Asian MathSci League, Inc (AMSLI)
Partner: National Olympiad in Informatics Philippines (NOI.PH)
Email address: amsliphil@yahoo.com / ask@noi.ph
Contact Nos: +632-9254526 +63906-1186067

Student Copy
Grade 9/10 Session 1

Session 1: Complete Search

Let's solve some problems!

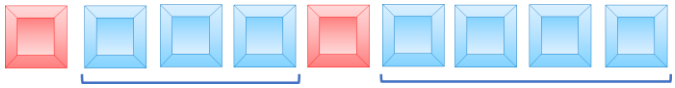
Problem 1 – A Block Flipping Game

In this game, you have a row of n red and blue blocks. There are at most 100 blocks in a row. Here is an example.

You can *flip* blocks to change their color. Flipping a red block will make it blue, and flipping a blue block will make it red. You are only allowed to flip blocks twice. You are allowed to flip the same block twice, but you must perform two flips.

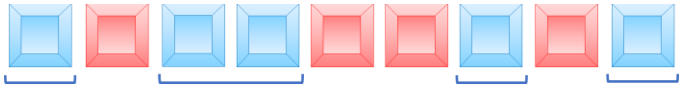
After flipping blocks, your score is the length of the longest consecutive chain of blue blocks. For example:

- Flipping the 6th and 8th blocks from the left results in the following pattern:

  There are two chains of blue blocks, indicated by the brackets. The first chain has 3 blue blocks, and the second chain has 4 blue blocks. Since the longest chain has 4 blue blocks, your score is 4.

- Flipping the first two blocks results in the following pattern:

  There are four chains of blue blocks, indicated with the brackets. The longest chain has 2 blocks, so your score is 2.

How would you write a Python program which determines your **maximum possible score**? Assume the boxes are represented by list of 0s and 1s, where zeros represent red blocks, and ones represent blue blocks.

If you try solving some examples on paper or in your head, you might come up with many observations and strategies, such as:

- There is no point in flipping a blue block only once.
- Try looking for chains of blue blocks separated by only one or two red blocks.
- The two blocks you flip must generally contribute to the same chain of blue blocks.
- If there are no red blocks, then just flip the same block twice to keep all the blocks blue.

These are good first steps for a pen-and-paper solution to the problem, and it's very natural for us to start thinking of strategies and generalizations. But implementing some or all of these strategies can get very complicated. We'll have to consider questions such as:

- Are these strategies even correct? Are we sure that there are no counterexamples to these strategies?
- How do we count existing chains of blue blocks which are "close together"?
- Which strategy/observation should we check first, and why?
- Do we even need all of these strategies or just some of them?

However, when solving a problem like this computationally, we can (and should) always start with a very easy solution: **try all possible options!** We can loop through every possible choice of which blocks to flip, and simply keep track of the maximum score.

```
highScore = 0
for block1 in range(0, n):
    for block2 in range(0, n):
        highScore = max(highScore, calculateScore(block1, block2))
```

In this type of solution, we **don't need to think very hard** about the problem. The only thing we need to worry about now is implementing `calculateScore` so we can evaluate each option, but that's a lot less to worry about than trying to figure out smarter ways of picking which boxes to flip.

Key insight: Trying all possible answers is always going to yield a correct program.

This approach can be applied to many problems. This next example shows how trying all possible options can be useful even if you know of a smarter solution.

Problem 2 – Factor counting

Exercise 1. Write a Python function which takes in two integers n and k, and counts how many integers from 1 to n (inclusive) are divisible by k.

Solution. Here are two ways to approach this problem.

- **Without thinking too hard**, simply try all of the integers from 1 to n and count all of them using a variable:

```
def countDivisible(n, k):
    count = 0
    for x in range(1, n+1):
        if x % k == 0:
            count += 1
    return count
```

- Think of a formula to use so your function can figure out the answer really quickly.

Exercise 2. Try running the `countDivisible(n, k)` function for fixed $\boldsymbol{k}$ (try $\boldsymbol{k = 3}$), while plugging in consecutive values for $\boldsymbol{n}$. Can you figure out a formula to solve **Exercise** 1 without a loop?

