Chapter 6

Recursive Backtracking

When only a single alternative for each choice needs to be considered, as is the case in greedy algorithms, the process of building up a solution through a series of choices naturally fits into the iterative paradigm.

Recursive backtracking is a technique for implementing exhaustive search, where (potentially) all possible legal solutions are enumerated. It belongs to the case analysis pattern within the recursive paradigm — each friend considers a different alternative for the choice.

6.1 16 Steps to a Recursive Backtracking Algorithms

Recursive backtracking algorithms are recursive, so the 16-step recursive algorithm development process (section 3.1) can be used.

6.2 Patterns

As with greedy algorithms, generating the solution is framed as making a series of (the same kind of) choices. The same common approach patterns apply:

- **process input**, where the choice is about what to do with each input element in turn
- **produce output**, where the choice is about what the next output element is

The same common flavors outlined in section 5.3 appear in recursive backtracking applications as well, and while the series of choices are no longer being made inside a loop, the nature of the choice — e.g. for a process input approach to a find a subset problem, the choice is whether or not to include the current element — remains the same.

These patterns affect the nature of the choice made, the possible values for each choice, and the number of subproblems at each step. These factors impact the running time, so if both process input and produce output seem to be feasible approaches, consider the running time to decide between them.

There are also three structural variations in how a recursive backtracking algorithm is implemented, based on the desired outcome:

- **find a legal solution**, where the goal is to find (any) legal solution
- **find the best solution**, where the goal is to find the best solution according to some optimization criteria
- **find all solutions**, where the goal is to enumerate all possible (legal) solutions

6.3 Developing Recursive Backtracking Algorithms

We now consider aspects of the recursive algorithm development process as it applies to recursive backtracking in more detail.

Size. Since the algorithm is formulated as a series of choices, the size can be expressed in terms of the number of choices left to make and the smallest problem is the one that is completely solved (all choices have been made, so none are left to make). However, it is not always known how many choices need to be made and the size may need to be defined more directly in terms of other parameters (such as input size).

Tactics. While exponential time is rarely good in any sense, distinctions between exponential growth rates can still be made — $b^n \neq \Theta(c^n)$ for $b \neq c$, for example. Keeping that in mind can influence the choice of approach (process input vs product output).

Approaches. Identify the particular flavor, if applicable, as well as the two patterns (process input and produce output) and what they look like for this problem.

Generalize / define subproblems. Friends get smaller versions of the original problem, which often takes the form of a generalized version of the original problem. For case analysis, where the original task was "solve the problem from scratch", the generalized version is "solve the rest of the problem given a solution-in-progress".

(a) *Partial solution.* Identify what constitutes a solution-so-far. This will be the same kind of thing as a legal solution for the whole problem, but less complete (fewer choices have been made).

(b) *Alternatives.* Identify the choice being made and what the (legal) alternatives are for that choice. (Alternatives that result in an illegal solution aren't considered farther.)

(c) *Subproblem.* The "rest of the problem". Define the generalized problem, its input, and its output. The input can include the partial solution so far, in which case the output would be a complete solution, or the input can only denote the subproblem, in which case the output is only the solution for the subproblem.

Base case(s). A base case occurs when there are no more choices to make. Because we only build legal partial solutions, this means a legal solution has been found.

(What exactly is done in this step depends on the structural variation — find a legal solution, find the best solution, or find all solutions — as well as specific implementation details, which are driven by efficiency concerns.)

Main case. This takes the form of: for each legal alternative for the current decision, a friend solves the remainder of the problem given the solution so far plus that choice. We return one of those solutions, the best of those solutions, or all of the those solutions depending on the structural variation.

Making progress. Progress is made because another choice is made. If the size of the problem isn't in terms of the number of choices, instead address how making one more choice reduces the size.

Reaching the end. The base case is a complete solution, so as long as progress is made, it will be reached. It's not possible to skip over a complete solution.

Establish base case(s). The base case condition corresponds to the condition for a legal solution, and the action taken (terminating, updating the best-so-far, saving) is the right action for the goal (find a legal solution, find the best solution, find all legal solutions).

Show the main case. Explain why all possible alternatives for the next choice are covered and that the right partial solution is passed to each friend. Then, assuming that the friends return the correct results for their subproblem, explain why the correct answer is produced from those results.

Implementation. The running time of a recursive backtracking algorithm is dominated by the exponential-time enumeration of all possible choices, however, there's no point in adding another multiple of n (or worse). Some algorithmic strategies will be addressed in chapter 7, but on the purely implementation side, choose an approach and a representation for partial solutions and subproblems that avoids copying collections of things as much as possible.

