Analysis of Four-Large-Numbers Selection in the Countdown Numbers Round and Its Solvability

Syahrizal Bani Khairan - 13523063[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]syahrizal.khairan@gmail.com, 13523063@std.stei.itb.ac.id

***Abstract*—The Numbers Round in the British television show Countdown is an interesting mathematical puzzle where players use a set of random numbers and arithmetic operations to reach a target value. This study focuses on the selection strategy involving "four large numbers" in the game and its impact on solvability. Using principles of discrete mathematics, we analyze the probability distribution of solvable targets, the combinatorial structure of number arrangements, and the complexity of achieving exact solutions. The findings contribute to a deeper understanding of strategic selection in mathematical games, providing insights into problem-solving and optimization techniques.**

***Keywords*—Countdown numbers round, combinatorical analysis, brute force.**

I. Introduction

The British television show *Countdown* is a mathematically intriguing game. In the show, two contestants play two types of round, called the letters and numbers round. This paper will consider a specific strategy in the numbers round.

In the numbers round, contestants try to reach a randomly generated three-digit figure by using basic arithmetical operations on a given selection of numbers. A contestant has some control over the selection of numbers. The numbers are divided into two groups: 20 “small numbers” consisting two of each of 1 to 10, and 4 “large numbers” consisting of the numbers 25, 50, 75, and 100. The contestant may specify how many “large numbers” are to be used, from none to all four, and then the remaining “small numbers” are picked for a total of six numbers.

Fig. 1 Numbers round board. A target of 200 is shown with the randomly picked numbers below it.

The contestants only ever has control over how many large numbers they desire. The numbers are picked randomly and the three-digit figure is also randomly generated. A contestant may have a preference as to how many large numbers to pick, each with its own strategy. One such selection is the four large selection.

By picking four large, the randomness in the selection is reduced as there are only four large numbers. The contestants effectively always know what four of the six numbers selection are. This selection has relatively smaller number of possible combination of selection.

Although this selection is relatively predictable, operating large numbers can be unwieldy and harder to fine tune in the sense that it is harder to reach a specific target. The strategy surrounding four large is deemed as being more mechanical [1].

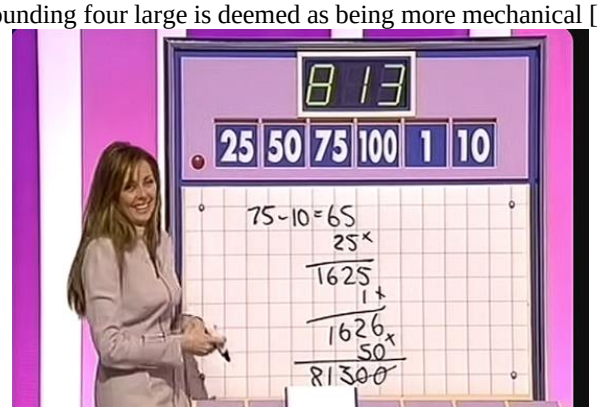

Fig. 2 A numbers round with four large selection. A mathematical calculation, demonstrated by presenter Carol Vorderman, is shown where the target 813 is reached by using all six given numbers (the number 100 is used in the last step as division). Image source: Channel 4.

Also of theoretical interest is the solvability of numbers round puzzles. There may be targets with which no possible numbers combination can provide an exact solution, or targets that require specific small numbers.

Another thing to note is that if no contestants found an exact solution, scores can still be given to whoever found a solution that is closest to the target. One may compare the viability of different selection strategy on each particular target by comparing the average of closest solutions for every possible selection, though no comparison will be made in this paper.

II. Methodology

In a four large selection, the remaining two numbers are randomly picked from 20 small numbers(two copies of numbers 1 to 10). This results in total possible selections of

$$\binom{20}{2} = 55\ combinations.$$

This is a relatively small amount of combinations. Picking six smalls or three large generates up to $\binom{20}{6} = 38760$ and $\binom{20}{3}\binom{4}{3} = 3420$ combinations respectively. The limited amount of possible combinations and the ability to know four of the six numbers beforehand leads to a more predictable solution. There are tricks and strategies built around solving four large [1].

