ICC4101 - Algorithms and Competitive Programing

Javier Correa

Backtracking

Generic Algorithm

```
backtracking(A, k):
  if done:
    then return
  if is_solution(A, k):
    process_solution(A, k)
  for each c ∈ extend_solution(A, k):
    A[k] = c
    if test(A,k):
      backtracking(A, k + 1)
```

Generic Algorithm

- `is_solution(·)` indicates if the argument if a complete solution to the problem.
- `process_solution(·, ·)` process a/the solution to the problem.
- `extend_solution(·)` given a partial solution, return/generates all solution one step larger.
- `test(.)` this function returns true if the extended solution is a valid solution.
- `done` global variable that signals if no more backtracking is required.

Knuth's Permutation

There are some permutation generation techniques in Knuth’s book “The Art of Computer Program- ming - Volume 1”. One of the processes is as follows:

For each permutation A1A2 . . . An−1 form n others by inserting a character n in all possible places obtaining

```
nA1A2 ...An−1,A1nA2 ...An−1,...,A1A2 ...nAn−1,A1A2 ...An−1n
```

For example, from the permutation 231 inserting 4 in all possible places we get 4231 2431 2341 2314.

Following this rule you have to generate all the permutation for a given set of characters. All the given characters will be different and there number will be less than 10 and they all will be alpha numerals. This process is recursive and you will have to start recursive call with the first character and keep inserting the other characters in order. The sample input and output will make this clear. Your output should exactly mach the sample output for the sample input.

Sample Input

```
abc
```

Sample Output

```
cba
bca
bac
cab
acb
abc
```

How to model this problem

A partial solution to this problem will be a Knuth permutation as follows:

```
w = A1A2...An
```

A final solution if found when the length of the word `w` has the desired length (number of given characters)

In [1]:
```python
def is_solution(w):
    return len(w) == N

def process_solution(w):
    print(w)
```

We extend a solution by creating all possible Knuth permutation of a word `w` using a character `c`:

In [17]:
```python
def extend_solution(c, w):
    for i in range(len(w) + 1):
        yield w[:i] + c + w[i:]
```

In [15]:
```python
def extend_solution(c, w):
    yield from (w[:i] + c + w[i:] for i in range(len(w) + 1))
```

For each generted word of the Knuth permutation, the generating procedure is called recursivelly:

In [24]:
```python
def backtracking(chars, w=""):
    if is_solution(w):
        process_solution(w)
    else:
        for w_k in extend_solution(chars[0], w):
            backtracking(chars[1:], w_k)
```

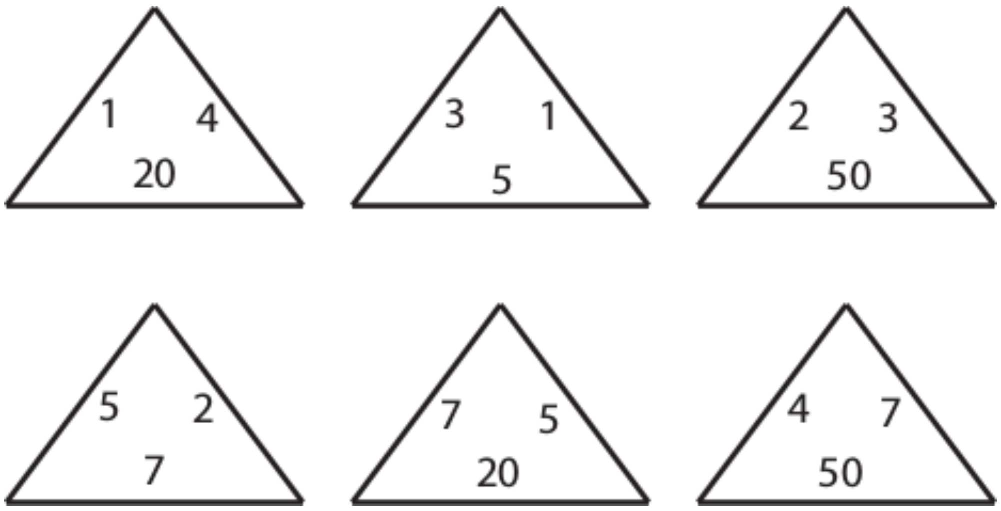
In [25]:
```python
N = 3
backtracking("abc")
```

```
cba
bca
bac
cab
acb
abc
```

