Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Markus Püschel, David Steurer
Gleb Novikov, Tommaso d'Orsi, Ulysse Schaller, Rajai Nasser

08. November 2021

Algorithms & Data Structures — Exercise sheet 7 — HS 21

Exercise Class (Room & TA): ______________________

Submitted by: ______________________

Peer Feedback by: ______________________

Points: _____

Submission: On Monday, 15 November 2021, hand in your solution to your TA *before* the exercise class starts. Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

Exercise 7.1 *Subset sum for general integers* **(1 point)**.

Let $a_1, \dots, a_n, t$ be $n+1$ integers in $\mathbb{Z}$. We would like to check whether there is a subset $I \subseteq \{1, \dots, n\}$ such that $\sum_{i \in I} a_i = t$. Here, we adopt the convention that if I is empty, then $\sum_{i \in I} a_i = 0$.

We have seen in class that if $a_1, \dots, a_n, t$ are positive, then we can solve this problem in $O(nt)$ time using dynamic programming. In this exercise, we would like to handle the case where some of the integers $a_1, \dots, a_n, t$ could be negative or zero.

Provide a *dynamic programming* algorithm that solves the subset sum problem for general integers. The algorithm should have $O\left(n \cdot \sum_{i=1}^{n} |a_i|\right)$ runtime.

Hint: *The DP table is two-dimensional, and its size is $(n+1) \times (1 + \sum_{i=1}^{n} |a_i|)$. Furthermore, for $i > 0$, the entry $DP[i][j]$ can be computed from $DP[i-1][j]$ and $DP[i-1][j - a_i]$.*

Address the following aspects in your solution:

1. *Definition of the DP table*: What is the meaning of each entry?
2. *Computation of an entry*: How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries that do not depend on others.
3. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
4. *Extracting the solution*: How can the final solution be extracted once the table has been filled?
5. *Running time*: What is the running time of your solution?

Solution: Let $N := \sum_{a_i<0} |a_i|$ and $P := \sum_{a_i>0} a_i$. We can compute N and P in $O(n)$ time. Note that

$$N + P = \sum_{i=1}^{n} |a_i|.$$

It is easy to see that for every $I \subseteq \{1, \ldots, n\}$, we have $-N \leq \sum_{i \in I} a_i \leq P$. Therefore, if $t < -N$ or $t > P$, we can immediately say that the answer is no. In order to handle the case $-N \leq t \leq P$, we need dynamic programming.

Definition of the DP table: For $0 \leq i \leq n$ and $0 \leq j \leq N + P$, the entry $DP[i][j]$ is a boolean value indicating whether there is a subset $I \subseteq \{1, \ldots, i\}$ such that $\sum_{k \in I} a_k = j - N$. Here, we adopt the convention that for $i = 0$, we have $\{1, \ldots, i\} = \varnothing$.

Computation of an entry: Initialize

- $DP[0][N] = \text{TRUE}$. This is because $\sum_{k \in \varnothing} a_k = 0 = N - N$.
- $DP[0][j] = \text{FALSE}$, for every $j \neq N$.

Now for $i \geq 1$ and $0 \leq j \leq N + P$, we can compute $DP[i][j]$ using the formula

$$DP[i][j] = DP[i-1][j] \text{ OR } (j \geq a_i \text{ AND } DP[i-1][j-a_i]). \tag{1}$$

Calculation order: We can calculate the entries of DP in order of increasing i. For fixed i, we can compute the entries $(DP[i][j])_{0 \leq j \leq N+P}$ in any order of j.

Extracting the solution: All we have to do is read the value at $DP[n][t + N]$.

Running time: The entry $DP[i][j]$ can be computed in $O(1)$ time. Therefore, the total runtime is

$$\sum_{i=0}^{n} \sum_{j=0}^{N+P} O(1) = O\big((n+1) \cdot (N+P+1)\big) = O(n \cdot (N+P)) = O\left(n \cdot \sum_{i=1}^{n} |a_i|\right).$$

Exercise 7.2 *Longest Snake.*

You are given a game-board consisting of hexagonal fields $F_1, \ldots, F_n$. The fields contain natural numbers $v_1, \ldots, v_n \in \mathbb{N}$. Two fields are neighbors if they share a border. We call a sequence of fields $(F_{i_1}, \ldots, F_{i_k})$ a *snake* of length k if, for $j \in \{1, \ldots, k-1\}$, F_{i_j} and $F_{i_{j+1}}$ are neighbors and their values satisfy $v_{i_{j+1}} = v_{i_j} + 1$. Figure 1 illustrates an example game board in which we highlighted the longest snake.

For simplicity you can assume that F_i are represented by their indices. Also you may assume that you know the neighbors of each field. That is, to obtain the neighbors of a field F_i you may call $\mathcal{N}(F_i)$, which will return the set of the neighbors of F_i. Each call of $\mathcal{N}$ takes unit time.

a) Provide a *dynamic programming* algorithm that, given a game-board $F_1, \ldots, F_n$, computes the length of the longest snake.

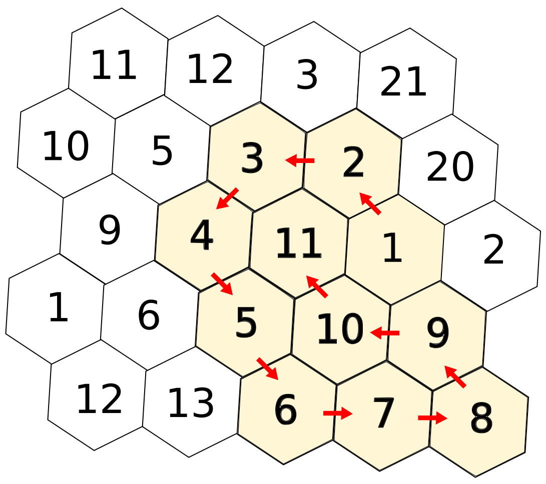

Figure 1: Example of a longest snake.

Hint: *Your algorithm should solve this problem using $\mathcal{O}(n \log n)$ time, where n is the number of hexagonal fields.*

Address the following aspects in your solution:

1. *Dimensions of the DP table*: What are the dimensions of the table $DP[\dots]$?
2. *Definition of the DP table*: What is the meaning of each entry?
3. *Computation of an entry*: How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries 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 final solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Solution:

Dimensions of the DP table: The DP table is linear, its size is n.

Definition of the DP table: $DP[i]$ is the length of the longest snake with head F_i (that is, the length of the longest snake of the form $(F_{j_1}, \dots, F_{j_{m-1}}, F_i)$).

Computation of an entry:

$$DP[i] = 1 + \max_{\substack{F_j \in \mathcal{N}(F_i) \\ v_j = v_i - 1}} DP[j].$$

That is, we look at those neighbors of F_i that have values v_j smaller than v_i exactly by 1, and choose the maximal value in the DP table among them. If there are no such neighbors, we define $\max$ in this formula to be 0.

Calculation order: We first sort the hexagons by their values. Then we fill the table in ascending order, that is, $i_1, \dots, i_n$ such that $v_{i_j} \leq v_{i_{j+1}}$ for all $j = 1, \dots n - 1$.

Extracting the solution: The output is $\max_{1 \leq i \leq n} DP[i]$.

Running time: We compute the order in time $\mathcal{O}(n \log n)$ by sorting $v_1, \dots, v_n$. Then each entry can be computed in time $\mathcal{O}(1)$ and finally we compute the output in time $\mathcal{O}(n)$. So the running time of the algorithm is $O(n \log n)$.

b) Provide an algorithm that takes as input $F_1, \dots F_n$ and a DP table from part a) and outputs the longest snake. If there are more than one longest snake, your algorithm can output any of them. State the running time of your algorithm in Θ-notation in terms of n.

Solution: At the beginning we find a head of a snake that is some F_{j_1} such that $DP[j_1] = \max\limits_{1 \le i \le n} DP[i]$. If $DP[j_1] \neq 1$, we look at its neigbours and find some F_{j_2} such that $DP[j_2] = DP[j_1] - 1$. If $DP[j_2] \neq 1$, then among neighbors of F_{j_2} we find some F_{j_3} such that $DP[j_3] = DP[j_2] - 1$ and so on. We stop when $DP[j_m] = 1$ (where m is exactly the length of the longest snake). Then we output the snake $(F_{j_1}, \dots, F_{j_m})$.

The running time of this algorithm is $\Theta(n)$, since we use $\Theta(n)$ operations to find F_{j_1} and we need $\Theta(1)$ time to find each F_{j_k} for $1 < k \le m \le n$ and $\Theta(m)$ time to output the snake.
Remark 1. An alternative solution would be to store the predecessor in a longest snake with head F_i directly in $DP[i]$ (in addition to the length of this longest snake), and store $\emptyset$ if the length of the longest snake is just 1. Then, in order to recover a longest snake, we simply need to find a head of a snake that has maximal length and then follow the sequence of predecessors until we reach an entry $DP[i]$ that has $\emptyset$ as predecessor.

c)* Find a linear time algorithm that finds the longest snake. That is, provide an $\mathcal{O}(n)$ time algorithm that, given a game-board $F_1, \dots, F_n$, outputs the longest snake (if there are more than one longest snake, your algorithm can output any of them).

Solution: We can use recursion with memorization. Similar to part a), we will fill an array $S[1, \dots, n]$ of lengths of longest snakes, that is, $S[i]$ is the length of the longest snake with head F_i. Consider the following pseudocode:

Algorithm 1 Fill-lengths($v_1, \dots, v_n$)

```
$S[1], \dots, S[n] \leftarrow 0, \dots, 0$
**for** $i = 1, \dots, n$ **do**
    **if** $S[i] = 0$ **then**
        Move-to-tails($i, S, v_1, \dots, v_n$)
**return** $S$
```

where the procedure Move-to-tails($i, S, v_1, \dots, v_n$) is:

Algorithm 2 Move-to-tails($i, S, v_1, \dots, v_n$)

```
**for** $F_j \in \mathcal{N}(F_i)$ **do**
    **if** $v_j = v_i - 1$ **and** $S[j] = 0$ **then**
        Move-to-tails($j, S, v_1, \dots, v_n$)
$S[i] = 1 + \max\limits_{\substack{F_j \in \mathcal{N}(F_i) \\ v_j = v_i - 1}} S[j]$
```

As in part a), we assume that max over the empty set is 0. Let us show why this procedure is correct. First, since the algorithm Move-to-tails is recursive, we have to check that it actually finishes. Move-to-tails($i, S, v_1, \dots, v_n$) is calling Move-to-tails only for indices j with $v_j < v_i$, and therefore an easy induction on v_j shows that the algorithm will always terminate. We now show the correctness of Move-to-tails$(i, S, v_1, \dots, v_n)$ by induction on v_i.

Base case $v_i = 1$: If $v_i = 1$, then there is no j such that $v_j = v_i - 1$. Therefore, the $\max$ in Move-to-tails$(i, S, v_1, \dots, v_n)$ is empty, so $S[i]$ is set to 1, which is indeed the length of a longest snake with head F_i when $v_i = 1$.

Induction hypothesis: After calling Move-to-tails$(i, S, v_1, \dots, v_n)$ with $v_i = k$, the value of $S[i]$ contains the length of the longest snake with head F_i.

Induction step $k \to k+1$: Let i be an index with $v_i = k + 1$. Then for any $F_j \in \mathcal{N}(F_i)$ such that $v_j = v_i - 1$, we have $v_j = k$, so by the induction hypothesis after calling Move-to-tails$(j, S, v_1, \dots, v_n)$ the value of $S[j]$ contains the length of the longest snake with head F_j. Therefore, after setting

$$S[i] = 1 + \max_{\substack{F_j \in \mathcal{N}(F_i) \\ v_j = v_i - 1}} S[j],$$

the value of $S[i]$ indeed contains the length of the longest snake with head F_i.