Strategies may incorporate numbers that are reachable by the four large numbers. The four large numbers spans, or in other words can reach by arithmetical operations, these 73 numbers:

1, 2, 3, 4, 5, 6, 7, 9, 18, 20, 21, 23, 25, 27, 29, 30,
34, 43, 46, 47, 48, 49, 50, 51, 52, 53, 54, 57, 67, 69
71, 73, 75, 77, 79, 81, 95, 97, 98, 99, 100, 101, 102
103, 105, 121, 125, 129, 146, 147, 150, 153, 154, 173
175, 177, 197, 200, 203, 225, 250, 275, 298, 300, 302
350, 450, 500, 525, 575, 625, 675, 725.

All these numbers can be reached by using 25, 50, 75, and 100. In particular, there are 33 numbers (in the range 100-725) which are valid targets, guaranteeing an exact solution.

The span of a group of numbers can be computed by a method that will be shown. How the introduction of two additional small numbers affect the span and in turn the solvability of numbers round will be investigated.

A. Representation of Arithmetical Expression

An arithmetical expression can be represented as an expression tree.

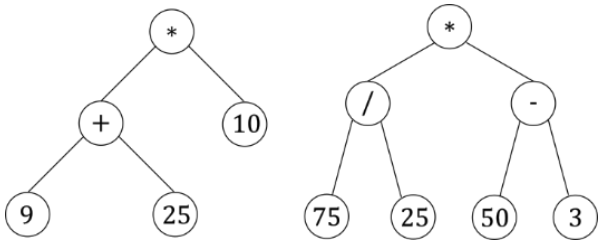

*Fig. 3 Examples of expression trees. The tree on the left evaluates to (9+25)*10 whereas the tree on the right evaluates to (75/25)*(50-3). Note the order of operation.*

In an expression tree, all the leaf nodes are numbers, whereas any other nodes are operators. All operator nodes always have two children, the left and right subtree, if the tree has only binary operators. This is the case for Countdown numbers round, where the allowed operations are addition, subtraction, multiplication, and division. Expression trees in the current context will always be a full binary tree.

B. Brute Force by Enumeration of Expression Tree

To solve a numbers round, a brute force method can be used to find a solution. Brute forces are inefficient and there are better, more efficient method. [2, 3] However, such method can still determine the existence of an exact solution. The span of a given group of numbers can be found by exhaustively generating and evaluating all possible arithmetical expression.

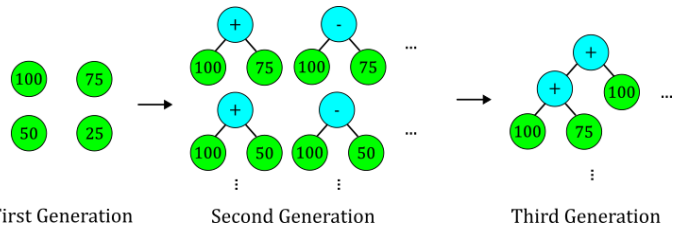

Fig. 4 Illustration of the generations of expression trees based on the four large numbers. A generation of trees are constructed by iteratively composing trees from previous generations until all leaf nodes are used. Multiple trees with the same structure is generated with differing internal nodes to enumerate all possible operations.

The brute force method starts by generating expression trees that only consists of one node representing the values of the given numbers. Then the next generation of trees are produced by pairing two trees as the subtrees of newly trees. The roots of the new trees are operators, denoting an expression involving the left subtree and right subtree as operands. To form an expression tree that uses n numbers, pair trees that uses $n-1$ numbers with trees that uses 1 number, $n-2$ numbers trees with 2 numbers trees, and etc. For each unique tree structure, permute all possible operators. Since the numbers can only be used once, the pairing of trees must check if any of their component numbers overlap.