Solution. For any n and k, the numbers divisible by k occur every k integers. Below are examples where $k = 3$.

n	1	2	3	4	5	6	7	8	9	10	11	12
Divisible by 3?	No	No	Yes	No	No	Yes	No	No	Yes	No	No	Yes
`countDivisible(n, 3)`	0	0	1	1	1	2	2	2	3	3	3	4

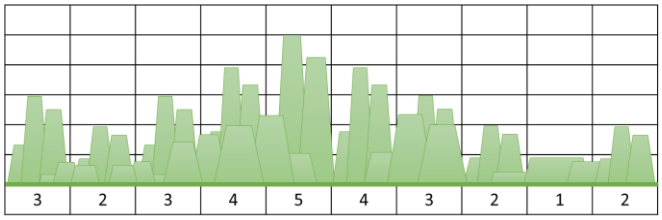
From this observation, we can deduce that we can implement `countDivisible(n, k)` as

```
def countDivisible(n, k):
    return n // k
```

Key insight. A program which tries all possible answers is always going to be correct (although it might be slower). You can use such a program as a concrete starting point, leading you to come up with a smarter, faster solution. But sometimes, simply trying all possible answers is already good enough (and you don't need to think too hard).

Problem 3 – Mountain Ranges

You have a list `L` of positive integers, representing heights of a mountain range.



The difference between consecutive elements in the list is always one. Write a Python program to count how many peaks the mountain range has. A peak is an element `L[i]` in the list where both `L[i+1]` and `L[i-1]` are less than `L[i]`. The first and last elements of the list are not peaks.

For example, in the image above, there is only one peak, the point at which the height of the mountain range is 5.

Exercise 3. The following solution tries to count the peaks by checking all possible locations. Is this solution correct? Why or why not?

```python
def countPeaks(L):
    count = 0
    for i in range(len(L)):
        if i > 1 and i < len(L) and L[i-1] < L[i] and L[i+1] < L[i]:
            count += 1
    return count
```

Key insight. When trying all possible answers, make sure you are indeed trying all possible answers. You're only guaranteed to get the correct answer if you try all possible answers.

What we've just seen in the previous three problems is a technique known as **complete search**. Complete search is a programming problem solving technique where a program checks all possible solutions to a problem in order to solve it. This approach is also called **brute force**, **trial-and-error** or **guess-and-check**.

As we've seen in the previous three problems, complete search is an easy approach which can be useful when applied correctly, even if a "smarter" solution also exists.

The Anatomy of Complete Search - Warehouse Inspection

How do we apply complete search? It turns out that many programming problems can be expressed in the following form.

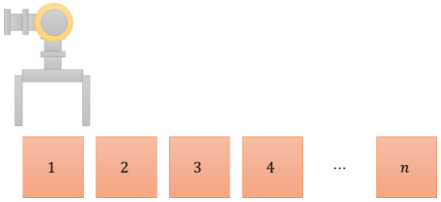
Out of these possible solutions, which one is correct?

The set of possible solutions to a problem is called the **search space**. Once you determine the search space, a complete search solution is simply:

```
For every solution S in the search space:
        Check if S is correct.
```

Searching the search space

Let's look at the following problem. A robot is used to inspect goods in a warehouse with n boxes, numbered from 1 to n. Each box contains a machine which can either be working or defective. The robot can inspect a single box at a time. You're in charge of coding simple Python programs to direct the robot.



You have a function `defective` which takes in an integer `i`. Calling `defective(i)` checks box i and returns a boolean.

- If box i contains a working machine, `defective(i)` returns `False`.
- If box i contains a defective machine, `defective(i)` returns `True`.

Exercise 4. You have received a report that one of the n machines is defective. Write a Python function which returns the integer label of the box with the defective machine.

Solution. We can simply check each box in order, from box 1 to box n, using a loop.

```
def checkBoxes():
    for i in range(1,n+1):
        if defective(i):
            return i
```

In the previous exercise, the possible solutions are the integers from 1 to n. In other words, our search space is the set of integers $\{1,2,\cdots,n\}$. In the problem, a solution i is "correct" if box i is defective.

Key insight: To create a complete search solution, we need two things:

1. **We need to be able to traverse the search space**. We need to be able to enumerate the entire search space. The easiest way to do this is using loops, but in the next session we'll see examples where the search space must be enumerated in a different way.
2. **We need a way to check if a possible solution is correct.** Implementing such a validator can vary in difficulty depending on the problem. We'll see examples of this later in this module.

Commented [VG1]: How about "can vary in difficulty"? So it doesn't sound pre-emptively discouraging

These are the main things you need to think about when crafting a complete search algorithm.

Exercise 5. You've received a report that defective machines might occur in series. Write a Python function which checks if there are defective machines in two adjacent boxes. If there are two adjacent boxes, both of which contain defective machines, return `True`. Otherwise, return `False`.

