Dynamic Programming Approach for Solving Rectangle Partitioning Problem

Sheinna Yendri, Rully Soelaiman, *Member, IAENG*, Umi Laili Yuhana, *Member, IAENG*, and Sheilla Yendri

***Abstract*—Rectangle partitioning is one of the most common combinatorial problems which main purpose is to count possible configurations based on some given variables. Typical approach to solve this problem needs a polynomial time complexity, which leads to a new issue when time and memory is a crucial factor in this problem solving. One specific rectangular partitioning problem that has this issue is to count how many possible partitions can be done on a $M \times N$ rectangle to divide it into exactly two subregions. In this paper, we propose a novel solution for the aforementioned problem using dynamic programming with profile method, also known as broken profile DP. This broken profile DP method is famous for solving complex 2D grid problems by breaking it into some simpler subproblems and exploiting the special structure of the particular problem, which is referred as a profile. A disjoint-set data structure is used alongside with broken profile DP to validate each line configuration's connectivity, since the DP transition is processed line-by-line. In the implementation later on, it is necessary to implement big integer in order to store very big values. Based on the case study testing result, the solution using broken profile dynamic programming method requires an average time of 0.0012 seconds and an average memory of 530 KiB. This solution managed to rank first both timewise and spacewise on E-Olymp Online Judge site, with a grade A in terms of memory usage, which means it uses less resources than 90% of all solutions submitted before.**

***Index Terms*—dynamic programming, broken profile, rectangle partitioning, disjoint-set data structure.**

I. Introduction

RECTANGLE partitioning problem mostly involves counting how many possible configurations that can be constructed with a given condition [1]. This problem can indeed be easily solved with a conventional way of using a polynomial time algorithm [2]. However, this naïve approach is only applicable to a narrow range of variable size, which is a major drawback for real life usage where time and memory is crucial. Henceforth, rectangle partitioning problem is considered as one of the most challenging counting problem [2] in the field of combinatorics domain [3], [4], [5]. One particular problem that is negatively impacted by this polynomial time complexity is to count how many valid rectangular partition configurations exist if the rectangle, sized $M \times N$, is partitioned into exactly two subregions, where each subregion must be a connected set of unit cells.

By using a complete search method (Figure 1), such as "Depth First Search" (DFS) algorithm [6], in case of a rectangle sized 2×2, it will take $\mathcal{O}(2^3 \times 8)$ to generate all possible configurations and validate each configuration by traversing the entire tree (runs in $\mathcal{O}(V + E)$, where V is the number of vertices and E is the number of edges). This will approximately take only 0.1 milliseconds. The detrimental side of this method will take effect when the problem is conditioned with 11×11 rectangle size, 2 seconds of time limit, and 64 MiB of memory limit. By using DFS algorithm, with the worst case scenario of a 11×11 rectangle, we need to traverse all 2^{120} possible configurations, that approximately requires $\mathcal{O}(2 \times 10^{36})$ and takes 10^{27} seconds which is equivalent to 31.7 quintillion years, and thus not an ideal solution for real life application.

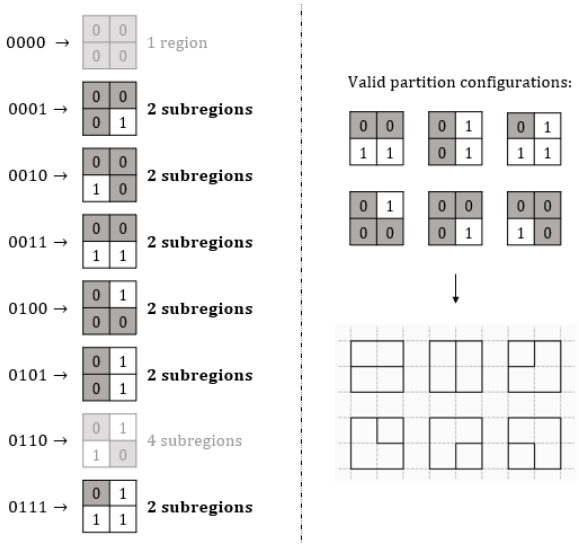

Fig. 1. Naïve approach to count valid partition configurations for a rectangle sized 2×2

In this paper, we proposed a novel solution that only needs an average time of 1.2 milliseconds to count all valid partition configurations for a rectangle sized 11×11, which is approximately 10^{30} times faster compared to the conventional way. This result can be achieved by using a broken profile dynamic programming method to solve each subproblem only once [7] while exploiting the special structure of the problem that is referred as "profile" [8]. In addition to the broken profile DP implementation, disjoint-set data structure is also needed to help validate each subregion connectivity [9], so we only need to maintain the valid configurations by storing it in a table for future references and thus significantly decrease the memory and time complexity.

The rest of the paper is organized as follows: Section

Manuscript received October 23, 2021; revised March 25, 2022. This work was supported in part by Sepuluh Nopember Institute of Technology, Surabaya, Indonesia.

Sheinna Yendri is a graduate student at Sepuluh Nopember Institute of Technology, Department of Informatics, Surabaya, Indonesia (e-mail: sheinnayendri@gmail.com)

Rully Soelaiman is an associate professor at Sepuluh Nopember Institute of Technology, Department of Informatics, Surabaya, Indonesia (e-mail: rully130270@gmail.com)

Umi Laili Yuhana is an associate professor at Sepuluh Nopember Institute of Technology, Department of Informatics, Surabaya, Indonesia (e-mail: yuhana@if.its.ac.id)

Sheilla Yendri is a bachelor of School of Materials Science and Engineering, Nanyang Technological University, Singapore (e-mail: sheilla001@e.ntu.edu.sg)

II presents broken profile dynamic programming method, which is proposed to solve the rectangle partitioning problem. Section III presents the experimental results and analysis. Finally, the conclusion is stated in Section IV.

II. Methodology

A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem [10]. Dynamic programming is usually applied to optimization problems [2], [11], [12], which makes it unusual that a combinatorial problem [13] is approached by using dynamic programming method rather than a combinatoric solution [14]. In solving this rectangle partitioning problem, we are using dynamic programming with profile method [15] which is a subset of bitmask dynamic programming. Dynamic programming with profile is also often referred as "broken profile dynamic programming" [16]. Problems falling under this category generally have the following properties [8]:

1) They are about filling a 2D grid.
2) One of the dimensions is much smaller than the other.
3) When filling the grid, each cell depends only on adjacent cells.
4) The cells do not have many possible values (usually only 2).

The properties mentioned above can all be found in our problem [17], which is counting how many valid configurations exists that can partition a rectangle sized $M \times N$ into two subregions. The rectangle we are going to partition is a 2D grid that fulfilled the first property. Next, with a $M \times N$ rectangle, the algorithm time complexity is exponentially proportional to the column size (N) and is directly proportional to the row size (M). Therefore, compared to the column size, the effect of the row size is negligible which satisfied the second property. The third property is the key factor of our proposed solution, as it means that we can process the cells line-by-line. This eliminates the need to care about each cell and put our main focus only on cells in the last processed row, hence the name "broken profile" [8]. As our problem is to partition the rectangle into two subregions, we can fulfill the fourth property by representing the subregions using "binary" value. In this case, we are not using $0/1$ (bitmask) as our "binary" value but an even and odd value instead [18]. The reason being is because there are some cases that are not able to be represented by $0/1$ as shown in Figure 2. These cases covered all line configurations which have cells that should be in the same subregion but have not been connected yet. As can be seen from Figure 2, the first three cells are represented by "0", while the last two cells are represented by "2", by our definition, subregions are defined based on the cells are represented by even or odd numbers. Since both "0" and "2" are even numbers, meaning these five cells should be in the same subregion, but they are not actually connected yet (no adjacent tiles connect them) since they are still separated by five "1" cells in the middle. Nonetheless, that does not mean that "0001111122" is an invalid configuration, since there are still more rows that can connect these even number cells.

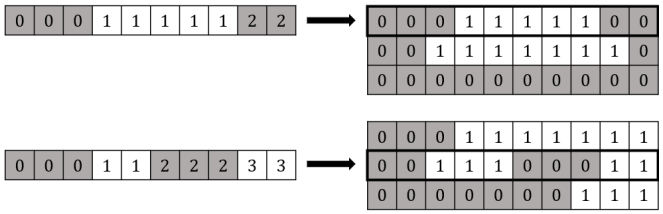

Fig. 2. Illustration of cases that cannot be handled by $0/1$ representations

When developing a dynamic-programming algorithm [10], Cormen suggested a sequence of four steps. First, characterize the structure of an optimal solution. Next, recursively define the value of the optimal solution identified in the first step. Third, compute the value of the optimal solution based on the recursive function formulated in the second step. Lastly, construct an optimal solution from the computed information in step three. In our problem, the fourth step will be omitted since we only need the amount of valid rectangular partition configurations, and not the configurations themselves. Each of these three steps will be further discussed in details in Sections II-A to II-C.

A. The structure of an optimal configuration

For our first step in the dynamic-programming paradigm, we find the optimal substructure and use it to find the optimal solutions for subproblems which will lead to an optimal solution for the main problem. In the rectangle partitioning problem, we can perform this step as follows. For convenience, let S be the i^{th} line configuration, and that there exists X valid configurations that can be constructed until the i^{th} line. Suppose that T is the $(i+1)^{th}$ line configuration, and both S and T line configurations can be defined as valid configurations; meaning the rectangle is partitioned into at most two subregions until the $(i+1)^{th}$ line, then we can be assured that at least we have X valid rectangle partitioning configurations up until $(i+1)^{th}$ line. Why? Because if up until i^{th} line there are X valid rectangle partitioning configurations, and that both S and T line configurations are also valid, this means if we combined $(i+1)^{th}$ line configuration, T, with the previous line configurations, namely $i^{th}, (i-1)^{th}, (i-2)^{th}, \ldots, 0^{th}$ (zero-indexing), at least X valid rectangle partitioning configurations will be constructed too. If it turns out the number of valid rectangle partitioning configurations until $(i+1)^{th}$ line with the line configurations T are less than X, then there must be also less than X valid rectangle partitioning configurations until i^{th} line with the line configuration S: a contradiction.

Now we implement our optimal substructure in the subproblems to show that the optimal solutions in subproblems can also be used as an optimal solution to the main problem. We have seen that for each line, they only depend on exactly one previous line configuration. Hence, we can build an optimal solution to an instance of the rectangle partitioning problem by splitting the problem into two subproblems (optimally combining 0^{th} until i^{th} line configurations with the $(i+1)^{th}$ line configuration), finding the optimal solutions to subproblem instances, and then combining these optimal subproblem solutions. We must ensure that when we search for valid rectangle partitioning configurations, we have con-

sidered all possible configurations, so that we are sure of having counted all the valid ones optimally.

B. A recursive solution

Next, we define the optimal solutions for the subproblems in a recursive formula. For the rectangle partitioning problem, we pick as our subproblems the problems of determining the number of valid partition configurations discovered until i^{th} line, for $0 \leq i \leq M-1$. Let $dp[i,S]$ be the total number of valid partition configurations that have been found until i^{th} line, where S is the line configuration in the i^{th} line; for the full problem, the total number of valid partition configurations for a rectangle sized $M \times N$ would thus be $\sum_{a \in A} dp[M-1,a]$, where A is the set of all valid $(M-1)^{th}$ line configurations that are referred as the end-states.

We can define $dp[i,S]$ recursively as follows. If $i = 0$, meaning it defines the amount of valid configurations found in the first row of the rectangle, which will be referred as the base cases. Trivially, each base case will have only one amount of valid configuration which is S itself. Thus, $dp[0,S] = 1$, where S represents all base configurations that is valid as a start. To compute $dp[i,S]$, we take advantage of the structure of an optimal solution from step 1. Let us assume that to count all possible valid partition configurations, we split the last line configuration (i^{th} line configuration) with all previous line configurations (0^{th} to $(i-1)^{th}$ line configuration), where $0 \leq i \leq M-1$. Then, $dp[i,S]$ equals the sum of current amount of valid partition configurations when S is the i^{th} line configuration with the amount of valid partition configurations when T is the $(i-1)^{th}$ line configuration, and it is assured that S and T can be connected and do not partition the rectangle into more than two subregions. Total amount of possible line configurations that are represented by T are at most 2^N configurations, based on the column size of the rectangle. Thus, our recursive definition for the total amount of valid partition configurations for a $M \times N$ rectangle is defined in equation (1), where T represents any valid line configurations that can be connected with S line configurations without making the rectangle partitioned into more than two subregions.

$$dp[i,S] = \begin{cases} 1, & i = 0 \\ dp[i,S] + dp[i-1,T], & 1 \leq i \leq M-1 \end{cases} \quad (1)$$

The $dp[i,S]$ values give the total amount of valid configurations to subproblems which then will be used to compute the total amount of valid rectangular partition configurations overall.

C. Computing valid configurations

At this point, we could easily write a recursive algorithm based on recurrence shown in equation (1) to compute the total amount of valid partition configurations for a rectangle sized $M \times N$. However, this recursive algorithm takes exponential time, which is no better than the brute-force method of checking all possible rectangular partition configurations [10].

Observe that we have relatively few distinct subproblems: one subproblem for each choice i and S satisfying $0 \leq i \leq M-1$, and at most 2^N possible S line configurations. A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree. This property of overlapping subproblems is the second hallmark of when dynamic programming applies (the first hallmark being optimal substructure).

