Search space and brute force

•••

If all you have is a hammer, everything looks like a nail

Search Space

- Ensemble of all solutions we have to choose from to get the optimal answer

- Often what is asked from us is not the solution but one of its properties

Search Space on an example : minimum pair

- Find the minimum pair on the list IE the pair with the minimum sum

0 5 8 6 -1 4 9 10 2

➡ Search space : all the pairs

Make sense of your search space

Another representation :

Matrix where

line : first element of the pair

Column : second element

Implementation : two for loops

	0	5	8	6
0	------	5	8	6
5	5	------	13	11
8	8	13	------	14
6	6	11	14	------

Brute force - introduction

Example: I want to find all subsets of a set which sum is 42

I can:

- make a list of all subsets
- for each subset, calculate the the sum and check if it is 42

Search space: all subsets of the array

Test: the sum is 42

A *brute force search* has a *search space* and a *test*

Brute force - construct the search space

The python module `itertools` provides useful tools to construct the search space.
See https://docs.python.org/3/library/itertools.html

```
>>> list(product("ab", "cd"))
[('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd')]
>>> list(permutations("abc", 2))
[('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'), ('c', 'a'), ('c', 'b')]
>>> list(combinations("abc", 2))
[('a', 'b'), ('a', 'c'), ('b', 'c')]
>>> list(combinations_with_replacement("abc", 2))
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', 'b'), ('b', 'c'), ('c', 'c')]
```

Note: we use `list()` because the return values are *lazy-evaluated* iterables
→ these are very useful to save memory

Quick tour of theses functions

product(q_1,q_2,...,[repeat=1])

`product('ABCD', repeat=2)`	`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`

Can take multiple iterables and output the cartesian product between them

The repeat field indicate how many time do we take the iterables

Complexity : $O((\prod q_i)^{repeat})$

Where q_i is the size of each iterable

Quick tour of theses functions

permutation(p, [r=len(p)])

`permutations('ABCD', 2)`	`AB AC AD BA BC BD CA CB CD DA DB DC`

Take an iterable and create every permutation of length r

A permutation is an ordered (order is important) of element without repetition

Complexity : $O(\frac{p!}{(p-r)!})$

For example with r=2, the complexity is p*(p-1)

Where p is used as the size of the iterable p

Quick tour of theses functions

combinations(p, r)

`combinations('ABCD', 2)`	`AB AC AD BC BD CD`

Take an iterable and create every combination of length r

A combination is a non ordered of element without repetition, since it is non ordered, AB and BA are the same

Complexity : $O(\binom{p}{r}) = O(\frac{p!}{r!(p-r)!})$

Where p is used as the size of the iterable p

Quick tour of theses functions

combinations_with_replacement(p, r)

`combinations_with_replacement('ABCD', 2)`	`AA AB AC AD BB BC BD CC CD DD`

Take an iterable and create every combination of length r with replacement

Same as combination, but we can repeat the same element multiple times

Complexity : $O(\binom{p+r-1}{r}) = O(\frac{(p+r-1)!}{r!(p-1)!})$

Where p is used as the size of the iterable p

Brute force - let's solve our example

```
from itertools import combinations

arr = [int(n) for n in input().split()]

gen = (sub for size in range(len(arr))
       for sub in combinations(arr, size))

for sub in gen:
    if sum(sub) == 42:
        print(sub)
```

```
IN:
10 12 25 30 17 8 14 9 6

OUT:
(12, 30)
(25, 17)
(25, 8, 9)
(10, 12, 14, 6)
(10, 17, 9, 6)
```

`gen` is Python magic called a *generator expression*.
Ask me if you want to know more.

Why not to use brute force

There is a problem called *combinatorial explosion*
Cf https://en.wikipedia.org/wiki/Combinatorial_explosion

Tl; dr: the search space is often a lot bigger than the input size (e.g exponential)

Previous example: for an array of size n, there are 2^n - 1 non-empty subsets...

Why ~~not~~ to use brute force

- Brute force is a good way to **test** if more complicated algorithms are correct (you compare their results on small sets)
- Sometimes you just don’t know a better algorithm

But, can’t we reduce the search space a little bit?

→ often you can use *backtracking*

Backtracking

•••

Stop it, before it's too late !

Backtracking - introduction

We need to reduce the *search space*

Example: sudoku

- Try to fill the sudoku
- Everytime you encounter a conflict, change the last number
- If no number is ok, erase and change the previous one

etc

5	3	4	6	7	8	9	1	2
6	2	7	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	1	6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Backtracking - concepts

The idea of backtracking is to:

- build the elements of the search space incrementally
- eliminate wrong partial solutions → and therefore all solutions that contain them

You can think of the search space as a *tree*, you will often use a *recursive function*

E.g for sudoku, each level of the tree corresponds to a empty cell in the grid

→ the size of the search space is 9^n but many cases can be discarded

Implementation of a backtracking sudoku solver in Python

```
def is_valid(grid, i, j, val):
    line = grid[i]
    column = [grid[k][j] for k in range(9)]
    square = [grid[3 * (i // 3) + k][3 * (j // 3) + l]
              for k in range(3) for l in range(3)]
    return not (val in line or val in column or val in square)

def backtracking(grid, i, j):
    if i == 9: return True
    nexti, nextj = (i if j < 8 else i + 1), (j + 1) % 9
    if grid[i][j] != 0:
        return backtracking(grid, nexti, nextj)
    for val in range(1, 10):
        if is_valid(grid, i, j, val):
            grid[i][j] = val
            if backtracking(grid, nexti, nextj): return True
            grid[i][j] = 0
    return False
```

```
grid = [list(map(int, input().split()))
        for _ in range(9)]

if not backtracking(grid, 0, 0):
    print("Impossible")
else:
    print("\n".join(" ".join(
            map(str, grid[i]))
            for i in range(9)))
```

Implementation of a backtracking sudoku solver in Python

IN:

```
5 3 0 0 7 0 0 0 0
6 0 0 1 9 5 0 0 0
0 9 8 0 0 0 0 6 0
8 0 0 0 6 0 0 0 3
4 0 0 8 0 3 0 0 1
7 0 0 0 2 0 0 0 6
0 6 0 0 0 0 2 8 0
0 0 0 4 1 9 0 0 5
0 0 0 0 8 0 0 7 9
```

OUT:

```
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
```

Partial candidates explored: 6428

Total size of the workspace: $8{,}86293812 \times 10^{21}$

Credits

Slides: Louis Sugy, Arthur Tondereau

Sudoku sample: Wikipedia