9. Search problems

Many algorithmic problems can be thought of as search problems, where we have a set of possible solutions and the goal is to find an optimal solution or to count the number of desired solutions.

A straightforward way of solving a search problem is a *brute force* search that goes through all possible solutions one by one. This method always produces the correct answer, but the search may take too much time to be practical if the number of solutions is big.

If the goal is to find an optimal solutions, there may be a *greedy algorithm* that constructs a solution efficiently following a certain logic without going through all possibilities. However, a greedy algorithm is not always guaranteed to find an optimal solution.

A more advanced method for search problems is *dynamic programming*, which can be used both for finding an optimal solution and for counting solutions. Dynamic programming is the topic of the next chapter.

Iterating solutions

A brute force algorithm for a search problem iterates through all possible solutions one by one. During the iteration, the algorithm can select solutions that satisfy certain conditions or are optimal in some way.

A brute force algorithm can be implemented to construct all solutions by combining given elements in a certain way. In the following we take a look at three common types of cases:

Sequences

The input consists of n elements and we want all sequences of m elements. There are n^m such sequences.

For example, if the elements are $[1,2,3]$ and $m = 2$, the sequences are $[1,1]$, $[1,2]$, $[1,3]$, $[2,1]$, $[2,2]$, $[2,3]$, $[3,1]$, $[3,2]$ and $[3,3]$.

In Python, sequences can be formed using the function `product` in the module `itertools`. The function parameters are a list of the elements and the length of the sequence as a parameter `repeat`. The function can be used as follows:

```
import itertools

for repetition in itertools.product([1, 2, 3], repeat=2):
    print(repetition)
```

```
(1, 1)
(1, 2)
(1, 3)
(2, 1)
(2, 2)
(2, 3)
(3, 1)
(3, 2)
(3, 3)
```

Permutations

The input consists of n elements and we want all permutations or orderings of the elements. The number of permutations is $n!$.

For example, if the elements are $[1, 2, 3]$, the permutations are $[1, 2, 3]$, $[1, 3, 2]$, $[2, 1, 3]$, $[2, 3, 1]$, $[3, 1, 2]$ and $[3, 2, 1]$.

In Python, the permutations of a list can be formed using the function `permutations` in the module `itertools` as follows:

```
import itertools

for permutation in itertools.permutations([1, 2, 3]):
    print(permutation)
```

The code outputs:

```
(1, 2, 3)
(1, 3, 2)
```

```
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

Combinations

The input consists of n elements and we want all combinations or subsets consisting of m elements. The number of combinations is $\binom{n}{m}$.

For example, if the elements are $[1, 2, 3, 4]$ and $m = 2$, the combinations are $[1, 2]$, $[1, 3]$, $[1, 4]$, $[2, 3]$, $[2, 4]$ and $[3, 4]$.

In Python, combinations can be formed using the function `combinations` in the module `itertools` as follows:

```
import itertools

for combination in itertools.combinations([1, 2, 3, 4], 2):
    print(combination)
```

The code outputs:

```
(1, 2)
(1, 3)
(1, 4)
(2, 3)
(2, 4)
(3, 4)
```

Example: Orderings

Task

You are given a positive integer n, and your task is to count how many ways the elements $1, 2, \ldots, n$ can be ordered so that no pair of adjacent elements are consecutive numbers, i.e., numbers that differ by 1.

> For example, when $n = 4$, the desired answer is 2, because the possible ways are $[2, 4, 1, 3]$ and $[3, 1, 4, 2]$.

The task can be solved using an algorithm that goes through all permutations of the elements $1 \ldots n$ and checks for each permutation if it satisfies the condition of having no adjacent pairs of consecutive numbers. If the condition is satified, the permutation is one of the possible solutions and a counter of solutions is incremented by one.

The algorithm can be implemented as a function `count_orders` as follows:

```
import itertools

def count_orders(n):
    items = range(1, n + 1)
    count = 0

    for order in itertools.permutations(items):
        if valid_order(order):
            count += 1

    return count

def valid_order(order):
    for i in range(len(order) - 1):
        if abs(order[i] - order[i + 1]) == 1:
            return False
    return True
```

The function `count_orders` iterates through all permutations of the numbers $1 \ldots n$. For each permutation, it calls the function `valid_order` that checks if the ordering is satisfactory. The solution count is computed in the variable `count`, which the function returns at the end.

For example, `count_orders(4)` returns the desired answer 2. To see more clearly how the function obtains the answer, we can modify it to print out all valid solutions:

```
    for order in itertools.permutations(items):
        if valid_order(order):
```

```
            print(order)
            count += 1
```

With the modification, the call `count_orders(4)` prints out the following lines:

```
(2, 4, 1, 3)
(3, 1, 4, 2)
```

The time complexity of the algorithm is $O(n!n)$, because it iterates through all permutations and the checking of each permutation takes $O(n)$ time. Since $n!$ grows rapidly, the algorithm works efficiently only when n is small.

The following table shows the number of valid solutions and the computation time on the test computer as n grows:

List size n	Number of solutions	Execution time
1	1	0.00 s
2	0	0.00 s
3	0	0.00 s
4	2	0.00 s
5	14	0.00 s
6	90	0.00 s
7	646	0.00 s
8	5242	0.03 s
9	47622	0.22 s
10	479306	2.31 s

When n is 10 or bigger, the computation takes a lot of time since the number of permutations $n!$ is big.

Example: Balanced parenthesis

> **Task**

A balanced parenthesis sequence is a string consisting of the characters `(` and `)` so that it corresponds to a valid mathematical expression. Your task is to count the number of balanced parenthesis sequences of length n.

For example, when $n = 6$, the desired answer is 5, because the possible balanced parenthesis sequences are `((()))`, `(()())`, `()(())`, `(())()` and `()()()`.

The task can be solved using an algorithm that generates all possible sequences of the parenthesis symbols `(` and `)` and checks which of them are balanced. The algorithm can be implemented as a function `count_sequences` as follows:

```
import itertools

def count_sequences(n):
    count = 0
    for sequence in itertools.product("()", repeat=n):
        if valid_sequence(sequence):
            count += 1
    return count

def valid_sequence(sequence):
    depth = 0
    for bracket in sequence:
        if bracket == "(":
            depth += 1
        if bracket == ")":
            depth -= 1
        if depth < 0:
            return False
    return depth == 0
```

The function `count_sequences` iterates through all parenthesis sequences using the function `product`.

The function `valid_sequence` checks if a sequence is properly balanced. The function goes through the symbols from left to right and maintains the parenthesis depth in a variable `depth`. The sequence is balanced if the depth is never negative and is zero at the end.

For example, the call `count_sequences(6)` returns the desired answer 5. Notice that a parenthesis sequence can be balanced only when n is even.

The time complexity of the algorithm is $O(2^n n)$, because it goes through 2^n parenthesis sequences and the checking of each sequence takes $O(n)$ time. The following table shows the number of valid solutions and the execution time of the code on the test computer:

Length n	Number of solutions	Execution time
2	1	0.00 s
4	2	0.00 s
6	5	0.00 s
8	14	0.00 s
10	42	0.00 s
12	132	0.00 s
14	429	0.01 s
16	1430	0.04 s
18	4862	0.17 s
20	16796	0.69 s

Compared to the preceding example of counting permutations, this algorithm can handle slightly larger values of n. With permutations, the limit of efficient computations is $n = 10$, but here we go up to $n = 20$. The reason is that 2^n grows slower than $n!$. However, this algorithm too slows down a lot beyond $n = 20$.

Speeding up the search

The two preceding examples implement the search by generating all solutions and checking each solution separately. This works correctly but does a lot of unnecessary work in processing invalid solutions.

Let us consider the problem of counting balanced parenthesis sequences. Many of the sequences generated by algorithm are trivially invalid. For example, no sequence beginning with the closing parenthesis `)` is valid, which already eliminates half of the sequences. The actual number of valid sequences is much smaller still. For example, when $n = 20$, the algorithm goes through $2^{20} = 1048576$ sequences, but only 16796 or about 1 out of 60 of them are valid.

There is another factor slowing down algorithms like this: Even though we only need to *count* the solutions, the algorithm actually *constructs* all solutions. This can be wasted work, because the algorithm only returns the count and not the sequences themselves.

The following is a more efficient algorithm for counting balanced parenthesis sequences:

```
def count_sequences(n, d=0):
    if d < 0 or d > n:
        return 0
    if n == 0:
        return 1
    return count_sequences(n - 1, d + 1) + \
           count_sequences(n - 1, d - 1)
```

This function does not use the module `itertools` to construct solutions, but iterates through the solutions directly using recursion. The idea is that each recursive call adds a new parenthesis to the end of the sequence. The parameter `n` tells how many additional symbols are still needed, and the parameter `d` maintains the parenthesis depth of the current sequence, which is 0 in the beginning when the sequence is empty. In each step, there are two possibilities: adding an opening parenthesis `(` increases the depth and adding a closing parenthesis `)` decreases the depth.

In each step, the algorithm checks if the current sequence can still lead to a valid solution. This is implemented using two conditions: If the depth becomes negative (`d < 0`) or if the depth is greater than the number of remaining symbols (`d > n`), the current sequence cannot be extended into a valid solution and the function returns `0`. If all symbols have been added (`n == 0`), one valid sequence has been completed, and the algorithm returns `1`.

Notice that the algorithm does not keep track of the actual current sequence but only the number of missing symbols and the parenthesis depth. For example, when counting balanced parenthesis sequences of length 8 and the current sequence is `(()`, the corresponding function call is `count_sequences(5, 1)`.

Implemented this way, the algorithm is much faster than the original algorithm. For example, when $n = 20$, the original algorithm runs in 0.69 seconds, while the improved algorithm runs in 0.02 seconds. With the improved algorithm we can extend the computation to larger values of n:

Length n	Number of solutions	Execution time
20	16796	0.02 s
22	58786	0.07 s
24	208012	0.23 s
26	742900	0.80 s

However, even the improved algorithm starts to slow down when $n = 26$ or greater. This is typical for algorithms based on brute force: There are ways of making the algorithm more efficient, but they lead to only a slight increase in the size of inputs that can be handled effciently.

We return to the problem of counting balanced parenthesis sequences in the next chapter, where we obtain a solution based on dynamic programming. With the dynamic programming algorithm even the case $n = 100$ is very fast.

Greedy algorithms

A *greedy algorithm* is an efficient method for finding an optimal solution without going through all possible solutions. Greedy algorithms are often simple but their correctness can be difficult to justify.

Let us consider the following example task:

Task

You have access to an unbounded number of coins with values 1, 2 and 5. What is the minimum number of coins needed to form the sum x?

For example, when $x = 13$, the desired answer is 4, because we can choose the coins $1, 2, 5, 5$ and there is no solution with fewer coins.

We can solve the task with brute force by going through all possible combinations of coins. The following function `find_coins` returns the smallest number of coins forming the sum x.

```
def find_coins(x):
    solutions = [[]]
```

```
    for solution in solutions:
        if sum(solution) == x:
            return len(solution)
        for coin in [1, 2, 5]:
            solutions.append(solution + [coin])
```

The function construct combinations of coins into the list `solutions` in increasing order of the number of coins. When we find the first combination summing up to x, we return the size of the combination.

With the example $x = 13$, the function first goes through combinations of 1, 2 and 3 coins. None of them achieve the desired sum 13. Then the function goes through combinations of 4 coins and discovers the solution $[1, 2, 5, 5]$ that sums up to 13. This allows the algorithm to conclude that 4 is the smallest number of coins needed and it returns this as an answer.

This is a correct algorithm but it needs a lot of time when the number of coins needed is large. The problem is that the function goes through a large number of solutions with the wrong sum. For example, the case $x = 100$ is already too large for the algorithm to handle efficiently.

The problem can be solved efficiently using a greedy algorithm that goes through the coin values from the largest to the smallest and adds as many coins of each value as possible. For example, when $x = 13$, the algorithm starts with the value 5 and adds two coins. Then the algorithm adds one coin of value 2 and one more coin of value 1. The result is the optimal combination $[1, 2, 5, 5]$.

The following code implements the algorithm:

```
def find_coins(x):
    count = 0
    for coin in [5, 2, 1]:
        while coin <= x:
            x -= coin
            count += 1
    return count
```

This algorithm is efficient because it does not iterate through all solutions but constructs an optimal solution directly. For example, the algorithm is very fast in the case $x = 12345$, where the optimal solution contains 2469 coins. It would not be possible to compute this with the brute force algorithm, which cannot handle

even the case $x = 100$.

Why the algorithm is correct?

When designing a greedy algorithm, it can be difficult to justify that the algorithm computes an optimal solution in all cases. A greedy algorithm does not go through all solutions and there is a risk that the constructed solution is not optimal.

In the coin task, how do we know that the greedy algorithm constructs a solution with the smallest number of coins?

The strategy of adding coins of largest possible value may seem valid at first sight, but this is not necessarily so. If we change the task slightly so that the coin values are 1, 4 and 5, the greedy algorithm becomes incorrect. In the case $x = 13$, the optimal solution is $[4, 4, 5]$ (three coins), but the greedy algorithm constructs the solution $[1, 1, 1, 5, 5]$ (five coins). In this case, adding the second coin of value 5 was an error by the greedy algorithm. Thus the correctness of the greedy algorithm depends on the set of coin values available.

Let us next prove that the greedy algorithm finds an optimal solution when the coin values are 1, 2 and 5. Assume that there is a situation, where the current sum is still at least 5 short of the target sum, but it is incorrect to add more coins of value 5. Then the remaining sum must be achieved using the coins of value 1 and 2 and at least three more coins is needed. But no set of three coins of value 1 or 2 is optimal:

- If the coins are $1, 1, 1$, a better solution is $1, 2$.
- If the coins are $1, 1, 2$, a better solution is $2, 2$.
- If the coins are $1, 2, 2$, a better solution is 5.
- If the coins are $2, 2, 2$, a better solution is $1, 5$.

This is a contradiction, which means that adding a coin of value 5 is always correct when the remaining sum is at least 5.

We still need to consider the case when the remaining sum is less than 5 and at least 2. Now we can similarly deduce that adding a coin of value 2 is the correct choice. Otherwise the solution has at least two coins of value 1, which is inoptimal, because the two coins can be replaced with one coin of value 2. This completes the proof that the greedy algorithm finds an optimal solution when the coin values are 1, 2 and 5.

Algorithm checking

The following test program compares the solutions of the brute force and the greedy algorithms in the coin task. The goal is to find a situation, where the greedy algorithm produces an inoptimal solution, if such a situation exists.

```
coins = [1, 4, 5]

def find_coins_brute(x):
    solutions = [[]]

    for solution in solutions:
        if sum(solution) == x:
            return len(solution)
        for coin in coins:
            solutions.append(solution + [coin])

def find_coins_greedy(x):
    count = 0
    for coin in reversed(coins):
        while coin <= x:
            x -= coin
            count += 1
    return count

x = 1
while True:
    result_brute = find_coins_brute(x)
    result_greedy = find_coins_greedy(x)

    if result_brute != result_greedy:
        print("different answer for", x, "coins")
        print("brute:", result_brute)
        print("greedy:", result_greedy)
        break

    x += 1
```

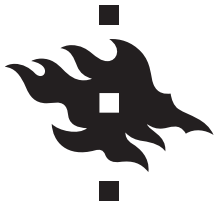
On the first line, the list `coins` defines the set of available coin values. The function `find_coins_brute` computes a solution by brute force and the function `find_coins_greedy` constructs a greedy solution. The program iterates through values $x = 1, 2, 3, \ldots$ and reports if the two functions return a

different answer.

When the coin values are $1, 4, 5$, the program finds the following situation:

```
different answer for 8 coins
brute: 2
greedy: 4
```

This means that $x = 8$ is the smallest case, where the greedy algorithm produces an incorrect answer when the coin values are $1, 4, 5$. In this case, the brute force solution is $[4, 4]$ while the greedy solution is $[1, 1, 1, 5]$.



HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI