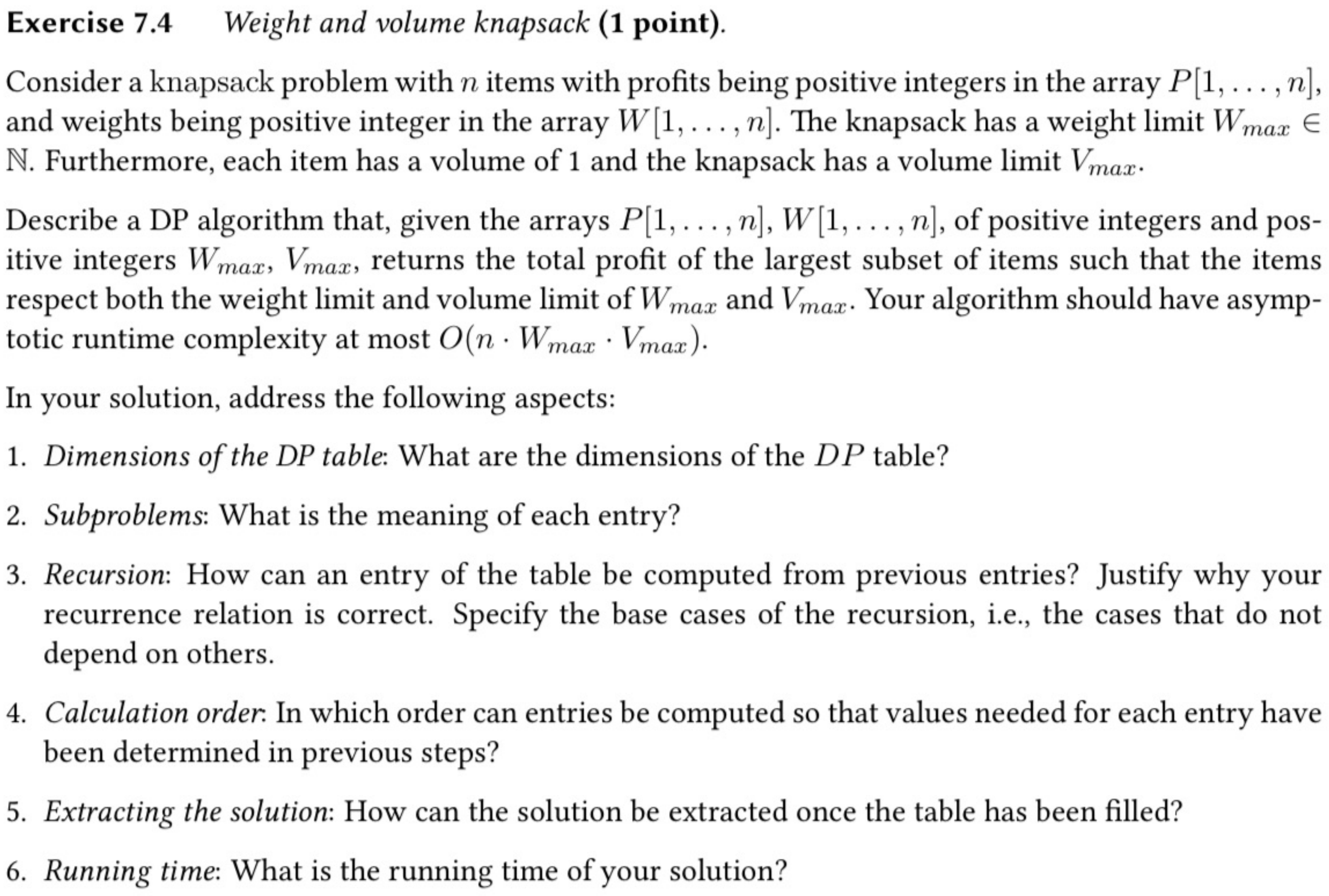


ALGORITHMEN & DATENSTRUKTUREN - ÜBUNGSSTUNDE 8



Exercise 7.4 *Weight and volume knapsack* **(1 point)**.

Consider a knapsack problem with n items with profits being positive integers in the array $P[1, \ldots, n]$, and weights being positive integer in the array $W[1, \ldots, n]$. The knapsack has a weight limit $W_{max} \in \mathbb{N}$. Furthermore, each item has a volume of 1 and the knapsack has a volume limit V_{max}.

Describe a DP algorithm that, given the arrays $P[1, \ldots, n]$, $W[1, \ldots, n]$, of positive integers and positive integers W_{max}, V_{max}, returns the total profit of the largest subset of items such that the items respect both the weight limit and volume limit of W_{max} and V_{max}. Your algorithm should have asymptotic runtime complexity at most $O(n \cdot W_{max} \cdot V_{max})$.

In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

-> Problem ist ahnlich zu knapsack

1. $DP[0..n][0..W_{max}][0..V_{max}]$

2. $DP[i][j][k]$ = Maximaler profit von items 1..i mit maximalem Gewicht j & maximalem Volumen k

3. $DP[0][j][k] = 0$ für $0 \leq j \leq W_{max}$, $0 \leq k \leq V_{max}$

   $DP[i][j][k] = \max\{DP[i-1][j][k], DP[i-1][j-W[i]][k-1] + P[i]\}$

   wir nehmen i-tes item nicht

   wir nehmen i-tes item

   funktioniert weil Array laut Aufgabenstellung 1 geindexed

   out of bounds hier beachten !

   für $1 \leq i \leq n$, $1 \leq j \leq W_{max}$, $1 \leq k \leq V_{max}$

4. erst in aufsteigendem i, dann j, dann k

5. Lösung in $DP[n][W_{max}][V_{max}]$ ← hier wieder auf outofbounds/indexing aufpassen

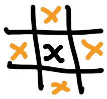
6. $O(nW_{max}V_{max})$, da wir $(n+1)(W_{max}+1)(V_{max}+1)$ Einträge in $O(1)$ berechne und die Lösung in $O(1)$ extrahieren.

Exercise 7.5 *Zebra arrays* (**1 point**).

Zebra array is square

A square two-dimensional array $Z[1 \ldots k][1 \ldots k]$ with entries in $\{0, 1\}$ is called a *zebra array* if no two adjacent entries of Z are equal. We say two distinct entries $Z[i_1][j_1]$ and $Z[i_2][j_2]$ in Z are adjacent if

- $i_1 = i_2$ and $|j_1 - j_2| \leq 1$; or
- $|i_1 - i_2| \leq 1$ and $j_1 = j_2$.



Aufzeichnen, ausser absolut klar beim draufschauen

Describe a DP algorithm that, given a two-dimensional array $A[1 \ldots n][1 \ldots m]$ with entries in $\{0, 1\}$, outputs the size of a largest zebra array contained in A. That is, the largest k such that, for some $1 \leq i \leq n - k + 1, 1 \leq j \leq m - k + 1$, the array $A[i \ldots i + k - 1][j \ldots j + k - 1]$ is a zebra array. Your algorithm should have asymptotic runtime complexity at most $O(nm)$.

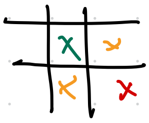
In your solution, address the following aspects:

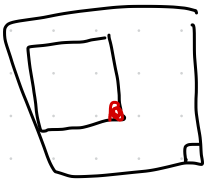
1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Hint: Use a DP table $B[1 \ldots n][1 \ldots m]$. The meaning of entry $B[i][j]$ is

$$B[i][j] = \text{size of the largest zebra array in } A \text{ whose bottom-right entry is } A[i][j].$$

Hint: Your recursion to compute $B[i][j]$ should involve the entries $B[i][j-1]$, $B[i-1][j]$, $B[i-1][j-1]$ and also the entries $A[i][j]$, $A[i][j-1]$, $A[i-1][j]$, $A[i-1][j-1]$.





A ist 1 geindexed

1. DP[1..n][1..m]

2. DP[i][j] = grösse von grösstem zebra array mit unterer-rechter-ecke A[i][j]

3. DP[i][1] = 1 , $1 \leq i \leq n$
   DP[1][j] = 1 , $1 \leq j \leq n$
   DP[i][j] = 1 , if A[i][j] = A[i][j-1] ‖ A[i][j] = A[i-1][j] ‖ A[i][j] ≠ A[i-1][j-1]
   DP[i][j] = 1 + min{DP[i][j-1], DP[i-1][j], DP[i-1][j-1]}
   wir nehmen nächstes item als ecke und nehmen min vom Rest, weil die es kaputt machen könnten

4. von links nach rechts & von oben nach unten

   for i = 1... n
     for j = 1...m

5. maximum aller Einträge

6. O(nm) da wir n·m Einträge in O(1) berechnen und das endresultat in O(1) extrahieren.

Exercise 7.3 *Road trip.*

You are planning a road trip for your summer holidays. You want to start from city C_0, and follow the only road that goes to city C_n from there. On this road from C_0 to C_n, there are $n-1$ other cities $C_1, \ldots, C_{n-1}$ that you would be interested in visiting (all cities $C_1, \ldots, C_{n-1}$ are on the road from C_0 to C_n). For each $0 \leq i \leq n$, the city C_i is at kilometer k_i of the road for some given $0 = k_0 < k_1 < \ldots < k_{n-1} < k_n$.

You want to decide in which cities among $C_1, \ldots, C_{n-1}$ you will make an additional stop (you will stop in C_0 and C_n anyway). However, you do not want to drive more than d kilometers without making a stop in some city, for some given value $d > 0$ (we assume that $k_i < k_{i-1} + d$ for all $i \in [n]$ so that this is satisfiable), and you also don't want to travel backwards (so from some city C_i you can only go forward to cities C_j with $j > i$).