The number of possible arithmetical expressions using n numbers is equivalent to the number of all full binary tree with n leaf nodes. C_{n-1} is the number of full binary trees with n leaves [4]. C_n is the n-th Catalan number where

$$C_n = \frac{1}{1+n}\binom{2n}{n}.$$

The amount of all expression trees that uses at most 6 numbers is then

$$\sum_{1}^{6} C_{n-1}\frac{6!}{(6-n)!}4^{n-1} = 33\ 665\ 406.$$

This amount does not account for illegal operations. Countdown prohibits non-integral division and negative integers even as intermediary result, so the amount of legal expression trees is fewer.

The span can be computed by evaluating all of the generated trees. Multiple trees may evaluate to the same value, but those do not affect the span as it only considers unique value.

III. IMPLEMENTATION

A computer program is used to find the results. The brute force technique starts by generating expression trees. The expression tree data structure is implemented as the following class:

```
Class Tree:
  def __init__(self, value, left=None, right=None):
    self.value = value  # Integer or one of ['+', '-',
'*', '/']
    self.left = left
    self.right = right

    # Bitmask. Showing which numbers are used in this
tree
    if self.left is not None and self.right is not None:
      self.component = left.component | right.component
    else:
      self.component = 0
```

The implementation is fairly standard, with the additional attribute of component. Since each given number can only be used once, tree generation must consider the numbers that are used when composing trees. The component attribute is used with bitwise operators and treated as a sequence of bits. Set bit on the rightmost bit means the rightmost given numbers is used on the expression tree.

To generate the trees, the following generator function is used.

```
def all_possible_trees(numbers):
  trees = {i: [] for i in range(1, len(numbers) + 1)}


  # Initialize trees with a single leaf node
  for i in range(len(numbers)):
    leaf = Tree(numbers[i])
    leaf.component = 1 << i
    trees[1].append(leaf)

  operators = ['+', '-', '*', '/']

  # Iteratively build trees with more leaf nodes
  for i in range(2, len(numbers) + 1):
    for j in range(1, i):
      for left_tree in trees[j]:
        for right_tree in trees[i - j]:
          if left_tree.component & right_tree.component
!= 0:
            continue
          for op in operators:
            new_tree = Tree(op, left_tree, right_tree)
            trees[i].append(new_tree)
            yield new_tree
```

The generator stores previously generated trees. Initially, the base trees that each only contains one node denoting each given numbers are created. The component attribute is set using the bitwise left shift operation. Each generation of trees has one more leaf nodes than the previous generation. The n-th generation tree is generated by pairing trees from previous generations whose number of leaves amount to n. A pairing of trees only generate a new tree if the components used in the two expression trees are different.

By using the generator, the span of a set of numbers can be computed. To save on computation time, since there are more than one set of numbers to compute the span of, only one set of all possible trees are generated. By using "template trees", the computation of span does not need to generate the expression trees every time. Regardless of the actual numbers used, the structure remain the same in all expression trees. To evaluate the trees, the actual numbers are simply substituted into each placeholder value used in the tree generation. Only a subset of the span lying in the range 100-999 is of interest as it corresponds to the range of target number.

```
def express(self, replace_placeholder={}):
  if self.left is None and self.right is None:
    # Leaf node
    if self.value in replace_placeholder.keys():
      return str(replace_placeholder[self.value])
    else:
      return str(self.value)
  else:
    if self.value in ['-', '/'] and
self.left.evaluate(replace_placeholder) <
self.right.evaluate(replace_placeholder):
      return
f"({self.right.express(replace_placeholder)}{self.value}{
self.left.express(replace_placeholder)})"
    else:
      return
f"({self.left.express(replace_placeholder)}{self.value}{s
elf.right.express(replace_placeholder)})"
```

```
def     get_numbers_span(numbers,     min=-1,     max=-1,
replace_placeholder={}, template_trees=None):
  span = {}

  if template_trees is None:
    trees = list(all_possible_trees(numbers))
  else:
    trees = template_trees

  for tree in trees:
    result = tree.evaluate(replace_placeholder)
    if result <= 0:
      continue
    if min!=-1 and max!=-1 and not (min<=result<=max):
      continue

    if result not in span.keys():
      span[result] = tree     # Prevent overriding by more
complex trees that have the same result

  sortedSpan = dict(sorted(span.items()))
  return sortedSpan
```

The implemented tree does not compute what value the expression tree evaluates to at generation. Any computation is done only when needed. This way, a "template tree" can be used effectively as any tree having the same structure but the numbers changed. The Tree class has the following evaluate method

```
def evaluate(self, replace_placeholder={}):
  if self.left is None and self.right is None:
    # Leaf node
    if self.value in replace_placeholder.keys():
      return replace_placeholder[self.value]
    return self.value

  left = self.left.evaluate(replace_placeholder)
  right = self.right.evaluate(replace_placeholder)
  if left==-1 or right==-1:
    # Invalid tree
    return -1

  if self.value=='+':
    return left + right
  if self.value=='-':
    return abs(left-right)
  if self.value=='*':
    return left * right
  if self.value=='/':
    if left<right:
      left, right = right, left
    if right==0 or (left%right!=0):
      return -1
    return left // right

  return -1
```

The method returns the value -1 if any illegal operation is found in the expression tree or its subtree. This is a consequence of the generator function not checking the actual generated tree. The method is commutative even if the operators are not. Thus the position of left subtree or right subtree does not matter.

All analysis uses these basic functions. Analysis is done using a jupyter notebook. The span for each selections are first computed and then stored for use. The notebook used to generate the results is available on the linked repository in the appendix.

```
# Compute spans of all four large selections
print("Generating all possible trees...")
template_trees = list(all_possible_trees(large_numbers+[1,
2]))

print("Computing span...")
small_numbers_combinations   =   [(i,   i)   for   i   in
small_numbers]
small_numbers_combinations                                +=
list(itertools.combinations(small_numbers, 2))

for small1, small2 in small_numbers_combinations:
```

```
  if
os.path.exists(f'data/span/{small1}_{small2}_span.txt'):
    continue

  replace_dict = {1: small1, 2: small2}
  print(f"Computing span for {small1}, {small2} ...")

  span = get_numbers_span(large_numbers+[small1, small2],
min=100,                                          max=999,
replace_placeholder=replace_dict,template_trees=template_
trees)

  res = []
  for v, t in span.items():
    res.append([v, t.express(replace_dict)])
  res = np.array(res)
   np.savetxt(f'data/span/{small1}_{small2}_span.txt',
res, fmt='%s')
```

The spans are computed for each selection by substituting the different small numbers into the "template trees". This computation is time-intensive so the result is stored in a text file.

IV. Results

A. Selection Solvability

By computing the spans of all four large selections, the amount of targets that has exact solutions for each particular selection can be found. This is computationally feasible because the small number of four large selections.

Table I. Table of all four large selection along with the number of targets that are reachable by the solution.

Small Numbers 1	Small Numbers 2	Reachable Targets	Small Numbers 1	Small Numbers 2	Reachable Targets
1	1	325	3	9	846
2	2	545	3	10	848
1	2	580	4	8	852
4	4	680	5	6	852
1	3	682	7	10	854
5	5	684	5	8	859
10	10	707	3	7	860
3	3	710	6	10	864
1	4	717	7	8	865
5	10	738	4	9	867
7	7	742	8	10	867
2	4	755	6	8	868
1	5	760	7	9	869
2	5	771	5	9	871
1	7	773	9	10	871
2	3	774	6	9	872
1	6	789	6	7	873
6	6	790	3	8	876
8	8	790	8	9	886
2	10	795	5	6	852
2	6	808	7	10	854
1	8	812	5	8	859
9	9	816	3	7	860
3	5	816	6	10	864
3	4	817	7	8	865
1	10	828	4	9	867
4	7	828	8	10	867
2	7	829	6	8	868
4	5	829	7	9	869
3	6	832	5	9	871
2	8	833	9	10	871
1	9	834	6	9	872
5	7	837	6	7	873
4	10	838	3	8	876
4	6	839	8	9	886
2	9	842			

