Chapter 8

Backtracking

Backtracking is a technique for implementing exhaustive search, where (potentially) all possible legal solutions are enumerated. It is applicable when the solution can be formulated as a series of choices building up a partial solution, similar to greedy algorithms, but is used when greedy doesn't work and two or more of the possible next choices must be considered.

Backtracking follows the case analysis pattern for recursive algorithms (section 4.1) — each friend gets a subproblem resulting from a different alternative for the next step of the solution.

8.1 Backtracking Patterns

The same series-of-choices patterns outlined in section 6.2 apply to backtracking algorithms.

Recursive backtracking is employed when exhaustive search is needed — a local strategy for choosing an alternative for each choice is insufficient. There are three common goals:

- **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

Recursive algorithms in general can be categorized by the nature of the subproblem. Two forms are common:

- *solve a smaller problem*, where the subproblem is treated as a standalone instance of the problem to be solved, and
- *solve the rest of the problem*, where the subproblem is solved in the context of actions taken up to this point — solving the subproblem completes a partial solution already begun.

While case analysis typically frames the subproblems in terms of "solve the rest of the problem", both formulations are potentially applicable depending on the nature of the problem.

8.2 Elements of Backtracking Algorithms

8.2.1 Size

Since the algorithm is formulated as a series of choices, the size of the problem is 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).

8.2.2 Subproblems

As with recursion in general, defining subproblems for backtracking algorithms means specifying the task (a generalized version of the original task), the input, and the output.

For case analysis, the original task is "solve the whole problem from scratch" and the subproblems are "solve the rest of the problem given a solution-in-progress". Combined with a series-of-choices formulation, this means that the subproblem task is to solve the rest of the problem after one more choice has been made, given the choices made so far. There is one subproblem for each alternative for the choice.

The subproblem's input is a smaller instance of the problem. As with divide-and-conquer and other recursive algorithms, prefer adding what is needed to specify which part of the input the subproblem is to work with rather than assuming the subproblem works with the whole of a smaller instance — for divide-and-conquer this portion is typically a range of elements in a list or array, while for case analysis the portion is the rest of a list or array so a single index suffices.

In the "solve a smaller problem" perspective, the smaller version of the input is all that is required for the subproblem. The friend returns the solution to the subproblem.

The "solve the rest of the problem" perspective is needed if the friend needs to know something about the solution-in-progress in addition to smaller version of the input. Only pass along what is needed — in particular, don't include the entire solution-so-far as input to the subproblem unless the whole solution-so-far is actually needed. The friend can either return the solution to the subproblem or (if given the whole solution-so-far) the solution to the whole problem.

8.2.3 Base Case(s)

In the "solve a smaller problem" perspective, subproblems are independent instances of overall problem and so the base case is the smallest meaningful problem instance in terms of the input, often size 0 or 1.

In the "solve the rest of the problem" perspective, the base case is when there is nothing left to solve — all of the choices have been made (legally), so a complete and legal solution has been reached. (Since the handling of the last choice is typically the same as the handling of every other choice in the series of choices, it is better to go one extra step and have the base case be when there are no choices left in order to avoid repeated code.)

The action taken in the base case depends on the goal and the recursive perspective.

For the "solve a smaller problem" perspective, the base case result is the answer for the subproblem. For the "solve the rest of the problem" perspective, the base case result can be either the answer for the subproblem or, if the solution-so-far has been passed along to the subproblem, the answer for the whole problem.

For finding a single solution (legal or best), the base case typically returns its answer (subproblem solution or complete solution). For finding all solutions, the base case typically outputs its complete solution or adds it to a collection of solutions.

8.2.4 Main Case

This takes the general form of: for each possible value for the next choice, a friend solves the remainder of the problem given what has been solved so far and that choice. The specifics depend on the goal and the recursive perspective.

Find a legal solution. If the subproblem output is a complete solution:

```
for each legal alternative for the current choice,
  result = solve the subproblem resulting from choosing that alternative

  if result is a solution,
    return it

return no solution
```

If the subproblem output is just the subproblem solution, then `return result+alternative` instead of just the result.

Find the best solution. If the subproblem output is a complete solution:

```
best result so far ← no solution

for each legal alternative for the current choice,
  result = solve the subproblem resulting from choosing that alternative

  if result is a solution and better than the best result so far,
    best result so far ← result

return the best result so far
```

If the subproblem output is just the subproblem solution, then replace `result` with `result+alternative` when checking whether the current alternative results in a better solution and when updating the best result so far.

Find all solutions.

```
for each legal alternative for the current choice,
  solve the subproblem resulting from choosing that alternative
```

The base case handles the complete solutions (outputting them or adding them to a collection of solutions).

8.2.5 Termination and Correctness

The termination argument follows a common pattern: the problem size is the number of choices remaining, progress is made because another choice is made before handing a subproblem off to a friend, and if you keep making choices, eventually you will have made them all (the base case is no choices left to make).

The nature of the correctness argument depends on the recursive perspective.

For base case(s): If the friends produce solutions to subproblems, correctness for the base case(s) means showing that the correct solution for those problem instances is returned. If the friends produce solutions to the whole problem, correctness for the base case(s) requires two things: that the solution-in-progress is a complete, legal solution and that the base case does the right thing with that solution. That the solution-in-progress is a legal solution is the responsibility of the caller passing it to this subproblem — it can be taken as a precondition. That the solution-in-progress is a complete solution is also a given — a solution is complete when there are no more choices to make, and that's exactly what defines a base case.

For the main case: Correctness requires explaining why the desired solution will be found (all possible legal alternatives for the next choice are either considered or are safe to skip), showing that the correct partial solutions and subproblems are handed to the friends, and, if a result is returned, showing that the correct result is returned. If the correct result is the subproblem solution, this means explaining why the result returned is correctly built from the friends' subproblem solutions. If the correct result is the complete solution, explain why the correct complete solution was chosen from amongst the multiple complete solutions returned by the different friends.

Final answer: Correctness includes establishing that the correct input is provided to the initial subproblem — the (empty) solution-in-progress is legal and the rest of the problem is actually the whole problem.

8.2.6 Efficiency

Backtracking is fundamentally an exponential-time approach due to the exponential number of subproblems — case analysis results in a recurrence relation of the form $T(n) = aT(n-b) + f(...)$ where the problem size n is the number of choices to be made, a is the number of alternatives for each choice, and b is 1, making $T(n) = \Theta(a^n)$ as long as $f(n) = \Theta(n^c log^d n)$.

Backtracking can be viewed as a depth-first search of the space of partial solutions — the DFS tree has a *branching factor* b equal to the number of alternatives for each decision and a *longest path* h equal to the maximum number of decisions needed to yield a complete solution. Expressed in these terms, the number of subproblems is $\Theta(b^h)$.

Even though this is exponential, it is still essential to seek to minimize the number of subproblems and the amount of work per subproblem whether that is in the big-Oh sense — $\Theta(nb^h)$ is worse than $\Theta(b^h)$, for example — or just in terms of constant factors.

The series-of-choices patterns (section 6.2) affect the number of choices made (and thus the longest path length) and the number of possible alternatives for each choice (the branching factor). In general, prefer the pattern with the lower growth rate for the longest path length; if they are both $\Theta(n)$ (common), prefer the lower branching factor.

The specifics of what exactly to pass as input and to return as output (and the representations) should be dictated by efficiency. Prefer a representation for "the rest of the problem" which doesn't require copying — add what is needed to specify which part of the input the subproblem is to work with rather than assuming the subproblem works with the whole of a smaller instance. For a process input approach, for example, the input elements are being processed in some order so an index k in an array or list is sufficient — "the rest of the problem" is to process input elements $k..n$.

For the solution-in-progress, identify what is relevant for the main case and base case(s) and pass only that to the subproblem. The subproblem needs the actual solution-in-progress if the output is the complete problem.

