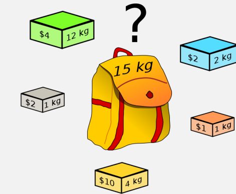
You are creating a playlist for a party. There are *n* songs that you are considering including; song *i* has a length l_i and a popularity p_i. Your goal is to create the most popular playlist possible (the playlist's popularity is the sum of the popularities of the songs included), but the playlist can't last any longer than the party does (a total time of *L*) so you have to decide which songs to include and which to leave out. However, you *can* include parts of songs; perhaps somewhat unrealistically, the popularity scales so that, for example, half of a song is half as popular as the whole song. Which songs do you include, and how much of each?

- this can be solved with a greedy algorithm
  - take the songs in order of …
- what if only whole songs can be included?
  - ... no longer works
  - (in fact, no polynomial time algorithm works)
    - key snag is that there may be empty space because no other song fits, which contributes nothing to the total popularity – but a different choice of songs, which might be individually less good, could result in less empty space and a higher total popularity

Knapsack Problem

- the popular playlist problem is an instance of the *knapsack problem*
- given a set of items, each with a weight w_i and a value v_i, find a subset of the items such that the total value is maximized but the total weight does not exceed W
  - *0-1 knapsack* only allows the item or not
    - greedy does not work
  - *fractional knapsack* allows parts of items to be chosen
    - has an efficient greedy algorithm

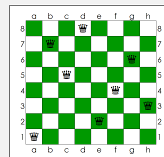


https://commons.wikimedia.org/w/index.php?curid=985491

Greedy Isn't Always Good

Greedy algorithms are typically efficient (polynomial time) but there are many situations where they don't work –

- it may not be possible to obtain a globally optimal solution via only locally optimal choices
  - e.g. 0-1 knapsack
- it may be hard to come up with a plausible greedy choice
  - e.g. n queens – place *n* queens on an *nxn* chess board so that no row, column, or diagonal contains more than one queen
- you may want to enumerate all possibilities
  - e.g. generating anagrams



https://commons.wikimedia.org/wiki/File:Solution_C_for_8_Queen_Puzzles.png

Recursive Backtracking

- applicable when the solution can be formulated as a series of choices building up partial solutions
  - e.g. solution is a particular ordering of objects (permutation)
    - choice: which to choose next?
  - e.g. solution is a particular subset of the input objects
    - choice: take or not take (or which to take next)
- needed when greedy doesn't work
  - greedy corresponds to finding a route when you can pick the right way to go every time

- **2+-friend solutions**, where there is more than one subproblem at each level
  - **divide-and-conquer**
    The problem is split into b subproblems of size n/b where $b \geq 2$ (typically 2).
  - **case analysis**
    Each friend considers a different choice.

Recursive Backtracking Formulation

We choose one alternative and then ask the friend to solve the rest of the problem in light of that choice.

- if the friend fails, we choose a different alternative and try again

- the friend's solution is constrained by the partial solution
  - partial solution is passed explicitly or implicitly (by the construction of the subproblem)

The other way around – we ask the friend to solve a smaller version of the problem and then we choose an alternative and add that to the solution – doesn't work.

- if we have several legal alternatives, which do we pick?
  - the discovery that our choice was bad comes after we've returned our answer

- the friend's choices are only constrained by the partial solution, so we have no way to direct them to come up with a different solution

Key Points – Recursive Backtracking

- (essentially) exhaustive search applies when greedy fails – the right choice at the moment depends on what else can happen later
- development via the 16-step process
  - very similar to the general recursive process
  - adds "partial solution" and "alternatives" as part of generalize / define subproblems
  - emphasis is on establishing the problem, assembling the algorithm, showing the main case, and time
    - most of the proving correctness steps are either trivial or always the same given the framework that the 16-step process provides
- the consequences of the series of choices on the branching factor and the longest path length
  - and the impact of these on running time
- the three structural variations (one solution, all solutions, best solution) and the corresponding code structure

16 Steps to Recursive Backtracking Success

establishing the problem

1. specifications
2. examples
3. size

brainstorming ideas

4. targets
5. tactics
6. approaches
   - process input
   - produce output

defining the algorithm

7. generalize / define subproblems
   - *partial solution*
   - *alternatives*
   - *subproblem*
8. base case(s)
9. main case
10. top level
    - initial subproblem
    - setup
    - wrapup
11. special cases
12. algorithm

showing correctness

13. termination
    - making progress
    - reaching the end
14. correctness
    - establish the base case(s)
    - show the main case
    - final answer

determining efficiency

15. implementation
16. time and space

16 Steps to Recursive Backtracking Success

- generalize / define subproblems
  - partial solution – what constitutes a solution-so-far
    - same kind of thing as a legal solution for the whole problem, but less complete (fewer choices have been made)
  - alternatives
    - the choice and the possible values for that choice
    - only consider legal alternatives
  - subproblems – the rest of the problem
    - same kind of problem (task, input, output) as the original problem but generalized – "solve the problem given this partial solution" instead of "solve the problem from scratch"
    - input may include the solution so far → output is complete solution
    - input may only be the subproblem → output is solution for subproblem

16 Steps to Recursive Backtracking Success

- making progress
  - progress is made because another choice is made
- reaching the end
  - if you keep making choices, eventually you will have made them all
- establish the base case(s)
  - the base case condition captures a complete solution
  - only legal alternatives been considered, so any complete solution reached is legal
- show the main case
  - explain why all possible alternatives for the next choice are covered
  - explain why the right partial solution is passed to each friend
  - explain why pruning is safe i.e. desired solution is not pruned away