Notice that the selection with (1, 1) small numbers has the fewest number of exact solves with only 325 targets. On average, a four large selection can exactly solves 798 targets. Selections other than the (1, 1) selection can exactly solve over half of valid targets.

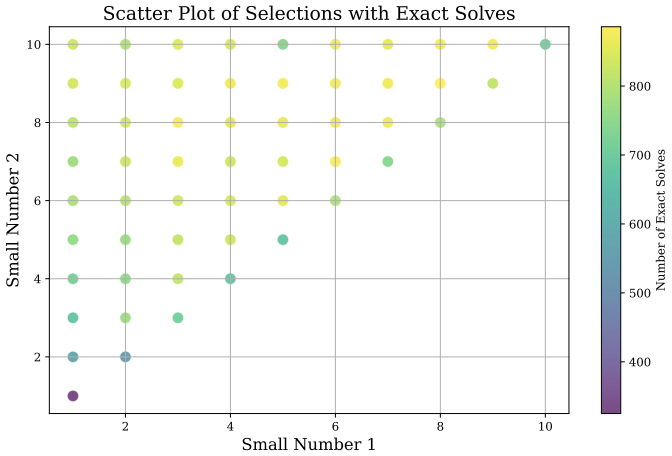

Fig. 5 Scatter plot showing the variation between combination of small numbers.

B. Target Solvability

Now the solvability for each target will be considered. It is possible that a certain target does not have an exact solution. In that case, the closest solution is considered.

It is found that the target that has the least amount of selection with which an exact solution can be found is the target 839 with only 22 selections that has the target in their span. There are 203 out of 900 targets that are always exactly solvable on all 55 four large selection. These targets are shown in the list below.

100, 101, 102, 103, 104, 105, 106, 107, 108, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 133, 134, 135, 136, 137, 138, 139, 140, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 160, 162, 167, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 210, 211, 222, 223, 225, 226, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 258, 267, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 292, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 308, 310, 322, 324, 325, 326, 328, 336, 342, 344, 346, 347, 348, 350, 352, 353, 354, 356, 357, 364, 372, 375, 376, 378, 396, 397, 399, 400, 403, 404, 406, 425, 446, 450, 453, 454, 456, 475, 492, 496, 500, 504, 525, 550, 575, 600, 624, 625, 650, 675, 700, 725, 750, 775, 800, 825, 850, 900, 925, 938, 950, 975

However, Countdown numbers rounds still reward non-exact solution if no contestant found an exact one, in which case the solution that is closest to the target wins however. The solvability of a target in a selection strategy can be described as the average of closest solutions for all selections. This metric directly translates to point-scoring ability. The following heatmap show the average of solutions closeness for each 900 targets.

Fig. 6 Heatmap of average difference of best solution across all four large selections for each target 100-999. Targets that are always exactly solvable has a closeness value of 0.

Closeness is measured as the difference between closest solution to the target. Exact solution has closeness of 0.

The heatmap above shows that smaller targets often have more exact solves. Also notice that the heatmap is divided into four bands, separated by targets that are close to ending with 25, 50, or 75. This is what was meant by four large selection being harder to fine tune.

Having more large numbers makes reaching higher targets easier, but the introduction of small numbers offers minute adjustment that widen the span of the numbers. Four large selection can easily reach targets that are multiples of 25 or within a small range around those numbers. Targets that are not close to multiples of 25 still have exact solutions, but only on slightly fewer selections.

C. Computer Simulation

A more direct way of measuring the viability of four large selection is by simulating many countdown numbers rounds. The simulation is done by randomly generating the target and small numbers, and calculate the score using the best solution possible. Prerviously, the spans of all selections are already calculated. Scoring is done based on whether the target is within the selection's span or whatever closest number is in the span.

In Countdown, an exact solution gives 10 points. Inexact solution gives 10 points subtracted by the difference between the solution and the target. A computer simulation produced the following result:

```python
# Simulation
n = 1000000
score_sum = 0
for _ in range(n):
    target = np.random.randint(100, 1000)
    selection = np.random.choice(small_numbers*2, 2, replace=False)
    selection.sort()
    selection = tuple(selection)

    c = 0
    while target-c not in selection_span[selection] and target+c not in selection_span[selection]:
        c += 1
    score_sum += 10 - c

average_score = score_sum / n
print(f"Four large selection average score : {average_score}")
```

```
✓ 1m 5.6s                                                    Python
Four large selection average score : 9.878865
```

Fig. 7 Python code that simulates a certain amount of numbers round. The average score obtained by picking four large is estimated to be approximately 9.9.

V. Reflection and Discussion

So far, this paper only discusses the solvability of four-large selection in numbers rounds without much comparison to other selection strategy. It is not possible to say whether four large is specifically better than others. Perhaps more research can be done on this matter.

Although algorithmical complexity is not the main interest in this paper, the code that is used to produce the result is far from perfect. There are many cases where the code can be improved which may improve computation time. Computing the span of 55 selections has taken roughly one hour of time. Generating all expression trees took about one minute. However, finding a solution for a solvable puzzle takes a surprisingly reasonable amount of time.

For example, the generation of trees produce expressions that are not legal in the Countdown numbers round such as non-integral division. A massive possible optimization is gained by utilizing the commutativity of operations. Currently, the generator simply permutes the leaf nodes, resulting in duplicate expression where only the operands got switched around operators such as addition. These problems does not affect the results in any way.

Modelling arithmetical expression as binary tree turned out to be better and is more natural. A previous attempt tries to use a list of numbers and operators to represent an expression. To enumerate all expressions, the members of the list are permuted. Then the expression can be evaluated by using a stack. This approach is abandoned halfway due to slow performance.

VI. Conclusion

Four-large selection is a sound strategy in Countdown numbers round. The majority of target and selection combination has either exact or close solutions.

Whether a solution is obvious to the human contestant is not considered in this paper. Comparison to other selection strategy is due.

VI. Appendix

All code that is used to generate the results in this paper is hosted on a GitHub repository. The raw results data is stored under the data subdirectory. The data can be reproduced by running the python notebook found in the repository. At time of writing, the repository is in the state referenced by the v1.0 tag. The link to the repository is given below:

https://github.com/rizalkhairan/countdown

VII. Acknowledgment

The author would like to express my deepest gratitude to Dr. Ir. Rinaldi, M. T., for his invaluable guidance, support, and encouragement throughout the preparation of this paper. His expertise, insightful feedback, and dedication to teaching have greatly enhanced my understanding of the subject matter.

References

[1] [Online]. Available: https://countdownresources.wordpress.com/2018/10/05/4-large/. [Accessed 8 January 2025].

[2] F. Amin, "Penerapan Algoritma Brute Force pada permainan Countdown Number," 2017. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2016-2017/Makalah2017/Makalah-IF2211-2017-050.pdf. [Accessed 7 January 2025].

[3] G. Hutton, "FUNCTIONAL PEARLS The countdown problem," *Journal of Functional Programming*, 2002.

[4] R. P. Stanley, Enumerative combinatorics, vol. 2.

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Januari 2025

Syahrizal Bani Khairan 13523063