Instead of computing the solution to recurrence recursively (using equation (1)), we compute amount of valid rectangular partition configurations by using a tabular, bottom-up approach. In order to implement the bottom-up approach, we must determine the base cases first. Base cases represent all dynamic programming states that are going to be the starting point for the dynamic programming transition. Since the problem requires the rectangle to be partitioned into exactly two subregions, the only valid configurations that can be defined as base cases are line configurations that have one region, two subregions, or three subregions, as stated in equation (2).

$$S = \begin{cases} \text{“}00\ldots00\text{”}, & 1 \text{ region} \\ \text{“}00\ldots0011\ldots11\text{”}, & 2 \text{ subregions} \\ \text{“}00\ldots0011\ldots11\ldots1122\ldots22\text{”}, & 3 \text{ subregions} \end{cases} \quad (2)$$

We can assure that the three possible configuration types stated in equation (2) have covered all the possibilities. For the first possibility, we only need to make sure that in the next lines there will be at least one “1” cell. While for the second possibility, we just need to make sure that every cell between the lines is connected to each other. For the third possibility, it has three subregions, which is over the maximum subregions allowed. However, this third type of line configuration is still possible as a base case because “0” and “2” cells can be connected through the next lines, does not have to be the exactly next line. In other words, all even number cells and all odd number cells must already be connected throughout all M lines. Thus, the rectangle is finally partitioned into two subregions, which are odd number subregion and even number subregion. An illustrative example of the three base cases can be seen in Figures (3-5).

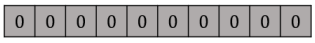

Fig. 3. Illustrative example of the first type base case line configuration

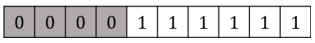

Fig. 4. Illustrative example of the second type base case line configuration

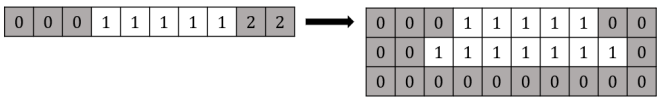

Fig. 5. Illustrative example of the third type base case line configuration

When a first line configuration S is divided into four subregions, it will be impossible to construct a valid final rectangular partition configuration, since in the end at least there will be three subregions formed. Suppose S =“$00\ldots0011\ldots1122\ldots2233\ldots33$”, then if we want to connect “0” cells and “2” cells into one subregion, “1” cells and “3” cells will not be able to be connected into one subregion. Thus, the rectangle will be partitioned to at least three subregions. Similarly, if “1” cells and “3” cells are can be merged into one subregion, hence “0” cells and

"2" cells cannot be one subregion. If "1" cells and "2" cells are one subregion, it will be the same as the third possibility, namely cells in the first row is divided into three subregions ("00 . . . 0011 . . . 1122 . . . 22"). An illustration for this case can be seen in Figure 6.

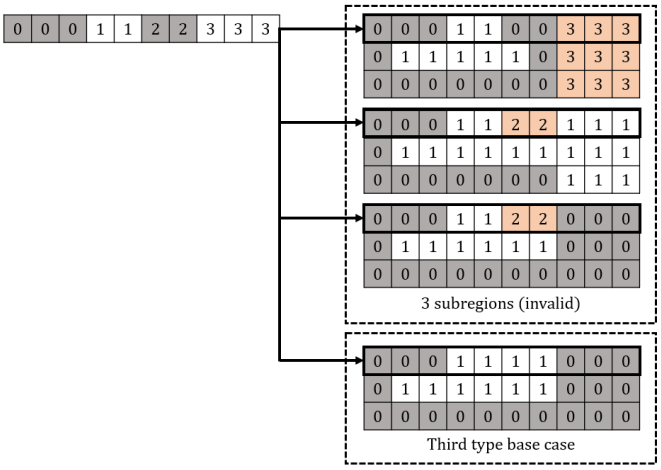

Fig. 6. Illustrative example when first line configuration cells are divided into four subregions

Based on the three possible first line configuration scenarios, it is found that the base case, which is the first line configuration, S, can be represented by a string with length N, and formally defined in equation (3) for any value of L and R where $0 < L \leq R < N$, other than the trivial base case: "00 . . . 00" (with length N). These base cases are valued one, which means to form the S line configuration on the first line, there is one valid configuration.

$$S = \begin{cases} \text{"0"}, & i < L \\ \text{"1"}, & L \leq i \leq R \\ \text{"2"}, & R < i. \end{cases} \tag{3}$$

Now that we have determined the formula for base cases, next is doing the dynamic programming transition based on equation (1). In the transition, we need to check all next possible line configurations and then validate if that next possible line configuration is a valid one. Valid means that the next line configuration's cells can be connected to the cells in the previous line configuration. To do this inter-cells' connection validation, we can use the help of disjoint-set data structure [9]. This validation needs to be done throughout all possible $(i+1)^{th}$ line configurations, 2^N line configurations at most, which are constructed from the previous line configuration, the i^{th} line configuration. Figure 7 shows how disjoint-set data structure can be used to help the validation process. If at least one cell cannot be connected, then the line configuration is assumed as an invalid one. Only valid line configurations will be counted. This validation process will be carried out iteratively from the second row until the last row, and the valid configuration computation will be accumulated in each row. Note that each cell from a line configuration can connect with another line configuration's cell only when both cells are represented by odd numbers or even numbers.

How about a line configuration that only consists of one kind of cell, for example, all "0"s or all "1"s? This

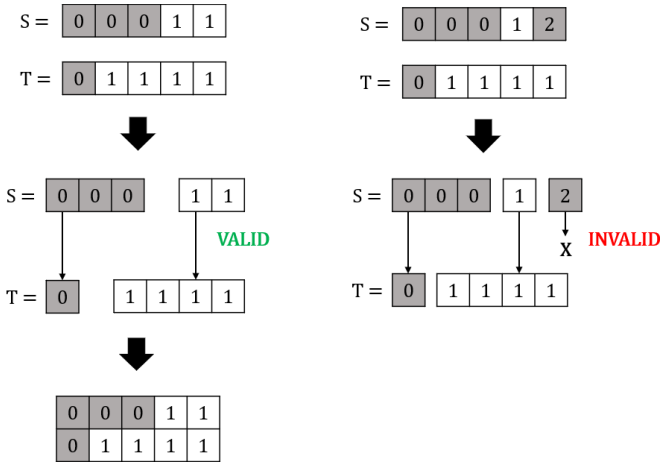

Fig. 7. Validation illustration of the possible next line configuration using the help of disjoint-set data structure

type of line configuration, that only has one region will have a different validation process. Why? Because this one-region line configuration has a unique characteristic, which is closing the chance to partition the rectangle in the upcoming rows. Thus, this kind of configuration can be stated as an ending line configuration, that from now on will be referred as "*end*" line configuration. Even so, not all one-region line configurations can be included as "*end*" line configurations, since the problem required the rectangle to be partitioned into exactly two subregions. Meaning, these one-region line configurations will be stated as "*end*" line configurations, only if the rectangle is already partitioned into exactly two subregions in the previous rows. Hence, we only need to make sure the previous line configuration consists of exactly two different cell's values that represent two subregions. These "*end*" line configurations will later contribute to the overall amount of valid rectangular partition configurations, since these line configurations already assure that the rectangle is partitioned into two subregions from the validation process. This validation process for one-region line configurations can be seen in Figure 8

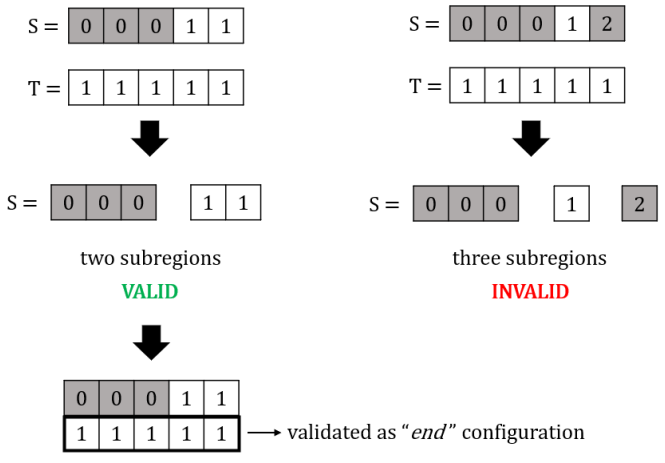

Fig. 8. Illustration of "*end*" configuration validation for one-region line configuration

While counting the total amount of valid rectangular partitions in each row, we need to make sure no double counting occurs [19]. This is why, after a next possible line

configuration is validated, it also needs to be compressed. The compression process is not only used to avoid double counting, but is also used to re-represent cells that are joining into one subregion, so these cells can be represented using the same number. The compression process consists of two steps. The first step is changing the next line configuration cells' values based on each cell's parent number representation. Each cell's parent number representation is already recorded previously while validating the connection between two line configurations using disjoint-set data structure. After all cells' values have been updated, second step will be executed, which is adjusting the line configuration cells' values into a zero-based numbering. This second step is the important step not to be missed so double counting will not occur. The illustrative example of compression process can be seen in Figure 9.

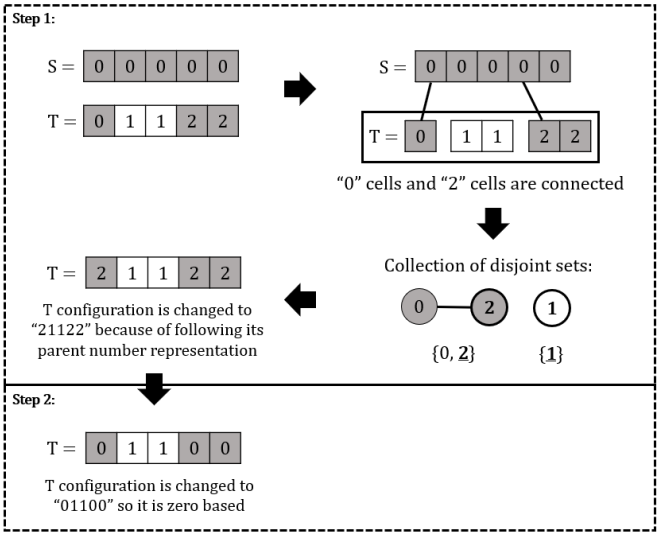

Fig. 9. Compression illustration of the possible next line configuration

The dynamic programming transition will continue until the last row, $(M-1)^{th}$ line. After reaching the last row, we still need to count the total valid rectangular partition configurations. To count them, we need to define which $(M-1)^{th}$ line configurations that can be the end states. As mentioned before, end line configurations can contribute directly to the final answer, so these configurations are valid end states. And, there are other line configurations that can also be valid end states. Thus, the final answer that consists the total valid rectangular partition configurations are formed in equation (4), where X represents the set of $(M-1)^{th}$ line configurations that are valid end states.

$$ans = dp[M-1, \text{“}end\text{”}] + \sum_{x \in X} dp[M-1, x] \qquad (4)$$

Based on equation (4), we can say that the only valid end states are those who referred as end configurations, plus other line configurations that are included in set X. Furthermore, set X consists of $(M-1)^{th}$ line configurations that are all line configurations represented by a string, x with length N, and formally defined in equation (5), for $1 \leq L < N$ and $1 \leq R \leq N-L$.

Algorithm 1 Counting valid rectangular partitions

Input: M, N
Output: ans

```
GENERATEBASECASE(N)
for i ← 1 to M − 1 do
    for each conf, x ∈ dp[i − 1] do
        S ← conf
        if S = "end" then
            dp[i][S] ← dp[i][S] + x
        else
            for mask ← 0 to 2^N − 1 do
                if mask is odd then T ← '1'
                else T ← '0'
                end if
                for j ← 1 to N − 1 do
                    T ← T • T[j − 1]
                    m ← ⌊mask / 2^j⌋
                    t ← ASCII(T[j])
                    if ISDIFFPARTITION(m, t) then
                        T[j] ← T[j] + 1
                    end if
                end for
                if ISONEREGION(mask) then
                    if ISVALIDENDCONF(S, T) then
                        T ← "end"
                    else T ← λ
                    end if
                else if ISVALIDCONF(T) then
                    T ← COMPRESSMASK(T)
                else T ← λ
                end if
                if T ≠ λ then
                    dp[i][T] ← dp[i][T] + x
                end if
            end for
        end if
    end for
end for
ans ← COUNTANS(M, N)
return ans
```

$$x_i = \begin{cases} 0, & i < L \text{ or } L+R \leq i \\ 1, & L \leq i < L+R. \end{cases} \qquad (5)$$

To be clear, for the example case of a rectangle sized 3×3, the computation for the final answer on how many valid rectangular partition configurations exist, with the configurations shown can be seen in Figure 10. No other end states will be a valid one for this example case, other than 010, 001, 011, and end line configurations. Thus, the end state formula represented in equation (5) is proven to be true. To conclude, the overall broken profile dynamic programming algorithm that is used to count valid rectangular partition configurations is summarized as follows in Algorithm 1.

In addition, since the answer might be very big, approximately over 2^{64}, to implement this algorithm, will need big integer [20] implementation or primitive data types modification.

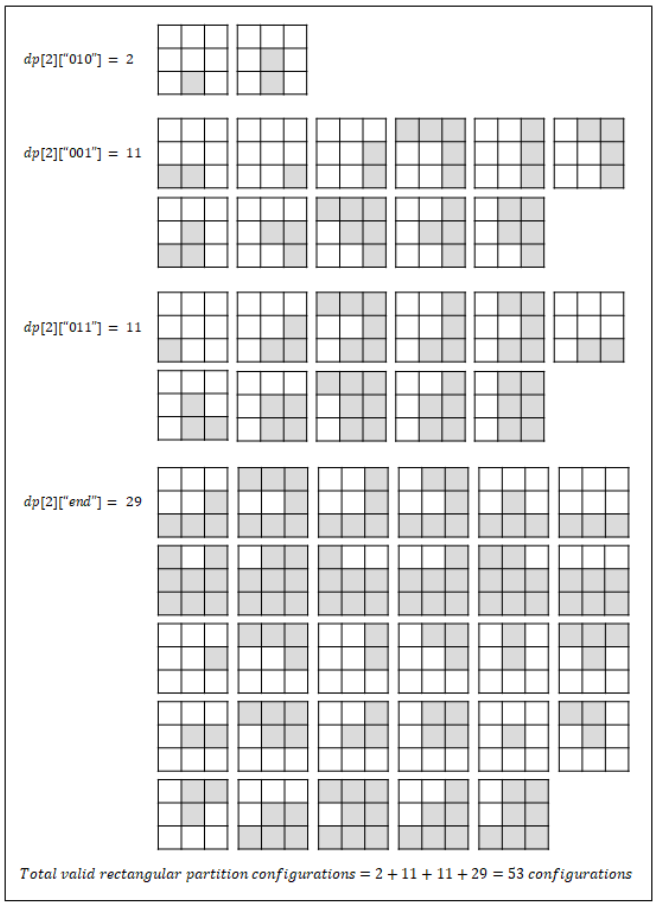

Fig. 10. Illustration of valid 3×3 rectangular partition configurations

III. Experimental Results and Analysis

In this section, we examine the validity and the performance of the proposed approach. All algorithms were implemented using C++11 programming language. To test the validity of the dynamic programming approach, we use the submission of source code on E-Olymp Online Judge as a third-party online platform for source code checker, that uses Alpine Linux 3.6 and GCC 6.3.0 compiler [21]. This source code is, then, executed and the resulted output will be compared with the expected answers that are provided by the problem maker. On the other hand, two environment testing platforms were used to evaluate the performance of the source code. The first one is on a live environment using E-Olymp Online Judge's feedback to assess both memory and time consumption compared with all previous solutions. While the other is on a local environment using a PC with Intel® Core™ i7-1165G7 4 Core Processor and 8192 MB of memory, running Windows 10 with GCC 7.50 compiler, to compare the program runtime of two big integer implementation methods in C++11: by modifying the primitive data type and by using the extension data type.

In Section III-A, we will elaborate on the validity check of the broken profile dynamic programming method that is used to count the number of valid rectangular partitions. Moreover, in Section III-B we will further go into detail about the performance examination of the proposed method, both timewise and spacewise.

A. Validity Examination

Figure 11 shows how the dynamic programming approach used according to Section II are scored on E-Olymp Online Judge. E-Olymp Online Judge's testing system will give various responses based on the program execution time, memory usage, exit code, and the resulted output. First of all, the system compares the program execution time with the limit. If it is exceeded, the response will be **"Time limit exceeded"**. Otherwise, system will continue check the memory usage. If the amount of memory used during execution exceeds the allowed limit, the testing system will respond **"Memory Overflow"**. According to exit code standard, the program exit code should be 0 in case of success, and other than 0 in case of an error. Thus, if the program returns a nonzero exit code, a **"Runtime error"** status will be received. At last, depending on the resulted output, the check may end with a result **"Accepted"** or **"Wrong Answer"** [22]. The code submissions have been executed at least 10 times to ensure the solution's both validity and consistency during the validity test.

All ten submissions received **"Accepted"** responses which prove that our approach to count valid rectangular partitions using the fusion of broken profile DP method and disjoint-set data structure can provide correct answers within the time and memory limitation. For each submission, our program will be executed with one test suite containing 122 test cases as shown in Figure 12. **"Accepted"** status is given only when the code passes all test suites and test cases that proves our solution is a valid one.

B. Performance Examination

There are two factors that must be taken into considerations in performance examination, they are program runtime and memory usage. The first factor, program runtime, will be tested in both local, using own PC, and live environment, with the help of E-Olymp Online Judge site. As for the second factor, memory usage, will only be evaluated in E-Olymp Online Judge site.

1) Runtime: In local environment, while implementing the proposed approach, we used two types of big integer implementation. First, is by manipulating the primitive data types, which is expressing the big number using *pair of unsigned long long*. Second, is by using an extension data type, namely *unsigned __int128*. Both implementations will be used and compared in this experiment to determine which implementation has a better performance. To obtain the statistics of the program runtime, we executed the implemented approach on a local PC thirty times for each method.

Figure 13 shows the bar chart of every thirty trials' runtime value by modifying C++ primitive data type, with a mean value of 8.933 minutes, the fastest performance at 8.665 minutes in the 25^{th} trial, and the slowest performance at 9.895 minutes in the 13^{th} trial. Based on the chart's variation from the average line, it can be concluded that the overall runtime performance is consistent throughout the thirty trials. Highest spike that occurred in the 25^{th} trial with only 1.01 minutes longer than the average time happened because the program runtime is highly dependent on the processor load and Random Access Memory (RAM) space that is very unstable [23].

#	Submit date	Lang	Time	CPU	Memory	State
9197036	Aug 12, 2021, 12:50:57 PM	C++ 11 (gnu 10.2)	1 ms	1 ms	532	✓ Accepted
9197027	Aug 12, 2021, 12:45:02 PM	C++ 11 (gnu 10.2)	3 ms	1 ms	532	✓ Accepted
9196719	Aug 12, 2021, 9:41:44 AM	C++ 11 (gnu 10.2)	1 ms	1 ms	532	✓ Accepted
9196699	Aug 12, 2021, 9:31:01 AM	C++ 11 (gnu 10.2)	1 ms	1 ms	532	✓ Accepted
9196662	Aug 12, 2021, 9:14:49 AM	C++ 11 (gnu 10.2)	1 ms	1 ms	528	✓ Accepted
9196650	Aug 12, 2021, 9:08:42 AM	C++ 11 (gnu 10.2)	1 ms	1 ms	532	✓ Accepted
9196648	Aug 12, 2021, 9:08:06 AM	C++ 11 (gnu 10.2)	1 ms	1 ms	532	✓ Accepted
9196646	Aug 12, 2021, 9:07:54 AM	C++ 11 (gnu 10.2)	1 ms	1 ms	532	✓ Accepted
9196620	Aug 12, 2021, 8:59:22 AM	C++ 11 (gnu 10.2)	1 ms	1 ms	532	✓ Accepted
7444507	Oct 4, 2020, 10:30:22 AM	C++ 11 (gnu 10.2)	1 ms	1 ms	524	✓ Accepted

Fig. 11. Validity Test Examined by E-Olymp Online Judge

Test #	Status	Score	Duration	CPU	Memory
✓ Test suite #1	Accepted	100 / 100	1 ms	1 ms	532 KiB
✓ Test #1	Accepted	0.5 / 0.5	1 ms	1 ms	500 KiB
✓ Test #2	Accepted	0.5 / 0.5	1 ms	1 ms	456 KiB
✓ Test #3	Accepted	0.5 / 0.5	1 ms	1 ms	488 KiB
✓ Test #4	Accepted	0.5 / 0.5	1 ms	1 ms	452 KiB
✓ Test #5	Accepted	0.5 / 0.5	1 ms	1 ms	528 KiB
✓ Test #6	Accepted	0.5 / 0.5	1 ms	1 ms	448 KiB
...	...	...	...	...	...
✓ Test #117	Accepted	1 / 1	1 ms	1 ms	460 KiB
✓ Test #118	Accepted	1 / 1	1 ms	1 ms	472 KiB
✓ Test #119	Accepted	1 / 1	1 ms	1 ms	528 KiB
✓ Test #120	Accepted	1 / 1	1 ms	1 ms	436 KiB
✓ Test #121	Accepted	1 / 1	1 ms	1 ms	452 KiB
✓ Test #122	Accepted	1 / 1	1 ms	1 ms	528 KiB
		100 / 100	1 ms	1 ms	532 KiB

Fig. 12. Test Cases Execution by E-Olymp Online Judge

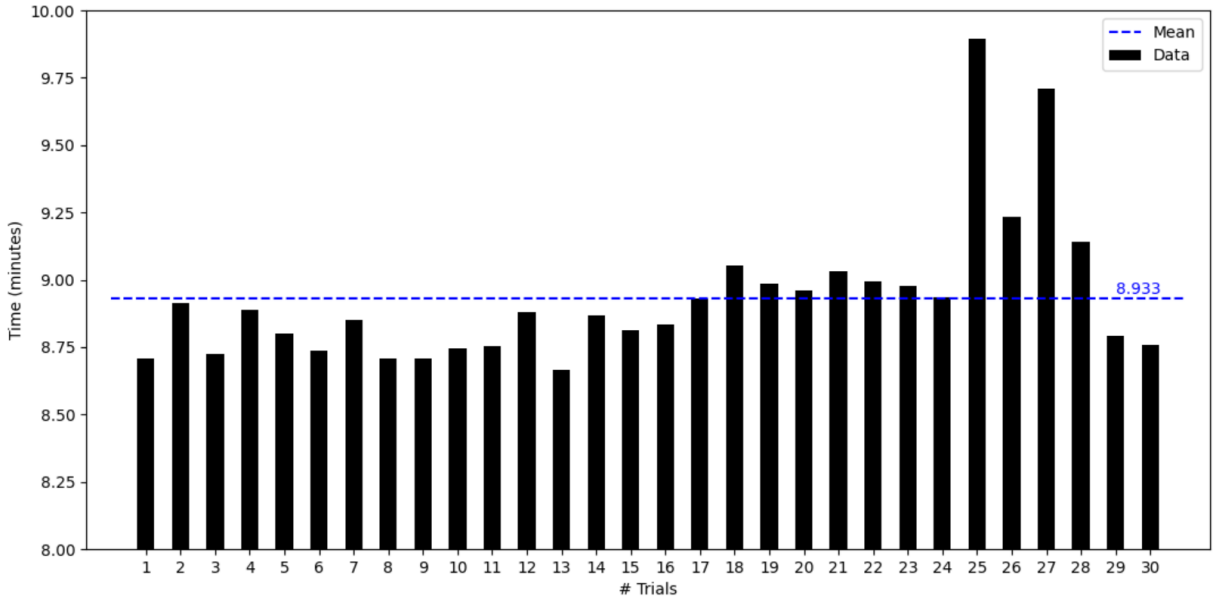

Fig. 13. Proposed Approach Runtime by Modifying C++ Primitive Data Type

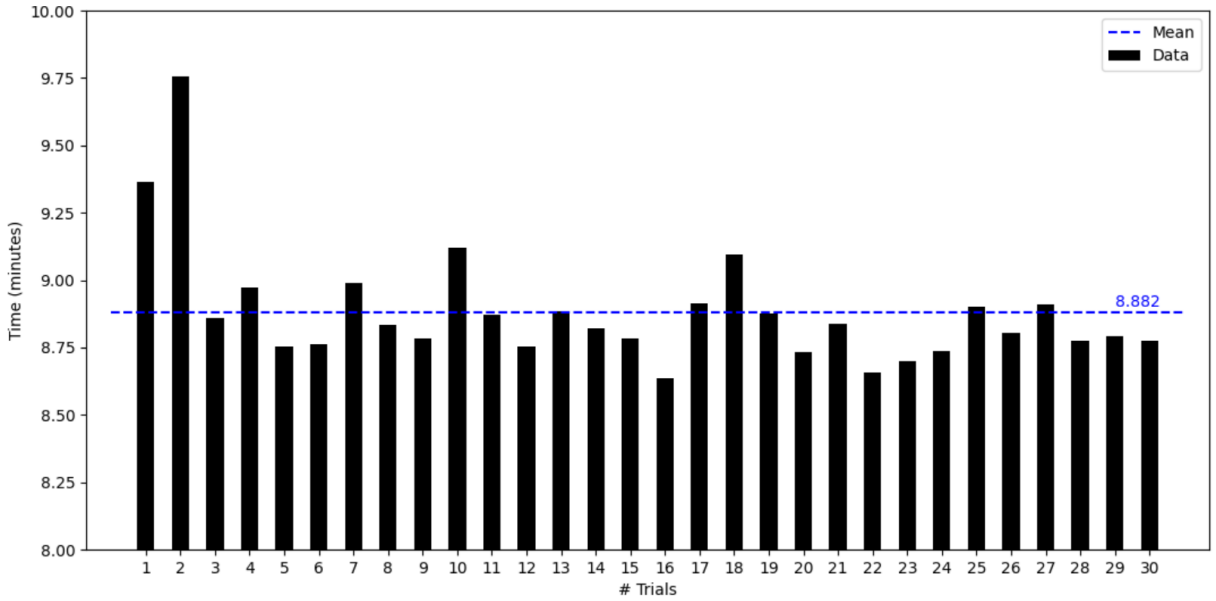

Fig. 14. Proposed Approach Runtime by Using C++ Extension Data Type

Best solutions

#	User	Submit date	Time	Memory	Lang
7444507	sheinnayendri	Oct 4, 2020, 10:30:22 AM	1 ms	524 KB	C++ 11 (gnu 10.2)
7104765	arena	Jun 2, 2020, 7:49:50 AM	2 ms	1664 KB	C++ 11 (gnu 10.2)
6741154	mstyva04	Mar 3, 2020, 4:29:13 PM	3 ms	1812 KB	C++ 11 (gnu 10.2)
5820556	Neota	Oct 10, 2019, 6:24:37 AM	4 ms	1804 KB	C++ 11 (gnu 10.2)
6605216	memmedlielnur2007	Feb 8, 2020, 7:12:18 PM	3 ms	1844 KB	C++ 11 (gnu 10.2)

