Enumeration and Backtracking

1. Enumeration
   - enumeration of all bit combinations
   - enumeration of letter combinations
   - adding a stopping condition

2. Backtracking
   - the knapsack problem
   - a recursive solution

3. Summary and Exercises
Enumeration and Backtracking

1. Enumeration
   - enumeration of all bit combinations
   - enumeration of letter combinations
   - adding a stopping condition

2. Backtracking
   - the knapsack problem
   - a recursive solution

3. Summary and Exercises
Motivation: automatic generation of truth tables.

Simplifying expressions via De Morgan’s law:

\[ \text{not (not x or not y)} = x \text{ and } y \]

A Truth Table proves De Morgan’s law:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

0 = False, 1 = True, (1) = not x, (2) = not y, (3) = (1) or (2), (4) = not (3), (5) = x and y
Writing all Bit Combinations
output of running `enumbits.py`

```
$ python enumbits.py
Give number of bits : 3
[0, 0, 0]
[0, 0, 1]
[0, 1, 0]
[0, 1, 1]
[1, 0, 0]
[1, 0, 1]
[1, 1, 0]
[1, 1, 1]
```

Observe the pattern: lexicographic enumeration.
Leaves of a Binary Tree

tree generated recursively

The bit combinations are leaves of a binary tree:

000 001 010 011 100 101 110 111

The tree grows from its root to the leaves.

- every leave is a print statement (base case)
- every internal node is a recursive function call
A Recursive Algorithm
to enumerate all bit combinations

Let’s define a recursive function `enumbits`.

Two parameters for the function `enumbits`:

- `bits` a list as accumulating parameter
- `k` current position in the list

Initially, `k == 0` and `bits` holds `n` objects.

Base case when `k == len(bits)`: `print bits`.

For `k < len(bits)`:

1. `bits[k] = 0`, call `enumbits` for `k+1`
2. `bits[k] = 1`, call `enumbits` for `k+1`
The Main Program

def main():
    
    """
    Prompts the user for number of bits and initializes the list.
    """

    nbr = int(input('Give the number of bits : '))
    lst = [0 for _ in range(nbr)]
enumbits(0, lst)

calls

def enumbits(k, bits):
    
    """
    Writes all bit combinations of len(bits), starting at the k-th position.
    """
```python
def enumbits(k, bits):
    """
    Writes all bit combinations of len(bits), starting at the k-th position.
    """
    if k >= len(bits):
        print(bits)
    else:
        bits[k] = 0
        enumbits(k+1, bits)
        bits[k] = 1
        enumbits(k+1, bits)
```
for truth tables

Exploit Python’s dynamic typing:

```python
>>> e = "not x or not y"
>>> x = True
>>> y = False
>>> eval(e)
True
```

Logical expressions as strings delays their evaluation. Apply `eval` only if variables have values.

Enumerating more than two alternatives:

- number $N$ of alternatives is extra parameter
- `for k in range(N)` in general case
Enumeration and Backtracking

1. Enumeration
   - enumeration of all bit combinations
   - enumeration of letter combinations
   - adding a stopping condition

2. Backtracking
   - the knapsack problem
   - a recursive solution

3. Summary and Exercises
combining letters into words

Enumerate all three letter words:
1. starting with c, s, or v
2. with one vowel in the middle
3. ending in d, t, or w

Combine three lists:

```python
csv = ['c', 's', 'v']
vowels = ['a', 'e', 'i', 'o', 'u']
dtw = ['d', 't', 'w']
```

Number of words: \(3 \times 5 \times 3 = 45\).
a recursive solution

Our recursive function `enumwords` has 3 parameters:

- `letters` a list of lists to choose from
- `k` index to current list
- `accu` string accumulating the word

Adding a character `c` to a string `accu`: `accu + c`.

Initially, `k == 0` and `accu == ""`.

The recursive algorithm compares `k` to `len(letters)`:

1. if `k == len(letter)`, then print `accu` (base case)
2. for all characters `letter in letters[k]` do
   - call `enumwords with k+1 and accu + letter`
def main():
    """
    enumerates letter combinations
    """

csv = ['c', 's', 'v']
vowels = ['a', 'e', 'i', 'o', 'u']
dtw = ['d', 't', 'w']
let = [csv, vowels, dtw]
enumwords(0, let, '')

calls
def enumwords(k, letters, accu):
    """
    Starting with the k-th list in letters, adds letters to the current string accu.
    """
the function enumwords

def enumwords(k, letters, accu):
    """
    Starting with the k-th list in letters, adds letters to the current string accu.
    """
    if k >= len(letters):
        print(accu)
    else:
        for letter in letters[k]:
            enumwords(k+1, letters, accu+letter)
Enumeration and Backtracking

1. Enumeration
   - enumeration of all bit combinations
   - enumeration of letter combinations
   - adding a stopping condition

2. Backtracking
   - the knapsack problem
   - a recursive solution

3. Summary and Exercises
adding a stop condition

Asking the user whether to continue:

```python
$ python enumwstop.py
the word is "cad" continue ? (y/n) y
the word is "cat" continue ? (y/n) y
the word is "caw" continue ? (y/n) y
the word is "ced" continue ? (y/n) y
the word is "cet" continue ? (y/n) n
```

Changes to `enumwords`:

- the base case calls an interactive function
- `enumwords` returns a boolean: to continue or not
- break out of loop in general case if recursive call returns `False`
def ask_to_continue(word):
    
