Algorithms and Data Structures

Seminars Guide, 2023–24

José Luis Balcázar

Departament de Ciències de la Computació, UPC

Session 1, 11/01: Review of various notions, including recursion

Today:

- Short briefing
- Higher order
- Graphs and trees
- Recursion
- Evaluable exercise (slight corrections made the day after)
- Exercises for the week
- Additional related material

Short, partial briefing including Jutge courses.

- Most info, including evaluation, deferred to the theory class on Monday.
- Mandatory lab exercises to do *in class* most Thursdays and to be submitted through the Aula Virtual are part of the evaluation, including *today*'s.
- Attendance to the Lab Sessions is *mandatory.* Only Aula submissions made from the classroom will be considered. *I want to see you submitting from our room.*
- Jutge courses:
  - Specific, *mandatory*, non-public (invitations sent round):
    - Algorithms and Data Structures Winter 2023-24 ESCI.
    - This course will be expanded along the quarter with the addition of some new lists or new problems;
    - further, exams will refer to it.
  - Optional, open, *recommended*: Ad computationem vIA reGIA.
  - Optional on demand: ask me for invitations to the first-year courses of 2023-24 if you believe they can be helpful to you.

Higher order

1. Functions as arguments *or results* of other functions. Examples.
2. Higher order for today:
    - Reading items with `pytokr` and
    - looping on them all.
    - The problems in the first three lists of the Jutge course will give you practice with `pytokr` and with general higher order programming.
    - They are also part of the current version of Programming and Algorithms 1.

Graphs and trees

Review and terminology:

- graphs, directed and undirected,
- free trees,
- rooted general trees,
- binary trees.

How are we representing binary trees in our programs?

And in our input data?

Recursion

The fourth list in the Jutge course contains problems to be solved *recursively*. You are likely to have iterative solutions of many of them. They are no longer what is required now! Work *only* recursively, *no loops*.

Goal for our first couple of weeks: proficiency with recursive programs with *more than one* recursive call per call.

Examples:

- reading binary trees,
- reading and traversals of binary trees (P98436).

Specific programming exercises:

1. Write first a program that reads binary trees from `stdin` as per P98436, but assuming just one integer per line, so that the standard function `input()` will keep providing you with the next element to be read.

2. Test it out, e. g. by asking for specific components of the tree just read within the Python interpreter.

3. Move on into a program that reads the tree without assuming anymore that the values come in different lines, using `pytokr()`, and test it out in the same way.

4. Then, complete the program with traversals and try to get green light (or at least yellow due to presentation error) in the Jutge problem P98436.

Evaluable exercise

Upload to the Aula a screen capture of your best attempt to solving Jutge problem P98436 before leaving the classroom.

Exercises for the week

All can be found in our course on `jutge.org`, within the list Recursion Revisited.

1. Solve P61120.
2. Generating (of course, *recursively*, no loops) all subsets of a given set of words: solve P18957.
3. Evaluating prefix expressions: solve P20006.
4. Size-based encoding of binary trees: solve X30150 (midterm exam problem last year).
5. Reading and traversal of general trees: solve P66413. Feel free to choose to ignore the `C++` code suggestions and work instead fully in Python, if you prefer that option.
6. Once you have working Python solutions, consider trying `C++` (P57669).

We will get back to all this later on with alternative programs for tree traversals and with the *graph variant* of tree preorder traversal, namely, *depth-first search* (DFS).

Additional related material

This code got green light on P98436:

```
from pytokr import pytokr

def postorder(tree):
    if tree:
        r = postorder(tree[1])
        r.extend(postorder(tree[2]))
        r.append(tree[0])
        return r
    else:
        return list()
```

```
def inorder(tree):
    if tree:
        r = inorder(tree[1])
        r.append(tree[0])
        r.extend(inorder(tree[2]))
        return r
    else:
        return list()

def read_tree_in(readitem):
    item = int(readitem())
    if item == -1:
        return tuple()
    else:
        return (item, read_tree_in(readitem), read_tree_in(readitem))

# Careful: in P98436, different from P57669, there is no size to ignore

readitem = pytokr()

t = read_tree_in(readitem)

# ~ print("pos:", " ".join(map(str, postorder(t)))) # PE in P98436
# ~ print("ino:", " ".join(map(str, inorder(t))))

print("pos:", end = '')
for item in postorder(t):
    print(' ' + str(item), end = '')
print()
print("ino:", end = '')
for item in inorder(t):
    print(' ' + str(item), end = '')
print()
```

Session 2, 18/01: Backtracking, I: Knapsack variants

Today:

- Individual review: Decisional Knapsack by backtracking, first solution
- Decisional Knapsack by backtracking, all solutions
- Subset Sum
- Evaluable exercise
- Exercises for the week
- Additional related material

Individual Review: Decisional Knapsack by backtracking, first solution

Review the notion and first examples of **set-based backtracking** from the theory session, namely, single-solution Knapsack by exhaustive search and by set-based backtracking. The codes discussed in the theory session are available through the Aula Virtual.