After we fill S, we can use the same algorithm as in part b) to find a longest snake (we should replace DP by S in the description of that algorithm).

For the runtime, we will show that for each $i \in \{1, \dots, n\}$ we call Move-to-tails$(i, S, v_1, \dots, v_n)$ exactly once. Indeed, it is called only when $S[i] = 0$, and after the first call of Move-to-tails$(i, S, v_1, \dots, v_n)$ has terminated, we have $S[i] > 0$ by the invariant for the rest of the algorithm. So Move-to-tails$(i, S, v_1, \dots, v_n)$ will not be called a second time after the first call has terminated. While the first call of Move-to-tails$(i, S, v_1, \dots, v_n)$ is running, Move-to-tails is only called for indices j with $v_j < v_i$, which follows from a very simple induction. So Move-to-tails$(i, S, v_1, \dots, v_n)$ is also not called a second time while the first call is still running. So we have shown that Move-to-tails$(i, S, v_1, \dots, v_n)$ is called exactly once for each i. Therefore, the running time is linear in n.

The technique that we used here is closely related to depth-first search and topological ordering of a graph. These topics will be studied later in this course.

Exercise 7.3 *Road trip* **(1 point).**

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 $C_1, \dots, C_{n-1}$ that you would be interested in visiting (all cities $C_1, \dots, C_{n-1}$ are right on the road from C_0 to C_n). For each $0 \le i \le n$, the city C_i is at kilometer k_i of the road for some given $0 = k_0 < k_1 < \dots < k_{n-1} < k_n$.

You want to decide in which cities among $C_1, \dots, C_{n-1}$ 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 satisfies these conditions, i.e., the number of allowed subsets of stop-cities. In order to get full points, your algorithm should have $O(n^2)$ runtime.

Address the following aspects in your solution:

1. *Dimensions of the DP table*: What are the dimensions of the table $DP[\ldots]$?
2. *Definition of the DP table*: What is the meaning of each entry?
3. *Computation of an entry*: How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries 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 final solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Solution:

Dimensions of the DP table: The DP table is linear, and its size is $n + 1$.

Definition of the DP table: $DP[i]$ is the number of possible routes from C_0 to C_i (which stop at C_i).

Computation of an entry: Initialize $DP[0] = 1$.

For every $i > 0$, we can compute $DP[i]$ using the formula

$$DP[i] = \sum_{\substack{0 \le j < i \\ k_i \le k_j + d}} DP[j]. \tag{2}$$

For completeness, we now show the correctness of this formula (you do not have to do it unless we explicitly ask for it in the task).

For a given route from C_0 to C_i, let j be the index of the last city where you stop before C_i. Since you do not go backwards we have $j < i$, and since you do not travel more than d kilometers between two stops we also have $k_i \le k_j + d$. The total number of routes from C_0 to C_i whose last stop (before C_i) is C_j is simply the number of routes from C_0 to C_j, which is $DP[j]$. Therefore, to get the total number of routes from C_0 to C_i we need to sum the entries $DP[j]$ over all indices j which are a possible last stop before C_i, which shows the formula in Equation (2).

Calculation order: We can calculate the entries of DP from the smallest index to the largest index.

Extracting the solution: All we have to do is read the value at $DP[n]$.

Running time: For $i = 0$, $DP[0]$ is computed in $O(1)$ time. For $i \ge 1$, the entry $DP[i]$ is computed in $O(i)$ time (as we potentially need to take the sum of i entries). Therefore, the total runtime is $O(1) + \sum_{i=1}^{n} O(i) = O(n^2)$.

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) ?

Solution:

Assuming that $k_i > k_{i-1} + d/10$ for all i, we know that $k_i > k_{i-10} + d$, and hence $k_i > k_j + d$ for all $j \le i - 10$. Therefore, the sum in formula (2) contains at most 10 terms $DP[j]$ (and for each of them we can check in constant time whether we should include it or not, i.e., whether $k_i \le k_j + d$). So in this case the computation of the entry $DP[i]$ takes time $O(1)$ for all $0 \le i \le n$, and hence the total runtime is $O(n)$.