Fig. 15. Solution ranking based on time and memory usage on E-Olymp Online Judge

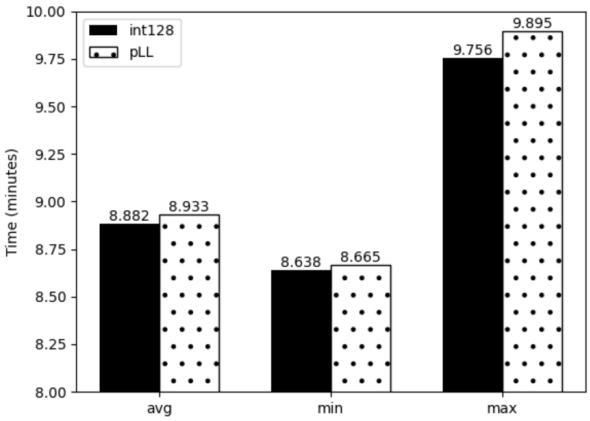

Fig. 16. Big Integer Implementation Comparison

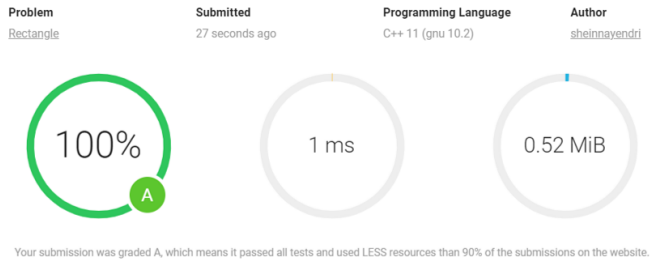

Fig. 17. Memory Usage Grading Feedback from E-Olymp Online Judge

Figure 14 shows the bar chart of every thirty trials' runtime value by using C++ extension data type: *unsigned __int128*. This chart has a mean value of 8.882 minutes, the fastest performance at 8.638 minutes in the 16^{th} trial, and the slowest performance at 9.756 minutes in the 2^{nd} trial. This bar chart's variation from the average line also concludes that the overall runtime performance is consistent throughout the thirty trials. Similar to the previous figure, there are several spikes happened in some trials because of the PC load and storage space that is inconsistent throughout the whole experiment [23].

To sum up (Figure 16), a double bar graph presents the comparison of two aforementioned big integer implementations in terms of average, minimal, and maximal runtime value. This graph clearly shows that the extension data type usage is relatively better than the usage of *pair of unsigned long long* since it has lower program runtime in all three aspects, with 8.882 minutes of average time, 8.638 minutes of fastest runtime, and 9.756 minutes of slowest runtime. Furthermore, based on each method's range value, it can be concluded that *unsigned __int128* is more consistent compared with *pair of unsigned long long*. This statement is also supported by the fact that *unsigned __int128* has a standard deviation of 13.3, which is 3.1 lower than *pair of unsigned long long*'s. Therefore, in big integer implementation, the usage of extension data type is a better approach in all statistics aspects. However, in terms of generality, the big integer implementation by modifying primitive data type is more practical since it can be implemented in other programming languages that do not have big integer built-in data types. Moreover, the average runtime value from the primitive data type modification is only 3 seconds slower than the extension data type usage which is still comparable.

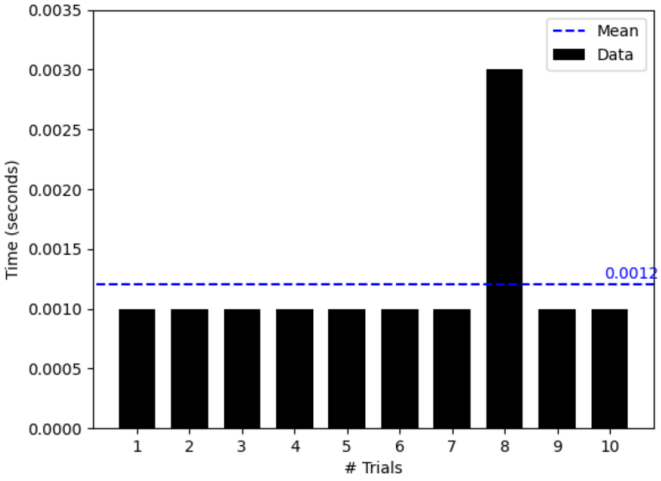

Fig. 18. Timewise Performance Measurement by E-Olymp Online Judge

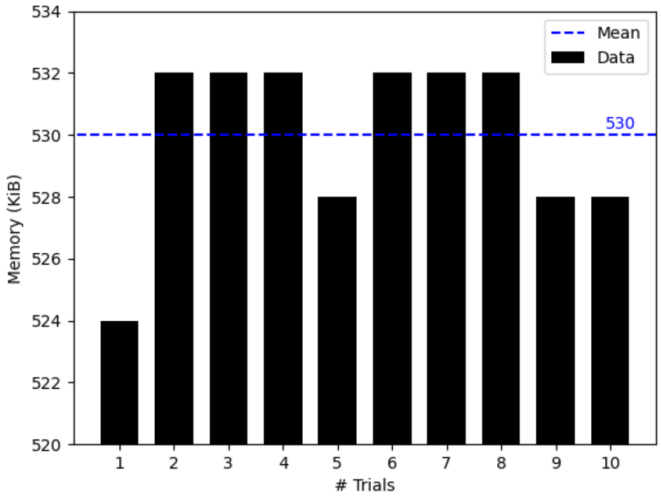

Fig. 19. Spacewise Performance Measurement by E-Olymp Online Judge

In live environment, the proposed method is submitted ten times and Figure 18 shows a bar chart of all ten submissions' execution time measured by E-Olymp Online Judge, with an average execution time of 0.0012 seconds. However, if we further observe this bar chart pattern, only one trial run (8^{th}) is 3 milliseconds while the remaining 9 trials consistently takes 1 millisecond to run. Therefore, this 8^{th} trial can be considered as an outlier caused by the server load inconsistency during certain busy time [24], [25].