Time and space. Recursive backtracking is a depth-first search of a tree where the initial subproblem is the root, the base case instances are the leaves, and each partial solution / subproblem corresponds to an internal node. DFS is $O(n + m)$ where n is the number of vertices and m is the number of edges. (Note that this assumes $O(1)$ time per vertex; the time increases if that is not the case.) n and m depend on two things: the *branching factor* b, which is the number of next possibilities for each choice, and the *longest path* h, which is the largest number of decisions needed to reach a base case. In the worst case, $n = O(b^h)$ and $m = O(b^{h+1})$. While still exponential, this can be reduced by addressing the branching factor (prefer more of the input "choose or not choose" over more of the output "which next") or the longest path length (prefer more of the output — the number of decisions is the size of the solution — over more of the input — the number of decisions is the number of input elements).

6.4 Example

Specifications. State complete specifications for the problem. What is the problem? What do you start with (input) and what is the end result (output)? What are the legal input instances and the required output for each? For optimization problems, distinguish between legal solutions and optimal ones.

> Given n items, each with a value v_i and weight w_i, find the maximum value set of items that fit in a knapsack wtih capacity W. Only whole items can be taken.
>
> Input: n items, each with a value $v_i > 0$ and weight $w_i > 0$; knapsack capacity $W > 0$
>
> Output: a subset of the items
>
> Legal solution: a subset of items with total weight $\leq W$
>
> Optimization goal: maximize total value of items taken

Size. What is the size of the problem, and what constitutes a smaller problem? What is the simplest/smallest instance of the problem?

> Both the number of items remaining to choose from and the remaining capacity of the knapsack are parameters which define the size of the problem — fewer items and/or less capacity in the knapsack get closer to having no more choices to make (because there aren't any more items to consider or there's no more space).

Approaches. Identify the particular flavor, if applicable, as well as the two patterns (process input and produce output) and what they look like for this problem.

> This is a find-a-subset problem.
>
> Process input would be to consider each item and decide whether or not to put it in the pack. Produce output would be to identify the next item to put in the pack.

Process input involves n yes-or-no decisions, so the running time will be at least $O(2^n)$. Produce output involves up to n items considered and up to n choices to get to a complete solution — $O(n^n)$. While this may be overcounting quite a lot, the number of alternatives is not likely to be ≤ 2 very often, so most likely process input will be more efficient. We'll proceed with that approach.

Generalize / define subproblems. Friends get smaller versions of the original problem, which often takes the form of a generalized version of the original problem.

Partial solution. Identify what constitutes a solution-so-far. This will be the same kind of thing as a legal solution for the whole problem, but less complete (fewer choices have been made).

The set of items taken so far.

Alternatives. Identify the choice being made and what the (legal) alternatives are for that choice.

Process input: the choice is whether or not to include the next item and the alternatives are "add to the pack" or "don't add to the pack". Add is only legal if the item fits in the pack.

Produce output: the choice is which item to add to the pack next. The legal alternatives are all items not yet in the pack which fit.

Subproblem. The "rest of the problem". This will be the same kind of problem (task, input, output) as the original, but generalized — instead of "solve the problem from scratch", "solve the rest of the problem given a solution-in-progress".

Since solving the rest of the problem depends only on how much space is left in the pack and not the particular items already chosen, we don't need to pass along the partial solution as such, just the remaining capacity.

`knapsack'(S',W')` — find the highest-value subset of items in S' whose total weight does not exceed the remaining capacity W' of the knapsack

Input: set S' of items left to consider, remaining (unfilled) capacity of the knapsack W'
Output: items chosen from S', total value of those items
Legal solution: a subset of items with total weight $\leq W'$
Optimization goal: highest value

Note that the friends only return the items chosen in solving their own subproblems.

Base case(s). A complete (and legal) solution has been reached — identify what should be done in that case.

There are no items left (S' is empty) — `knapsack'`($\{\}$,W'). Return ($\{\}$,0) — choose no items, for a total value of 0.

It is not necessary to include the case of no room left (`knapsack'`(S',`0`)) as this and the case of there being space but no remaining items fit are handled by the main case — there won't be any legal alternatives so no subproblems passed on to friends.

Main case. This takes the form of: for each legal alternative for the current decision, a friend solves the remainder of the problem given the solution so far plus that choice.