8.3 How to Design Backtracking Algorithms

Backtracking algorithms are recursive algorithms so the development process builds on the recursive template (section 4.3) with a few additions/modifications:

Identify avenues of attack.

- *Paradigms and patterns.*
  Consider the patterns defined in section 8.1. This includes the series-of-choices patterns the backtracking patterns, and the recursive perspectives.
- *The series of choices.*
  Identify the series of choices (section 6.4.1). Take into account efficiency considerations when choosing between alternatives.

Define the algorithm.

- *Generalize / define subproblems.*
  Identify several elements:
  - *Partial solution.*
    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).
  - *Alternatives.*
    What are the legal alternatives for the next choice?
  - *Subproblem.*
    Define the subproblem — the task, the input, and the output. Write this as a function definition where parameters and return values are explicitly identified. Take into account efficiency considerations when defining the intput and output.

Determine efficiency.

- *Room for improvement.*
  Algorithm design decisions — the series of choices pattern, the specific input and output for the subproblems — should have already been considered, but revisit them if not.

That makes the full process as follows:

Establish the problem. Defining — and understanding — the precise task to be solved is an essential prerequisite for algorithm development.

- *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, identify and distinguish between legal solutions and optimal ones.
- *Examples.*
  If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

Identify avenues of attack. Determine the algorithmic approach(es) to try.

- *Known targets.*
  Backtracking algorithms are fundamentally exponential.
- *Paradigms and patterns.*
  Consider the patterns defined in section 8.1. This includes the series-of-choices patterns the backtracking patterns, and the recursive perspectives.
- *The series of choices.*
  Identify the series of choices (section 6.4.1). Take into account efficiency considerations when choosing between alternatives.

Define the algorithm. Assemble the pieces of the algorithm according to the template for the particular paradigm and pattern, culminating in a statement of the algorithm itself.

- *Size.*
  The size of the problem is the number of choices left to make. The smallest problem size is 0 (a complete solution).
- *Generalize / define subproblems.*
  Identify several elements:
  - *Partial solution.*
    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).
  - *Alternatives.*
    What are the legal alternatives for the next choice?
  - *Subproblem.*
    Define the subproblem — the task, the input, and the output. Write this as a function definition where parameters and return values are explicitly identified. Take into account efficiency considerations when defining the input and output.
- *Base case(s).*
  A complete solution has been found — address what to do with it.

- *Main case.*
  Address how to solve a typical large problem instance (one that is not a base case). This takes one of the forms outlined in section 8.2.4.

- *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.
  - *Setup.*
    Whatever must happen before the initial subproblem is solved.
  - *Wrapup.*
    Whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

- *Special cases.*
  Make sure the algorithm works for all legal inputs — identify the cases that need to be handled and address how that handling is incorporated into the previous steps (if not already accounted for).

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

  There shouldn't be new elements here, instead bring together the base case(s), main case, and top level with any handling needed for special cases and state the whole algorithm.

Show termination and correctness. Show that the algorithm produces a correct solution.

- *Termination.* Show that the recursion — and thus the algorithm — always terminates.

  - *Making progress.*
    Explain why what each of your friends get is a smaller instance of the problem: in each step, another choice is made.
  - *The end is reached.*
    Explain why a base case is always reached: eventually all choices have been made.

- *Correctness.* Show that the algorithm is correct.

  - *Establish the base case(s).*
    Explain why the solution is correct for each base case: the right thing is done with a complete solution.
  - *Show the main case.*
    Explain why the desired solution will be found (all possible legal alternatives for the next choice are either considered or are safe to skip) and showing that the correct partial solutions and subproblems are handed to the friends.
  - *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. For the initial subproblem, establish that the correct input is provided – the (empty) solution-in-progress is legal and the rest of the problem is actually the whole problem.

Determine efficiency. Evaluate the running time and space requirements of the algorithm.

- *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.
- *Time and space.*
  Assess the running time and space requirements of the algorithm given the implementation identified.
- *Room for improvement.*
  Algorithm design decisions — the series of choices pattern, the specific input and output for the subproblems — should have already been considered, but revisit them if not.