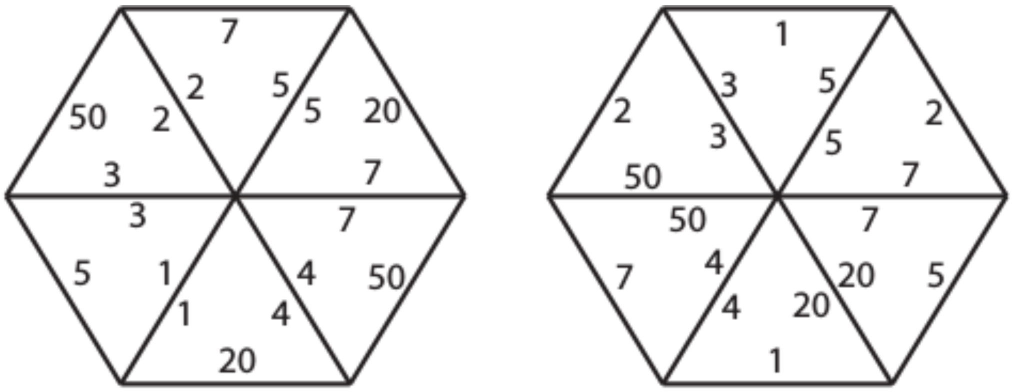
In [26]:
```python
def knuth_perm(chars, w=""):
    if len(chars) == 0:
        print(w)
        return

    for i in range(len(w) + 1):
        knuth_perm(chars[1:], w[:i] + chars[0] + w[i:])
```

Triangle Game



In the triangle game you start off with six triangles numbered on each edge, as in the example above. You can slide and rotate the triangles so they form a hexagon, but the hexagon is only legal if edges common to two triangles have the same number on them. You may not flip any triangle over. Two legal hexagons formed from the six triangles are illustrated below.



20+5+50+7+20+50=152   1+7+2+1+2+5=18

The score for a legal hexagon is the sum of the numbers on the six outside edges.

Your problem is to find the highest score that can be achieved with any six particular triangles.

Input

The input file will contain up to ten datasets. Each dataset is a sequence of six lines. On each line will be three integers in the range [1,100] separated by space characters. These three integers are the numbers on the triangles given in clockwise order. Datasets are separated by a line containing only an asterisk. The last dataset is followed by a line containing only a dollar sign.

Output

For each input data set, the output is a line containing only the word “none” if there are no legal hexagons, or the highest score if there is a legal hexagon.

How to model a solution to the problem?

An incomplete solution will be modelled as a list of rotated pieces:

```
sol = [..., (i, n1, n2, n3)]
```
where `i` is the index of the tile and `n1`, `n2` and `n3` are the tile numbers.

The solution is valid when the last number of a tile is the same number as the first number of the next tile. The solution will be completed when `len(sol) == 6`.

In [ ]:
```
def test(sol):
    for sol1, sol2 in zip(sol[:-1], sol[1:]):
        if sol1[3] != sol2[1]:
            return False
    if len(sol) == 6:
        if sol[5][3] != sol[0][1]:
            return False
    return True

def is_solution(sol):
    return len(sol) == 6
```

To extend a solution add a new tile which has not been use before with all it's possible rotations.

In [ ]:
```
def extend_solution(sol, triangles):
    triangle_index = [si[0] for si in sol]
    if len(sol) < 6:
        for x in range(6):
            if x not in triangle_index:
                yield sol + [(x, triangles[x][0], triangles[x][1], triangles
                yield sol + [(x, triangles[x][2], triangles[x][0], triangles
                yield sol + [(x, triangles[x][1], triangles[x][2], triangles
```

Processing the solution implies finding the best solution by adding the middle numbers of the tiles.

```
In [15]: def process_solution(sol):
             global best_val, best_sol
             value = sum(si[2] for si in sol)
             if best_val is None or value > best_val:
                 best_val = value
                 best_sol = sol
```

The main `backtracking` procedure follows the same structure as the one shown in the previous class.

```
In [16]: def backtracking(triangles, sol=[]):
             if is_solution(sol):
                 process_solution(sol)
             else:
                 for sol_k in extend_solution(sol, triangles):
                     if test(sol_k):
                         backtracking(triangles, sol_k)
```

Now, let's try the backtracking.

```
In [19]: best_val = None
         best_sol = None
         triangles = [[4, 7, 50],
                      [3, 1, 5],
                      [5, 2, 7],
                      [1, 4, 20],
                      [7, 5, 20],
                      [50, 2, 3]]

         backtracking(triangles)
         best_sol
```

```
Out[19]: [(0, 7, 50, 4),
          (3, 4, 20, 1),
          (1, 1, 5, 3),
          (5, 3, 50, 2),
          (2, 2, 7, 5),
          (4, 5, 20, 7)]
```

```
In [20]: best_val
```

```
Out[20]: 152
```

1. Getting in line

`https://www.udebug.com/UVa/216`

2. ASCII Labyrinth

`https://www.udebug.com/UVa/10582`

3. The House Of Santa Claus

`https://www.udebug.com/UVa/291`

4. DNA

`https://www.udebug.com/UVa/11961`

5. The Settlers of Catan

`https://www.udebug.com/UVa/539`

6. A Gentlemen’s Agreement

`https://www.udebug.com/UVa/11065`

In [ ]: