Algorithms: Design and Implementation

Mr. Mattie

21st June, 2024

Contents

1 Algorithms: Design and Implementation

Correct Thinking leads to correct Code!

Algorithms, even the ones that seem simple can be very difficult to design and implement. Purely technical algorithms frequently have mistakes in them until they have been debugged over a significant period of time. Business Logic algorithms are unique to the domain and there is little guidance to help the developer.

The biggest failing is not one of coding, but rather a failure of imagination to conceive of all the scenarios that the algorithm is executed in. Not realizing that a list might be empty, or a input not supplied can wreck an algorithm.

We will go through a systematic process that is a discovery processs, discovering all the ways the algorithm will be used and stretched. The code is intuitive once all the facets of the problem domain have been discovered.

1.1 Definition

Defining your goals, your problem, and clearly stating your assumptions will do more for your design process than any other step. Your code reflects your assumptions more than your skill in most cases.

1.1.1 SCENARIOS

State & Clarify – The context of CASES and EXPECTATIONS

(Inputs, Expections, Outputs). The Inputs, Outputs, and Expectations are the CASES of the algorithm. The EXPECTATIONS are what the algorithm should do in that CASE. Scenarios are the situational and practical constraints surrounding the CASES. The Expectations are usually a ”result”, but it can include side-effects: changing the environment.

It is important to ask questions and clarify these scenarios. They are the foundations of the entire process and a mistake here will ripple all the way to implementation in a very costly way.

1.1.2 BEHAVIOR

State & Clarify

BEHAVIOR is the actions that lead to the EXPECTATIONS being met by the algorithm. It is vital to clarify the behavior and make sure it covers all facets of the environment, cases, and results.

It is important to clarify behavior because if it doesnt thread through the scenarios it will result in a broken implementation.

1.1.3 INPUTS

State & Clarify - Types and Scale

The type, scale, and possible anomolies in the inputs to the algorithm have a huge impact on the algorithm. Designing something for one thousand elements is a very different from designing for one million elements. A thousand will fit easily to memory, a million elements is a different design entirely.

1.1.4 RETURN

State & Clarify - Types and Scale

The types and results that EXPECTATIONS requires are crucial to the design process. Early in the computation the assembly of output should begin.

1.1.5 CHANGE

State & Clarify - Before and After

Sometimes the algorithm must make a change in the environment. This is less-desirable from a design and implementation standpoint but if it is the EXPECTATION then it must be done well.

If a change must be made it is best to make the algorithm idempotent, or where repeated calls have the same result. For example: a light button as a toggle will turn on/off the lights every time it's pressed. A switch or paddle pressed "off" will result in off every time. That is idempotent.

1.2 Design (Iteration)

1.2.1 CASES and Experiments

Brainstorm the different CASES in the context of an environment. Conduct thought experiments, Give up on bad ideas quickly. See if the CASES enumerate all the scenarios, how common they are: common case, corner case.

1.2.2 Operations (API)

Maximize Idempotent side-effect free operations [1]

Breakdown the cases into functions, and try and maximize side-effect free idempotent functions. Where there is state handle it carefully defining the entire life-cycle of the state.

1.2.3 Isolate Operations (API)

Functions are isolated, minimal sharing of state between functions (API)

Try and avoid cross-connection between functions, so they can be tested in isolation and more easily understood than tangled functions.

1.3 Paradigm

Solution Comprehension

At this point you should have a collection of functions, with a description of the part they play in the algorithm. Next is paradigm.

Paradigm is what model best describes the problem (dynamic greedy, lazy, streams, Relational, divide and conquer) and most efficiently produces an answer.

Spot check the paradigm against the CASES to see if it adequately describes the problem. Find the right paradigm.

1.3.1 Recursion

$$\theta(\log_n) \tag{1}$$

Recursion is elegant and compact. In languages that support it it is practical as well as simple and transparent.

1. recurrence

   Distill the problem down into a solution that can be applied to all the elements.

2. termination

   Define the base case or **termination** as return of the solution that unwinds the recursion.

1.3.2 Divide & Conquer

$$\theta(n * \log_n) \tag{2}$$

Divide and Conquer is a technique where the problem is dived into parts, each part is solved, and then the sub-solutions are combined into the complete solution.

1. Divide the problem into n/x parts.

   Decide the granularity of the division.

2. Solve each part

   Solve the sub-problem. The reduced scale of n reduces the complexity or run time of the solution.

3. Combine the solutions for the final solution

   With each sub-problem solved combine the solutions into a final solution.

1.3.3 Dynamic

Dynamic Programming uses a technique of caching answers to frequently computed problems.

Memoization [5] is a powerful technique and in Python the ”functools” package has a LRU [6]

1.3.4 Linguistic (DSL)

DSL stands for Domain Specific Languages. Thes can be simple declarative language processors, or full blown domain specific languages like ”R” [10]. They can be used to define complex problems and organize the problem into something more easily solved, like a parse tree.

1.3.5 Query

Query Languages like SQL can go beyond transactional into the space of analytical queries either providing processing of data, or even computations such as ”GROUP BY” and MIN and MAX in SQL [11].

The underlying model behind relational databases is the Relational Algebra [4]

1.3.6 Logic

Logic systems are basically rule systems like Prolog [9] They are used in mathematical and logic applications. Their solution finding approach can also be useful in solving difficult problems like cross-wiring network links for redundancy and expert systems.

1.3.7 Single Pass

Single pass approaches are significant when the data set is so large it cannot be contained in memory. These kinds of problems are becoming more important as the size of data in general skyrockets.

1.3.8 Multi-Pass

Sometimes huge gains can be made by making multiple passes. This is basically a variant on Dynamic Programming. Database Indexes. When data is queried the location can be found quickly in the index instead of a full table scan.

Sorting ahead of time is another example, making possible a Binary Search technique.

1.3.9 Pre-Compute

Pre-Computing unlike multi-pass where the complete problem set is traversed, is instead the compilation of tables that are expensive to compute.

In the early days of computing the computation of sine/cosine and other graphic operations were prohibitely expensive.

Since the answers were a small table pre-computing the equations greatly sped up programs. Bitmaps were even compiled to machine code for faster rendering.

1.3.10 Multi-Process

There is an entire field of programming dedicated to muli-process computing. It is based upon parallel computation which is currently in vouge, due to the large number of cores on CPU's and the use of massively parallel dedicated chips like video cards.

It's even possible to crack passwords, do machine learning, and mine crypto currencies on dedicated chips.

1.3.11 Dynamic Programming

Applied to recursion is (descent + memoization) recursively can be no cycles in the DAG of the recursion, or it will get into an infinite loop. It is fundamentally a brute force approach, good for computing min/max style answers.

1.3.12 Greedy Programming

Greedy algorithms, like the parser compiler packer function I wrote in my Emacs Parser Compiler used a greedy technique with push back to maximally fill functions with code [8].

1.3.13 Lazy Programming

When the computation may not be needed or when the problem cannot fit into memory it can be lazy loaded, or lazy computed.

1.3.14 Streams

Streams [2] are a finite sequence of discrete elements of the same type processed in a linear sequence of operations. They are produced by a generator function which allows a subset of the stream to be computed.

2 Sketch the Code

Sketch the code in functions, loops, with comments on purpose and O-notation complexity

1. **Initialize**: establish a return value, empty containers over nulls
2. **Terminate**: determine the base case. When is it done?
3. **First, Common, Last Cases**: The basic sequence of the algorithm
4. **Corner**: cases
5. **Input Validation**: System errors, stale state, deadlocks, and sync errors, timeouts
6. **State**: initialize, update, delete [1]

3 Data Structures

3.1 Array

Typed and indexed they are extremely fast with O(1) read/write for any element. Insert is very slow as the array elements have to be copied to make room for each insertion. The equal cost of access to any element makes algorithms like binary search, and some sorting algorithms possible.

3.2 List

Single or Double Linked lists have efficient inserts but perform poorly in most cases.

Counting length or adding to end is $\theta(n)$

3.3 Trees

Good for storing hierarchal data and a natural fit for recursive algorithms, trees require only $\theta \log_n$ to find an element.

Performance is maintained only when the tree is balanced, re-balancing on insert can be an expensive operation.

3.4 Stack/LIFO

Last In First Out

Stacks are an excellent structure for back-tracking problems. They are LIFO, or Last In First Out. They can be used as a substitute for recursion, and generally for back-tracking.

3.5 QUEUE FIFO

First In First Out

Good for processing in chronological order. It can also be used for a breadth traversal of a tree.

3.6 Hashes

A bread and butter data structure used pervasively to look up non-integer keys in $\theta(1)$ complexity.

References

[1] H. Abelson, G.J. Sussman, and J. Sussman. In: *Structure and Interpretation of Computer Programs, second edition*. MIT Electrical Engineering and Computer Science. Cost of Assignment, Mutable State. MIT Press, 1996, pp. 175–178. ISBN: 9780262510875. URL: `https://books.google.com/books?id=iL34DwAAQBAJ`.

[2] H. Abelson, G.J. Sussman, and J. Sussman. In: *Structure and Interpretation of Computer Programs, second edition*. MIT Electrical Engineering and Computer Science. Stream Model. MIT Press, 1996, pp. 243–275. ISBN: 9780262510875. URL: `https://books.google.com/books?id=iL34DwAAQBAJ`.

[3] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs, second edition*. MIT Electrical Engineering and Computer Science. MIT Press, 1996. ISBN: 9780262510875. URL: `https://books.google.com/books?id=iL34DwAAQBAJ`.

[4] Edgar F Codd. "A Relational Model of Large Shared Data Banks (1970)". In: (2021).

[5] T.H. Cormen et al. In: *Introduction to Algorithms, fourth edition*. MIT Press, 2022, pp. 391–399. ISBN: 9780262367509. URL: `https://books.google.com/books?id=RSMuEAAAQBAJ`.

[6] T.H. Cormen et al. In: *Introduction to Algorithms, fourth edition*. MIT Press, 2022, pp. 803–806. ISBN: 9780262367509. URL: `https://books.google.com/books?id=RSMuEAAAQBAJ`.

[7] T.H. Cormen et al. *Introduction to Algorithms, fourth edition*. MIT Press, 2022. ISBN: 9780262367509. URL: `https://books.google.com/books?id=RSMuEAAAQBAJ`.

[8] Michael Mattie. *parser.el*. 2008. URL: `https://github.com/bitcathedrals/grail/blob/main/archive/parser.el`.

[9] Wikipedia. *Prolog*. 2024. URL: `https://en.wikipedia.org/wiki/Prolog`.

[10] Wikipedia. *R Programming Language*. 2024. URL: `https://en.wikipedia.org/wiki/R_(programming_language)`.

[11] Wikipedia. *SQL*. 2024. URL: `https://en.wikipedia.org/wiki/SQL`.