Solution. There are multiple ways to approach this problem. One is to check all pairs of machines:

```
def defectivePairs():
    for i in range(1,n+1):
        for j in range(1,n+1):
            if j == i + 1 and defective(i) and defective(j):
                return True
    return False
```

Another way is to only check the pairs $(1,2), (2,3), (3,4), \ldots, (n-1,n)$. We loop through all boxes i, where $1 \leq i < n$, and consider the pair $(i, i+1)$. Note that in the loop, we do not include the case where $i = n$.

```
def defectivePairs():
    for i in range(1,n):
            if defective(i) and defective(i+1):
                return True
    return False
```

Both of these functions solve the problem, but which one is better? The second approach has a smaller search space, and therefore runs faster than the other solution.

Key insight: Smaller search spaces are better.

Other problem types solvable with complete search

As we've seen in the problems at the start of the lesson, complete search can also be used when we are interested in the *number of correct solutions*, or if we want to find the *best solution (according to some criteria)*.

Exercise 6. You have received an order to check how many of the n machines are defective. Write a Python function which returns the number of defective machines.

Solution. Like before, we use a loop to go through all of the machines. To keep track of how many machines are defective, we use a variable which we increment every time we encounter a defective machine.

```
def countDefective():
    totalDefective = 0
    for i in range(1,n+1):
        if not inspect(i):
            totalDefective += 1
    return totalDefective
```

Note that since the search space and the method of checking the solution are the same as in Exercise 4, most of the code is similar.

Key insight: A complete search algorithm which checks if a solution exists can be easily adapted to an algorithm which *counts the number of possible solutions*.

Exercise 7. Write a Python function which returns the number of *distinct pairs* of working machines.

For example, if there are 5 boxes, and boxes 2, 3, and 5 have working machines, then the distinct pairs of working machines are $(2,5)$, $(3,5)$ and $(2,3)$. Thus, there are 3 distinct pairs.

Solution. There are many different solutions to this problem. One is to loop through all *non-distinct pairs*, since it is easy to do so using a nested loop. To only count distinct pairs, we can only consider pairs (i,j) where $i < j$.

```
def countWorkingPairs():
    numberOfWorkingPairs = 0
    for i in range(1,n+1):
        for j in range(1, n+1):
            if i < j and not defective(i) and not defective(j):
                numberOfWorkingPairs += 1
    return numberOfWorkingPairs
```

Another solution is to *only* loop through pairs (i,j) where $i < j$. This can be done by changing the loop ranges.

Commented [VG2]: How about 1 or 2 examples where complete search is applied to optimization problems too?

Commented [VG3R2]: Probably no need for new examples. Just reuse the intro problems and show how they fit into the general framework

Commented [VG4R2]: Oh I just saw exercise 10. That should suffice for an optimization example

```
def countWorkingPairs():
    numberOfWorkingPairs = 0
    for i in range(1,n+1):
        for j in range(i+1, n+1):
            if not defective(i) and not defective(j):
                numberOfWorkingPairs += 1
    return numberOfWorkingPairs
```

Exercise 8. Which one of the algorithms presented for Exercise 7 is better?

Exercise 9. You have received an order to ensure that defective machines do not occur too frequently. Write a Python function which returns an integer k, where k is the minimum number of boxes between two defective machines. (For this exercise, suppose there are at least two defective machines present.)

In this example, there are four boxes between 1 and 6, two boxes between box 1 and 4, but only one box between boxes 4 and 6. Thus, your function must return 1.

Solution. We can consider all pairs of machines and keep track of the minimum distance of pairs where both machines are defective.

```
def minimumDistance():
    minDistance = n
    for i in range(1,n+1):
        for j in range(i+1, n+1):
            if defective(i) and defective(j):
                minDistance = min(minDistance, j - i - 1)
    return minDistance
```

Key insight: A complete search algorithm which checks if a solution exists can be easily adapted to an algorithm which *finds the best solution according to some criteria*.

Systematically Traversing a Search Space - Combination Locks

Suppose you wanted to find the correct combination to open a four-digit combination lock through brute-force. Our search space is the set of all combinations from `0000` to `9999`. Here are two approaches to traversing the entire search space.

Approach 1. One is by simply going through all integers from 0 to 9999.

```
for combination in range(0, 10000):
        print(combination)
```

Approach 2. Another way is to use nested loops to consider each digit in sequence.

Commented [VG5]: Define this term at the beginning too. Technical difference between "complete search" and "brute force" not important at this point. Just introduce as synonyms. Incidentally, students who have done a bit of competitive math may be familiar with "trial-and-error" so it might be a good idea to say that complete search and brute force are just new terms for the same idea.

```
for a in range(0,10):
    for b in range(0, 10):
        for c in range(0,10):
            for d in range(0,10):
                print(a, b, c, d)
```

Exercise 10. Compare the output of the two approaches by running each program. Which approach do you think is better for solving each of the following problems?

a) Find the correct combination to a 4 digit lock if you know that the first digit is less than 3.
b) Find the correct combination to a 4 digit lock if you know that the digits' sum does not exceed 16.
c) Find the correct combination to a 4 digit lock if you know that the correct combination must be between 4270 and 5367.

Answer. Since both approaches traverse the same search space, both approaches will still answer the problem. For any of the problems (a) to (c), we can disregard the additional information and just check every combination from `0000` to 9999 anyway. But to make the search space smaller, depending on the problem, one approach could be easier to adapt than the other one.

a) Find the correct combination to a 4 digit lock where the first digit is less than 3.

To reduce the search space for this problem, we adjust the ranges of the loops. Either approach can be adapted for this problem.

```
for combination in range(0, 3000):
        print(combination)
```

```
for a in range(0,3):
    for b in range(0, 10):
        for c in range(0,10):
            for d in range(0,10):
                print(a, b, c, d)
```

b) Find the correct combination to a 4 digit lock if you know that the digits' sum does not exceed 16.

For this problem, since the search space is dependent on the individual digits, this is a sign that a traversal which considers each digit in turn would make it easier to reduce the search space.

We can stop considering other digits if the sum of currently chosen digits already exceeds 16.

```
for a in range(0,10):
    for b in range(0, 10):
        if a + b > 16:
            break
        for c in range(0,10):
            if a + b + c > 16:
                break
            for d in range(0,10):
                    if a + b + c + d > 16:
                        print(a,b,c,d)
```

Notice that after a and b are decided, if $a + b > 16$, then we don't check further.

c) Find the correct combination to a 4 digit lock if you know that the correct combination must be between 4270 and 5367.

To reduce the search space for this problem, it is more natural to use the first approach, since the search space cannot be reduced on a digit-by-digit basis.

```
For combination in range(4270, 5367):
        print(combination)
```

In all of the examples, we're able to reduce the search space based on information we already know about the problem. In example (b), by breaking up the traversal into several steps, we can use information about the values we already picked to reduce the size of the search space.

Key insight. Complete search relies on a *systematic approach* to traversing the search space. A different traversal of the search space can take advantage of additional information provided by the problem, allowing the search space to be reduced.

Exercise 11. Let $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{n}$ be fixed positive integers less than 50. Write a Python program to count how many triples of positive integers $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ satisfy $\boldsymbol{ax} + \boldsymbol{by} + \boldsymbol{cz} = \boldsymbol{n}$.

Solution. We are not explicitly given a finite search space for (x, y, z). Since we know that n is at most 50, and all integers involved are positive, we can reasonably say that we only need to check triples (x, y, z) where each integer is between 1 and 50 inclusive.

```
count = 0
for x in range(1,51):
    for y in range(1,51):
        for z in range(1,51):
            if a*x + b*y+ c*z == n:
                count += 1
print(count)
```

However, we can make the search space even smaller than this. In our traversal of the search space, we choose values for x, y, and then z, in that order. If the values we chose so far cause $ax + by + cz$ to exceed n, we can cut off the search there, since no choice of the remaining values will yield a valid solution.

```
count = 0
for x in range(1,51):
    if a*x > n:
        break
    for y in range(1,51):
        if a*x + b*y > n:
            break
        for z in range(1,51):
            if a*x + b*y + c*z > n:
                break
            elif a*x + b*y + c*z == n:
                count += 1
print(count)
```

Exercise 12. We can implement search space reduction by modifying the ranges of the loops. For example, instead of the following:

```
for x in range(1,51):
    if a*x > n:
        break
    # ...
```

We can use the following loop, obtained by rearranging the inequality $ax > n$.

```
for x in range(1, n // a):
    # ...
```

Rewrite the solution in Exercise 11 so that all search space reduction is done in this manner.

Combinatorial Explosion - Solving Puzzles with Complete Search

Even with a basic understanding of complete search, you can solve many small puzzles computationally!

In solving these problems, implementing a verifier which checks if a solution is correct will generally be more complicated than the examples we've encountered so far.

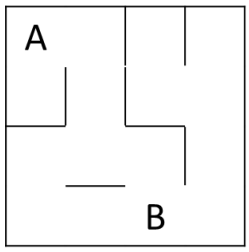
Exercise 13. Consider solving the following $\mathbf{4} \times \mathbf{4}$ Sudoku puzzle by writing a Python program.

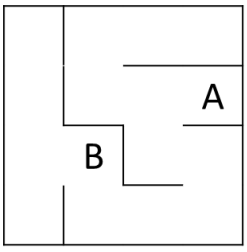
3		4	
	1		2
	4		3
2		1	

In a 4×4 Sudoku puzzle, every row, column and 2×2 box (indicated with thick borders) must be filled with the numbers 1 to 4 without repetitions.

Without reducing the search space, how big is the search space of this problem?

Exercise 14. Suppose your program takes as input, a $\mathbf{4} \times \mathbf{4}$ maze such as any of the ones pictured below, as well as two locations in the maze, $\boldsymbol{A}$ and $\boldsymbol{B}$. In one move, you can move one space up, down, left or right.

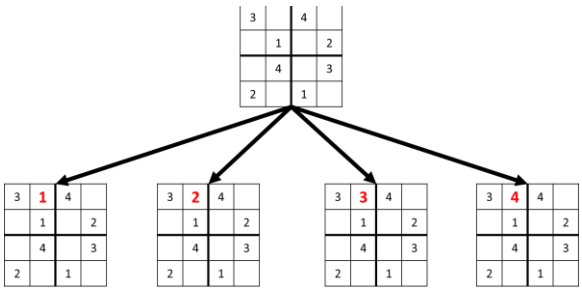




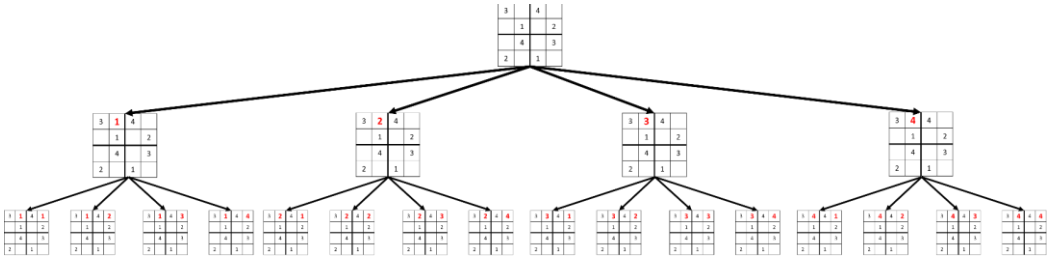
a) How would you solve the problem: "Is it possible to go from point A to point B in only 5 moves?" What would the search space be? How big is this search space?
b) How would you solve the problem: "Is it possible to go from point A to point B if you can only move down and to the right?" What would the search space be?

Answer for (b): Using loops, a quick solution would be to only consider sequences of at most 8 moves, since after moving down or to the right 8 times, you're guaranteed to no longer be able to move anywhere else (since the maze is only 4×4. It is possible to reduce the search space further, by only considering move sequences where you can move at most 4 times downward and 4 times to the right, however, enumerating this search space using loops is much more tedious.

Notice that in the previous exercises, even for a relatively small puzzle, the search space can get really large. Even if computers are fast, they are still limited and if the search space grows too large relative to the input size, even fast computers become helpless. For example, when applying complete search to the 4×4 Sudoku puzzle, a program would try all possible numbers for the first empty cell.



From each of these branches, we then have to consider choices for the next cell...



Due to the nature of the problem, the size of our search space *blows up* quite quickly. Even a small increase would dramatically increase the size of the problem.

- There are 288 possible 4×4 Sudoku grids.
- There are $6{,}670{,}903{,}752{,}021{,}072{,}936{,}960$ possible 9×9 Sudoku grids.
- There are an estimated 5.96×10^{98} possible 16×16 Sudoku grids.

This rapid (explosive) growth in the number of combinations possible in a problem's search space is called a **combinatorial explosion**.

To end this session, we'll check out another concrete example of a simple puzzle which results in a combinatorial explosion. Watch the following subtitled video by *Miraikan: National Museum of Emerging Science and Innovation*: https://www.youtube.com/watch?v=Q4gTV4r0zRs

This video demonstrates combinatorial explosion with the problem of counting paths in a grid.

Complete search is easy and always correct, but as we've seen in these last examples, it has its limits. While complete search is a good starting point, in order to solve problems faster, we need to be smarter. That's what the rest of the sessions are for: how can we do better than brute-force?