You will find also there a text file with examples to cut and paste from. How to read in the data for one of these cases? Observe that, first, we find the desired value and the weight bound and, after them, the number of objects is given. Then, we need to read a known number of integers (two for each object). For all these readings, the first function returned by `pytokr` suffices: no iterator is necessary.

Decisional Knapsack by backtracking, all solutions

1. *Optionally*: Start by modifying the exhaustive search algorithm based on the tree of subsets (file name `...simple_first_no_back.py`) so as to return the list of all the valid solutions for a given upper bound on weight and a given lower bound on value. Try it on the examples in the text file mentioned above. *Alternatively*, you can choose to skip this part and move on directly to the next point.

2. Modify the single-solution backtracking (file name `...simple_first.py`) so as to return the list of all the valid solutions for a given upper bound on weight and a given lower bound on value. Try it on the examples in the text file mentioned above.

3. Once your backtracking program works correctly on all these examples, adapt it a bit so as to try it out in the Jutge: X94664. Your Jutge submission can print out the solutions in any order; also, within each solution, the object numbers may be printed out in any order. Follow the format of the public tests.

Subset Sum

Consider the following problem, similar to Knapsack but slightly simpler. Input consists of a given positive integer as a *goal* (you may think of it as change to be given for a payment) and a sequence of positive integers, the *addends*, possibly repeated (you may think of it as the values of the currency items available to us like coins and bills). Is there a way of selecting a set of addends from the sequence that adds up exactly to the goal?

This problem is widely known as the *Subset Sum* problem.

Your program must read these data and answer the question by applying the **set-based backtracking** schema.

1. First, write a program that returns one solution, provided that one exists.
2. Then, change your program so that it returns all solutions.

Evaluable exercise

Solve P40685 in list Combinatorial Search Schemes I; upload to the Aula a screen capture of your best attempt before leaving the classroom.

This exercise is to be solved individually.

You will recognize in P40685 yet another variant of Subset Sum.

However, **be careful:** the condition that all input integers are positive is not guaranteed anymore. Hence, your programs for Subset Sum that assume that goal and addends are positive might not work. Given this change, how do you propose to organize your algorithm so as to detect subtrees where we can safely cut out the exploration?

Exercises for the week

1. Solve some pending exercises you may have still left from the list Recursion Revisited.
2. Like P40685, except that you only look for one solution, and stop the search as soon as found. (Of course, it's no use submitting this solution to the Jutge.)
3. Read the statement of Interval Subset Sum, X56351 in list Combinatorial Search Schemes I. The text refers to an optimization criterion ("shortest solution") that will be covered in the next sessions. For the time being, solve instead the following two simplified forms: finding just one arbitrary solution and finding all the solutions. (Of course, again it would be no use submitting these intermediate programs to the Jutge yet.)

Additional related material

This program gets green light on P40685 Equal Sums (1). It displays a couple of variations over the materials seen before. (A) It prints out solutions as they are found instead of returning them all via the recursive calls as a list. This variation is common in the (infrequent) cases of requiring to display all the solutions as it spares the memory of the list of solutions returned. (B) As we pointed out, there may be negative numbers; if not taken into account, a current sum above the limit might get later reduced, so that we cannot prune the exploration on these grounds. One way to work around this problem is to make sure that negative addends are handled first. Then, when we see a positive addend, we know that all coming ones after it are also positive and cannot make up for a quantity already too high. Of course, this allows for a symmetric argumentation by pruning on the converse inequality: work it out on your own.

```
from pytokr import pytokr

item = pytokr()

def solutions(addends, goal, current_item, curr_sol, sum_sol):
    if current_item == -1:
        if sum_sol == goal:
            report(curr_sol)
    else:
        solutions(addends, goal, current_item - 1, curr_sol, sum_sol)
        if (addends[current_item] <= 0 or
            sum_sol + addends[current_item] <= goal):
                solutions(addends, goal, current_item - 1,
                    curr_sol + [addends[current_item]],
                    sum_sol + addends[current_item])

def report(solution):
    print('{' + ','.join(map(str, solution)) + '}')

goal = int(item())
size = int(item())
addends = []
for _ in range(size):
    addends.append(int(item()))

addends.sort(reverse = True) # make sure negative addends are explored first

solutions(addends, goal, len(addends) - 1, [], 0)
```

Session 3, 25/01: Backtracking, II: Optimization and more

Today:

- Individual review: variants of backtracking
- Knapsack once more
- Evaluable exercise
- Exercises for the week
- Additional related material

Individual review: variants of backtracking

Review all the variants and examples of **backtracking** from the theory session. The codes discussed in the theory session are available through the Aula Virtual. **But**: don't look yet at the codes for the optimization version of Knapsack; the one in the slides will be replaced by a slightly clearer one, and the new one comes conveniently password-protected to facilitate your behaving.

Knapsack once more

Without looking at the codes in the slides or in the zip file, as said, try to solve on yourself the optimization version of Knapsack up to green light on Jutge X59240 in list Combinatorial Search Schemes (I).

Hint:

```
def _knapsack(weights, values, current_item, max_w):`
    if current_item == -1:
        ...
    else:
        ...
    return ... # (returns three things)

def knapsack(weights, values, n, max_w):
    sol, wei, val = _knapsack(weights, values, n-1, max_w)
    return sol
```

As the problem name suggests, the official solution of X59240 is *extremely* slow so that, even if your backtracking is inefficient, you still have good options to get green light. Pay attention to the somewhat nonstandard form of the doctests.

Evaluable exercise

The evaluable exercise will be announced after 45 minutes.

Exercises for the week

1. Complete, if necessary, what you did for the evaluable exercise.
2. As you have seen, the zip file for this session contains fresh code for optimization knapsack, as well as codes for nonbinary backtracking on edge-colorability and N-queens. Run these codes and explore them: try various values, change the number of colors. . .
3. Solve X71353 Rod Cutting in list Combinatorial Search Schemes (II) up to green light.

Additional related material

Here is a simple backtracking that gets green light on Interval Subset Sum, X56351.

```
def _sub_sum_interval(addends, current_item, lower, upper, cand, sumcand):
    "sumcand < upper guaranteed"
    if current_item == -1:
        if lower <= sum(cand):
            return cand
        else:
            return list()
    else:
        without_it = _sub_sum_interval(addends, current_item - 1,
                                       lower, upper, cand, sumcand)
        if sumcand + addends[current_item] < upper:
            with_it = _sub_sum_interval(addends, current_item - 1,
                                        lower, upper,
                                        cand + [addends[current_item]],
                                        sumcand + addends[current_item])
            if not without_it or 0 < len(with_it) < len(without_it):
                return with_it
        return without_it

def sub_sum_interval(lower, upper, addends):
    "upper is a positive integer so 0 < upper"
    return _sub_sum_interval(addends, len(addends) - 1,
                             lower, upper, list(), 0)
```

Session 4, 01/02: Dynamic Programming, I

Today:

- Individual review: basic ideas of Dynamic Programming
- Complete a Minimum Change variant
- Evaluable exercise
- Knapsack again and again
- Shortest paths
- Exercises for the week
- Infrastructure warning
- Additional related material

Individual review: basic ideas of Dynamic Programming

Review all the indications about **dynamic programming** from the theory session; **but**: work individually for this task. The codes discussed in the theory session are available through the Aula Virtual. Update as well your theory slide deck, where a couple of mistakes have been corrected.

On the other hand, evaluate how up-to-date are you on *backtracking* too: it is now just 11 days to the midterm and you may need to plan carefully your work up to then.

Complete a Minimum Change variant

Check out the files for Minimum Change in the Aula Virtual: the code that keeps a single list, instead of a 2-dimensional table, does not allow you to trace back the solution. The code that allows that has, instead, still a 2-dimensional table. Take inspiration in both and construct your own version of a Dynamic Programming program for Minimum Change that keeps a single list instead of a 2-dimensional table, yet is able to trace back the solution. Ideally, you should be able to get a green light on P81009 (which, mind, does not actually need to see the solutions). **But**: work individually for this task.

Evaluable exercise

Submit through the aula your best approximation to the variant that traces back the solution without spreading out the whole 2-dimensional table.

Knapsack again and again

Complete Dynamic Programming solutions for the optimization variant of Knapsack. Start by keeping a 2-dimensional table, then reduce the storage to a single dimension. Once there, try hard to get a green light on X59240 for your solution. Ask for hints from your lecturer if necessary.

Shortest paths

Explain to yourself or to your friends the recurrence

```
dist(i, j, k) = min( dist(i, j, k-1), dist(i, k, k-1) + dist(k, j, k-1) )
```

and employ it to construct a dynamic programming algorithm to find all the shortest paths between all pairs of vertices in a directed graph (Floyd's version of Floyd-Warshall-Roy). How about undirected graphs?

Exercises for the week

1. Make sure you are up to speed on multiple recursive calls and tree traversals (first seminar session).
2. Make sure you are up to speed on the level we have covered of backtracking (further sophistications will be mentioned after the midterm).
3. If you are not there yet, complete what you did for the evaluable exercise on Minimum Change by Dynamic Programming, creating a program that is able both to provide the solution, when requested, and to get green light on P81009.
4. If you are not there yet, complete what you did in order to solve Knapsack by Dynamic Programming, up to green light again on X59240.
5. Make sure that you have already a backtracking solution to X71353 Rod Cutting in list Combinatorial Search Schemes (II) up to green light; then, construct an alternative solution by Dynamic Programming, also up to green light. Then, explore alternatives (can you invent a faster backtracking?, can you reduce the dimensionality of the dynamic programming table?. . .)

Infrastructure warning

You have probably seen the indication that we might not have the Jutge available for next week's seminar, February 8th. Experience shows that, as we will be at the end of the maintenance time, chances are that it will be again available at the time of our labs. However, please plan for the shutdown by making a few screen captures of some problems, including the public test cases. The work in the labs will be designed with the lack of Jutge in mind.

Additional related material

To be added here one week after the session.