Exercise 7.4 *Animals in the zoo* **(1 point)**.

A number n of animal species have been recently discovered in Africa. The zoo of Zürich is interested in acquiring as many animals from the new species as possible before a special exhibition that is taking place on December 1st, and you were put in charge of this task. Because of the time constraint, you can only organize one shipping of animals. The shipment can hold a maximum total weight of W. Furthermore, due to logistical constraints, you cannot isolate the animals during the shipment. Therefore, you cannot simultaneously bring two animals where one of them is a predator of the other.

Let $A_1, \dots, A_n$ be the $n > 4$ discovered species. You know that the species A_1, A_2 and A_3 are not predators, but for $4 \leq i \leq n$, the species A_i is a predator of only the species A_{i-1}, A_{i-2} and A_{i-3} (this means that, for example, A_i it is not a predator of species A_{i-4} or A_{i+1}).

For every $1 \leq i \leq n$, an animal from the species A_i has weight $w_i > 0$, and provides a value $v_i > 0$ to the zoo. You would like to figure out the collection of animals that you can bring to the zoo, and which provides the maximum total value to the zoo. We assume that $(w_i)_{1 \leq i \leq n}$ and W are all positive integers. If you bring one animal from a species, then bringing another animal from the same species does not provide any additional value to the zoo. Therefore, there is no point in bringing two or more animals from the same species.

Provide a *dynamic programming* algorithm that solves this problem. The input to your algorithm are the weights $(w_i)_{1 \leq i \leq n}$ and values $(v_i)_{1 \leq i \leq n}$ of the animal species, and the maximum total weight W that is allowed in one shipping. In order to get full points, the runtime of your algorithm should be $O(nW)$.

Address the following aspects in your solution:

1. *Dimensions of the DP table*: What are the dimensions of the table $DP[\dots]$?
2. *Definition of the DP table*: What is the meaning of each entry?
3. *Computation of an entry*: How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries 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 final solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Solution:

Dimensions of the DP table: The DP table is two-dimensional, and its size is $(n+1) \times (W+1)$.

Definition of the DP table: For $0 \leq i \leq n$ and $0 \leq j \leq W$, the entry $DP[i][j]$ represents the maximum value of a collection of animals among $\{A_1, \dots, A_i\}$, which has a total weight of at most j, and which does not contain any two animals where one of them is a predator of the other. Here, we adopt the convention that for $i = 0$, we have $\{A_1, \dots, A_i\} = \varnothing$.

Computation of an entry: Initialize $DP[0][j] = 0$ for every $0 \leq j \leq W$.

For $1 \leq i \leq 3$ and $0 \leq j \leq W$, we can compute $DP[i][j]$ exactly like the knapsack problem using the formula

$$DP[i][j] = \max\Big\{DP[i-1][j] \;\;,\;\; \mathbf{1}_{\{j \geq w_i\}} \cdot \big(v_i + DP[i-1][j-w_i]\big)\Big\}. \tag{3}$$

Now for $4 \leq i \leq n$ and $0 \leq j \leq W$, we can compute $DP[i][j]$ using a modified formula that takes into account the predator constraint:

$$DP[i][j] = \max\Big\{DP[i-1][j] \ , \ \mathbf{1}_{\{j \geq w_i\}} \cdot \big(v_i + DP[i-4][j-w_i]\big)\Big\}. \tag{4}$$

Calculation order: We can calculate the entries of DP in order of increasing i. For fixed i, we can compute the entries $(DP[i][j])_{0\leq j\leq W}$ in any order of j.

Extracting the solution: All we have to do is read the value at $DP[n][W]$.