Process input approach, chosen based on running time prospects —

```
let s be an element of S'
if $w_s$ <= W'
  (items1,totalvalue1) ← knapsack'(S'-{s},W'-$w_s$) // take s
(items2,totalvalue2) ← knapsack'(S'-{s},W') // don't take s
return the one of (items1+s,totalvalue1+$v_s$) and (items2,totalvalue2)
 with the higher total value
```

Top level. The top level puts the context around the recursion.

Initial subproblem. Specify the inputs and parameters for the initial subproblem — the one whose solution solves the original problem.

knapsack'(S,W) — S is the original set of items, W is the original capacity of the pack

Setup. Whatever must happen before the initial subproblem is solved.

Nothing to do.

Wrapup. Whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

Return the items.

Special cases. Make sure the algorithm works for all legal inputs — revise the previous steps to add handling for special cases as needed.

If all of the items fit in the pack, take them all. This will be detected in the normal course of the algorithm, but could be checked specifically at the first step to greatly improve performance in that case without much impact to the running time when the items don't all fit.

The case of none of the items fitting will be detected in the first main case, so no need to handle it specially.

Algorithm. Assemble the algorithm from the previous steps and state it.

`knapsack(S,W)` — find the highest-value subset of items in S whose total weight does not exceed the capacity W of the knapsack

Input: set S of n items, each with a value $v_i > 0$ and weight $w_i > 0$; knapsack capacity $W > 0$

Output: a subset of the items

Legal solution: a subset of items with total weight $\leq W$

Optimization goal: maximize total value of items taken

```
(items,value) ← knapsack'(S,W)
return items
```

`knapsack'(S',W')` — find the highest-value subset of items in S' whose total weight does not exceed the remaining capacity W' of the knapsack

Input: set S' of items left to consider, remaining (unfilled) capacity of the knapsack W'

Output: items chosen from S', total value of those items

Legal solution: a subset of items with total weight $\leq W'$

Optimization goal: highest value

```
if S' is empty,
  return ({},0)
else
  let s be an element of S'
  if $w_s$ <= W'
    (items1,totalvalue1) ← knapsack'(S'-{s},W'-$w_s$) // take s
  (items2,totalvalue2) ← knapsack'(S'-{s},W') // don't take s
  return the one of (items1+s,totalvalue1+$v_s$) and (items2,totalvalue2)
   with the higher total value
```

Making progress. Explain why what each of your friends get is a smaller instance of the problem.

The friends get $S' - \{s\}$ so the set of elements is one smaller.

Reaching the end. Explain why a base case is always reached.

The size of S is decreased by one in each subsequent subproblem, so eventually it will be empty.

Establish base case(s). Explain why the solution is correct for each base case.

There aren't any more items to consider, so nothing can be taken and no value accumulated.

Show the main case. Explain why all possible alternatives for the next choice are covered and that the right partial solution is passed to each friend. Then, assuming that the friends return the correct results for their subproblem, explain why the correct answer is produced from those results.

All legal alternatives are considered: Since the solution is a subset of the input items, every item is either in that subset or not. Adding the item to the pack is only considered if there's room for it.

The right subproblems: s is removed from the set of items left to be considered, and its weight is deducted from the remaining capacity if it is taken.

The friends return the items selected to fill the remaining capacity of the pack after s is considered along with the value of those items. We then need to incorporate the selection of s and pick the better option.

Final answer. Explain why the top level — the setup plus a correct solution to the initial subproblem followed by the wrapup — means that the final result is a correct answer to the problem.

The initial subproblem is the original problem, and the subset of elements to include is the desired answer. There is no setup to consider.

Implementation. Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic details that are needed in order to establish the running time.

We pass $S' - \{s\}$ to the friends. Since it doesn't matter what order we consider the elements in, s can be picked however is convenient. We can represent S' with an array `S` containing all of the items and an index `k` — $S' = S[k..n-1]$. Pick `s = S[k]`, thus $S' - \{s\}$ is `S[k+1..n-1]`.

We do need to build up the set of items in the optimal solution, but there's no need to create a copy of either of the friend solutions — just return the appropriate one, possibly with s added.

Time and space. Assess the running time and space requirements of the algorithm, providing sufficient implementation details (data structures, etc) to justify those answers.

With the implementation described, the work other than the recursive calls is O(1).

The branching factor is 2 (take or not take) and the longest path length is n (the number of items), leading to $\Theta(2^n)$ overall.

The produce output approach results in a branching factor of $\leq n-k$, where k is the number of items already chosen, and the longest path length is $\leq n$. Which is better depends on how many items will fit in the pack, but since $n-k \geq 2$ until the very end, the process input approach is likely to be better unless there's only room for a few items.