(a) Provide a *dynamic programming* algorithm that computes the number of possible routes from C_0 to C_n that satisfy these conditions, i.e., the number of allowed subsets of stop-cities. Your algorithm should have $O(n^2)$ runtime.

In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps? Describe the calculation order in pseudocode.
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

1. $n+1$ ← counterintuitive bc. of n^2 runtime

2. $Dp[i]$ = #routes from C_0 to C_i (stopping at C_i)

3. $Dp[0] = 1$; $\forall i > 0$, $Dp[i] = \sum_{\substack{0 \leq j < i \\ k_i \leq k_j + d}} Dp[j]$ we look through all entries and check whether we can attach C_i after C_j. this is possible, exactly when $k_i \leq k_j + d$

4. from 0 to n

5. $DP[n]$

6. $\sum_{i=0}^{n} \sum_{j=0}^{i-1} 1 = O(n^2)$

(b) If you know that $k_i > k_{i-1} + d/10$ for every $i \in [n]$, how can you turn the above algorithm into a linear time algorithm (i.e., an algorithm that has $O(n)$ runtime) ?

for each entry we only need to look back 10 entries before we have $k_i > k_j + d$. once reached that bound it can only get bigger so elements further back do not need to be considered

Exercise 7.5 *Searching a doubly sorted array.*

You are given a 2D array of numbers $A[1 \dots n][1 \dots n]$. The goal is to understand the search problem: given x, determine whether x occurs in A, and if so, find it. Without any structure on A, this would take $O(n^2)$ time since we would have to check every entry. However, by assuming more structure on A, we can hope to bring this runtime down. All sorting in this problem will be in ascending order.

Suppose A satisfies the following property: Every row is sorted left-to-right.

(a) Design a search algorithm that runs in time $O(n \log(n))$.

Now suppose A satisfies a stronger property: Every row is sorted left-to-right and every column is sorted top-to-bottom.

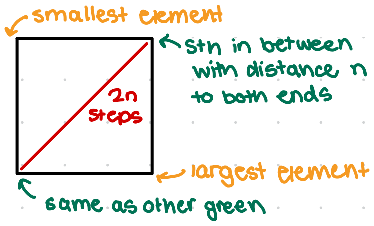
(b) Design a search algorithm that runs in time $O(n)$.

Hint: *Say we want to search for an element x. Suppose that $A[i, j] < x$, for some $1 \leq i, j \leq n$, then what can you say about entries $A[k, j]$ for $1 \leq k < i$ and $A[i, \ell]$ for $1 \leq \ell < j$?*

(c)* Argue that this bound is tight. Namely, that any search algorithm must run in time $\Omega(n)$.

Hint: *Consider an assignment of numbers to the off-diagonal entries. Off-diagonal entries are entries at $A[i, n - i + 1]$ for $1 \leq i \leq n$. When do we have a 2D array of numbers which satisfies that off-diagonal assignment and the stronger property?*

b)



Start at the top green element. compare to x if $x < A[i][j]$ goto $A[i-1][j]$ if $x > A[i][j]$ go to $A[i][j+1]$. if $x = A[i][j]$ return.

so we walk trough the table step by Step. In total we can take at most 2n steps this is reached when the element is in the opposite corner (depicted in red)

Exercise 7.4 *String counting* **(1 point).**

Given a binary string $S \in \{0,1\}^n$ of length n, let $f(S)$ be the number of times "11" occurs in the string, i.e. the number of times a 1 is followed by another 1. In particular, the occurrences do not need to be disjoint. For example $f(\text{"}\underline{1}\underline{1}101\underline{1}1\text{"}) = 3$ because the string contains three 1 (underlined) that are followed by another 1. Given n and k, the goal is to count the number of binary strings S of length n with $f(S) = k$.

Describe a DP algorithm that, given positive integers n and k with $k < n$, reports the required number. Your solution should have complexity at most $O(nk)$.

In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps? Describe the calculation order in pseudocode.
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Hint: *Define a three dimensional DP table* $DP[1 \ldots n][0 \ldots k][0 \ldots 1]$.

Hint: *The entry* $DP[i][j][l]$ *counts the number of strings of length* i *with* j *occurrences of* "11" *that end in* l *(where* $1 \le i \le n$, $0 \le j \le k$ *and* $0 \le l \le 1$*).*

1. $n \times (k+1) \times 2$

2. dp[i][j][l] = # strings of length i with j times "11" that end in l

3. Base cases: dp[1][0][l] = 1 ; dp[1][j][l] = 0, $\forall j > 0$

recursion: dp[i][j][0] = dp[i-1][j][0] + dp[i-1][j][1]

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

4. calculation order i from 1 to n rest doesn't matter

```
for i=1...n
  for j=0...k
    for l=0..1
```

5. dp[n][k][0] + dp[n][k][1]

notation of master solution this is weird bc. then we have 1 part 0-indexed and 1 part 1-indexed. in practice you have to take care of that.

6. O(nk) entries, each processed in O(1).