Running time: The entry $DP[i][j]$ can be computed in $O(1)$ time. Therefore, the total runtime is

$$\sum_{i=1}^{n}\sum_{j=0}^{W} O(1) = O\big((n+1)\cdot(W+1)\big) = O(nW).$$

Exercise 7.5 *Partitioning integers in three equal parts* ***(This exercise is from the January 2021 exam)****.*

You are given an array of n natural numbers $a_1, \dots, a_n \in \mathbb{N}$ summing to $A := \sum_{i=1}^{n} a_i$, which is a multiple of 3. You want to determine whether it is possible to partition $\{1, \dots, n\}$ into three disjoint subsets I, J, K such that the corresponding elements of the array yield the same sum, i.e.

$$\sum_{i\in I} a_i = \sum_{j\in J} a_j = \sum_{k\in K} a_k = \frac{A}{3}.$$

Note that I, J, K form a partition of $\{1, \dots, n\}$ if and only if $I \cap J = I \cap K = J \cap K = \emptyset$ and $I \cup J \cup K = \{1, \dots, n\}$.

For example, the answer for the input $[2, 4, 8, 1, 4, 5, 3]$ is *yes*, because there is the partition $\{3, 4\}$, $\{2, 6\}$, $\{1, 5, 7\}$ (corresponding to the subarrays $[8, 1]$, $[4, 5]$, $[2, 4, 3]$, which are all summing to 9). On the other hand, the answer for the input $[3, 2, 5, 2]$ is *no*.

Provide a *dynamic programming* algorithm that determines whether such a partition exists. Your algorithm should have an $\mathcal{O}(nA^2)$ runtime to get full points. Address the following aspects in your solution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\dots]$? What is the meaning of each entry?
2) *Computation of an entry*: How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries that do not depend on others.
3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
4) *Extracting the solution*: How can the final solution be extracted once the table has been filled?
5) *Running time*: What is the running time of your algorithm? Provide it in Θ-notation in terms of n and A, and justify your answer.

Size of the DP table / Number of entries: $(n+1) \times (A+1) \times (A+1)$.

Meaning of a table entry: For $0 \le m \le n$ and $0 \le B, C \le A$, the corresponding entry in the DP table is defined as

$$DP[m,B,C] = \begin{cases} 1 & \text{if there are two disjoint sets } I, J \subseteq \{1,\dots,m\} \text{ such} \\ & \text{that } \sum_{i\in I} a_i = B \text{ and } \sum_{j \in J} a_j = C, \\ 0 & \text{otherwise.} \end{cases}$$

Computation of an entry (initialization and recursion):

We initialize the values for $m = 0$ as

$$DP[0,B,C] = \begin{cases} 1 & \text{if } B = C = 0, \\ 0 & \text{otherwise.} \end{cases}$$

The other entries are then computed as

$$DP[m+1,B,C] = \max\{DP[m,B,C], DP[m,B-a_{m+1},C], DP[m,B,C-a_{m+1}]\}\,.$$

In this formula we assume that if $a_{m+1} > B$, then $DP[m, B - a_{m+1}, C] = 0$, and if $a_{m+1} > C$, then $DP[m, B, C - a_{m+1}] = 0$.

For completeness, we now show the correctness of this formula (you do not have to do it unless we explicitly ask for it in the task).

It is possible to get two disjoint subsets of $\{a_1, \dots, a_{m+1}\}$ summing to B and C if and only if there are two disjoint subsets of $\{a_1, \dots, a_m\}$ that are summing to either B and C (so we don't need to use a_{m+1}), $B - a_{m+1}$ and C (so we add a_{m+1} to the first subset), or B and $C - a_{m+1}$ (so we add a_{m+1} to the second subset).

Order of computation: We can compute the values $DP[m, B, C]$ by increasing order in m. The order for B and C doesn't matter.

Extracting the result: The answer to the problem is *yes* if $DP[n, A/3, A/3] = 1$ and *no* if $DP[n, A/3, A/3] = 0$.

Running time: We need to fill $(n+1)(A+1)^2$ entries, and each of them can be computed in constant time $\Theta(1)$. Therefore, the running time is $\Theta(nA^2)$.