    Shows the string word and then asks the user to continue or not.
    
    qst = 'the word is "' + word + '" continue ? (y/n) ' 
    ans = input(qst) 
    return ans == 'y'
def enumwords(k, letters, accu):
    """
    Starting with the k-th list in letters, adds letters to the current string accu, returns a boolean: to continue or not.
    """
    if k >= len(letters):
        return ask_to_continue(accu)
    else:
        for letter in letters[k]:
            cont = enumwords(k+1, letters, accu+letter)
            if not cont:
                break
        return cont
Enumeration and Backtracking

1. Enumeration
   - enumeration of all bit combinations
   - enumeration of letter combinations
   - adding a stopping condition

2. Backtracking
   - the knapsack problem
   - a recursive solution

3. Summary and Exercises
The Knapsack Problem

A traveler selects objects to put in a knapsack.

Input/Output specification:

**Input** : a list $V$ of values, lower bound $lb$ and upper bound $ub$.

**Output** : all selections $s$ with corresponding values in $L$: $lb \leq \sum(L) \leq ub$.

Extended version has on input a list of weights.

- The upper bound is for the sum of weights.
- The lower bound is for the sum of values.

Restricted problem: subset sum problem.

- Find all sums of numbers in a given list that match a given value.
running **knapsack.py** script

$ python knapsack.py
give number of objects : 5
give a list of 5 values : [2.1, 1.8, 3.2, 4.1, 0.8]
lower bound on sum : 3
upper bound on sum : 5

\[ V([0, 1, 4]) = 2.10 + 1.80 + 0.80 = 4.70 \]
\[ V([0, 1]) = 2.10 + 1.80 = 3.90 \]
\[ V([1, 2]) = 1.80 + 3.20 = 5.00 \]
\[ V([2, 4]) = 3.20 + 0.80 = 4.00 \]
\[ V([2]) = 3.20 = 3.20 \]
\[ V([3, 4]) = 4.10 + 0.80 = 4.90 \]
\[ V([3]) = 4.10 = 4.10 \]
Enumeration and Backtracking

1. Enumeration
   - enumeration of all bit combinations
   - enumeration of letter combinations
   - adding a stopping condition

2. Backtracking
   - the knapsack problem
   - a recursive solution

3. Summary and Exercises
a recursive solution

The recursive function \texttt{knap sack} takes on input the list of values \( V \) and the bounds on the sum.

The parameter \( k \) is the index of the current object being considered for inclusion in the knapsack.

Base case: \( k == \text{len}(V) \), if within bounds, print.

For \( k < \text{len}(V) \), consider adding object \( k \)

- only if its value \( V[k] \) plus the current sum does not exceed the upper bound
- also call for \( k+1 \) \textit{without} adding

Two accumulating parameters: current selection and list of values of current selection.
specification of a recursive function

```python
def knapsack(things, low, upp, k, sel, val):
    """
    Shows all selections sel of things whose sum
    of values in val: low <= sum(val) <= upp.
    Input parameters:
things : a list of values;
low : lower bound on sum;
upp : upper bound on sum;
k : index to the current thing.
Accumulating parameters:
sel : stores selection of things;
    if selected i-th thing in things,
    then sel[i] == 1, else sel[i] == 0;
val : values of the selected things.
Initialize sel and val both to [], k to 0.
```
def knapsack(things, low, upp, k, sel, val):
    if k >= len(things):
        if low <= sum(val) <= upp:
            write(sel, val)
    if k < len(things):
        if sum(val) + things[k] <= upp:
            sel.append(k)
            val.append(things[k])
            knapsack(things, low, upp, k+1, sel, val)
            del sel[len(sel)-1]
            del val[len(val)-1]
    knapsack(things, low, upp, k+1, sel, val)
def write(sel, val):
    
    Does formatted printing in knapsack, for the selection sel and values val.
    
    result = 'V(' + str(sel) + ') = ' 
    for i in range(0, len(val)):
        if i > 0:
            result = result + ' + ' 
        result = result + '%.2f' % val[i]
    result = result + ' = ' + '%.2f' % sum(val)
    print(result)
Code for `main()`

```python
def main():
    ""
    Prompts user for the number of objects
    asks for the list of their values,
    a lower and an upper bound on the sum.
    ""
    
    from ast import literal_eval
    nbr = int(input('give number of objects : '))
    qst = 'give a list of %d values : ' % nbr
    vals = literal_eval(input(qst))
    low = float(input('lower bound on sum : '))
    upp = float(input('upper bound on sum : '))
    knapsack(vals, low, upp, 0, [], [])

    if __name__ == '__main__':
        main()
```
Enumeration and backtracking are *exhaustive* problem solving methods, applying recursive algorithms.

The exhaustiveness often leads to exponential time. Greedy algorithms and heuristic methods may prune the search space and give satisfactory practical performance.
Exercises

1. Use the enumeration of all bit combinations in a script to verify using a truth table whether two logical expressions are equivalent.

2. Write code to enumerate all sublists of a given list.

3. Modify the code for the recursive function \texttt{knapsack} with an interactive stopping condition.

4. Let the script for the knapsack problem also take on input the weights of the objects. Prompt the user for an additional list of weights and use the upper bound for the total weight of the selected objects.

5. In chess, a queen can attack all squares that run horizontally, vertically, and diagonally starting at the queen’s position. On an 8-by-8 chess board, compute in how many ways you can put 8 queens so they do not attack each other.