2) Memory Usage: As mentioned in the beginning of this section, the proposed approach memory usage examination is done by E-Olymp Online Judge that provides a grading system for users to determine whether the submitted approach is efficient spacewise. Figure 17 shows that our proposed approach is graded A, meaning the solution used less resources than 90% of the submissions on the website. In average, our proposed approach only needs 530 KiB of resources out of 64000 KiB, the maximum memory limit given, shown in Figure 19. In conclusion, our broken profile dynamic-programming approach is indeed efficient spacewise.

Furthermore, Figure 15 shows how the solution is ranked among other users' solution on E-Olymp Online Judge. Our broken profile DP approach managed to be the best

solution that takes the first rank on the submission scoreboard both timewise and spacewise. In other words, our proposed approach is by far the most efficient method to count the amount of valid rectangular partitions exist that split the rectangle into exactly two subregions.

IV. Conclusion

In this paper, we apply a novel approach to solve a combinatorial problem in counting valid rectangular partitions by using broken profile dynamic programming method. This approach is based on three main concepts. First, we represent the two subregions of the rectangular partitions using odd and even number. From there, we get the line configurations as the profile that is being used in the dynamic programming approach. Second, by using the structure of an optimal configuration, we can get the recursive solution of this problem using dynamic-programming: bottom-up approach. Last, the usage of disjoint-set data structure that is used to validate each partition connectivity complements the dynamic programming transition in each line iteration.

The experimental results for this rectangle partitioning problem have shown that the proposed approach using broken profile dynamic programming method is by far the best approach, both timewise and spacewise. In future work, it might be possible to have a faster approach by finding the analytical solution in closed-form expression.

References

[1] I. Pak, "Complexity problems in enumerative combinatorics," *arXiv preprint arXiv:1803.06636*, 2018.

[2] Y. Cui, "Dynamic programming algorithms for the optimal cutting of equal rectangles," *Applied Mathematical Modelling*, vol. 29, no. 11, pp. 1040–1053, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0307904X05000314

[3] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Partitioning a Square into Rectangles: NP-Completeness and Approximation Algorithms," Laboratoire de l'informatique du parallélisme, Research Report LIP RR-2000-10, Mar. 2000. [Online]. Available: https://hal-lara.archives-ouvertes.fr/hal-02101984

[4] D. Du, K. Ko, and X. Hu, *Rectangular Partition*, ser. Springer Optimization and Its Applications. Springer New York, 2011, pp. 165–170. [Online]. Available: https://books.google.co.id/books?id=j2GGXlXozVMC

[5] M. Atallah and M. Blanton, *Floorplan Sizing and Classic Divide and Conquer*, ser. Chapman & Hall/CRC Applied Algorithms and Data Structures series. CRC Press, 2009, pp. 8–10, 8–13. [Online]. Available: https://books.google.co.id/books?id=SbPpg_4ZRGsC

[6] S. Halim and F. Halim, *Depth First Search (DFS)*. Lulu.com, 2013, no. v. 3, pp. 122–123. [Online]. Available: https://books.google.co.id/books?id=vUc-nwEACAAJ

[7] G. P. . S. S. George T. Heineman, *Dynamic Programming*. O'Reilly Media, Inc., 2015, no. v. 2, pp. 51–56.

[8] A. Qu. (2021) Dp on broken profile. [Online]. Available: https://usaco.guide/adv/dp-more?lang=cpp#dp-on-broken-profile

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Disjoint-set operations*. Cambridge, Massachusetts: The MIT Press, 2009, no. 3rd ed., pp. 561–564.

[10] ——, *Dynamic Programming*. Cambridge, Massachusetts: The MIT Press, 2009, no. 3rd ed., pp. 359–413.

[11] W.-Z. Sun, M. Zhang, J.-S. Wang, S.-S. Guo, M. Wang, and W.-K. Hao, "Binary particle swarm optimization algorithm based on z-shaped probability transfer function to solve 0-1 knapsack problem," *IAENG International Journal of Computer Science*, vol. 48, no. 2, pp. 294–303, 2021.

[12] J. Wang, "A novel firefly algorithm for portfolio optimization problem," *IAENG International Journal of Applied Mathematics*, vol. 49, no. 1, pp. 45–50, 2019.

[13] E. Lockwood, N. H. Wasserman, and E. S. Tillema, "A case for combinatorics: A research commentary," *The Journal of Mathematical Behavior*, vol. 59, p. 100783, 2020.

[14] B. E. Sagan, *Combinatorics: The art of counting*. American Mathematical Soc., 2020, vol. 210.

[15] A. Laaksonen, *Counting Tilings*. Helsinki, Finland: Springer, 2017, pp. 74–75. [Online]. Available: https://doi.org/10.1007/978-3-319-72547-5

[16] MAXimal. (2008) Profile dynamics. parquet problem. [Online]. Available: https://e-maxx.ru/algo/profile_dynamics

[17] E-OlympOJ. (2011) Rectangle. [Online]. Available: https://www.e-olymp.com/en/problems/1478

[18] D. Joyner, *Elements, My Dear Watson*, ser. Adventures in Group Theory. Johns Hopkins University Press, 2008, pp. 7–12. [Online]. Available: https://books.google.co.id/books?id=iM0fco-_Ri8C

[19] W. Jiang, "Solving double counting problems with dynamic programming algorithm," *Journal of Physics: Conference Series*, vol. 1746, no. 1, p. 012058, jan 2021. [Online]. Available: https://doi.org/10.1088/1742-6596/1746/1/012058

[20] S. Halim and F. Halim, *Java BigInteger Class*. Lulu.com, 2013, no. v. 3, pp. 198–199. [Online]. Available: https://books.google.co.id/books?id=vUc-nwEACAAJ

[21] S. Kolodyazhnyy. (2020) The incomplete history of e-olymp. [Online]. Available: https://blog.eolymp.com/en/posts/the-incomplete-history-of-eolymp/

[22] ——. (2021) How e-olymp tests submissions. [Online]. Available: https://blog.eolymp.com/en/posts/online-judging/

[23] Hwang, *Advanced Computer Architecture, 2E*, ser. McGraw-Hill computer science series. McGraw-Hill Education (India) Pvt Limited, 2011. [Online]. Available: https://books.google.co.id/books?id=m4VFXr6qjroC

[24] S. S. Mopuri. (2018) System design — online judge with data modelling. [Online]. Available: https://medium.com/@saisandeepmopuri/system-design-online-judge-with-data-modelling-40cb2b53bfeb

[25] X. X. Qin Zhang, Xinyu Li and J. Ou, "Design of experimental platform of data structure based on online judge," *Journal of Chemical and Pharmaceutical Research*, vol. 7, no. 3, pp. 241–2421, 2015.