01. Problem Solving for Algorithm Design

CPSC 535

Kevin A. Wortman

CALIFORNIA STATE UNIVERSITY
FULLERTON

Overview

Strategies for designing algorithms in the context of

- novel scholarly research
- industrial software development *(algorithm engineering)*
- technical interviews
- class exercises
- exam questions

Commonalities

- partial credit
- another person to talk to

A Checklist

1. **Understand** the problem definition
2. **Baseline** algorithm for comparison
3. **Goal** setting: improve on the baseline how?
4. **Design** a more sophisticated algorithm
5. **Inspiration** (if necessary) from patterns, bottleneck in the baseline algorithm, other algorithms
6. **Analyze** your solution; goal met? trade-offs?

1. Understand

- Read the problem definition (input/output) carefully and critically.
- Make sure that you understand the data types involved, constraints on them, technical terms, and story of the problem.
- Write out a straightforward input and output. Try additional examples involving non-obvious solutions.
- What are the *corner cases* of input? Can the input be empty? When is the solution obvious, and when is it not?
- What is a use case for this algorithm in practical software?
- Does this problem resemble other problems you know about?
- If any of this is unclear, **ask questions**.

Example Problem: Top 25 Words [1]

input: a document D (list of n strings) and set S of stop-words (strings)
output: a list of pairs $T = \{(w, k) : w \in D, w \notin S, \text{ and } k \in \mathbb{N}\}$ containing the 25 most frequently-occuring words, not including stop-words, and the number of times each appears

Think about:

- Data types, constraints, of input/output
- Example input and output
- Corner cases
- Use case
- Similar to anything familiar?

[1] Credit: Exercises in Programming Style, Christa Lopes

2. Baseline

Now that you understand the problem, quickly identify at least one algorithm that solves it.

- (research context) literature review: CLRS, Wikipedia, Google Scholar
- sketch a **naïve algorithm**: brute force, exhaustive search, for loops
- What is the efficiency of your baseline algorithm?
- Baseline alg. almost always exists
  - in our contexts, problems are almost always in *NP*
  - $\implies$ exponential-time exhaustive search algorithm
- In the unlikely event no alg. seems to exist, consider whether the problem is provably undecidable.

Stop after the Baseline?

Consider: baseline algorithm is a working, but possibly very inefficient, solution to your problem.

- might be acceptable in S.W.E. if efficiency is low priority
- shows technical interviewer that you can communicate and think about algorithms
- earns partial credit on class assignments
- time/labor management

Pareto principle (80/20 rule): in many settings, spending 20% of maximum effort achieves 80% of maximum benefit

3. Goal

What about the baseline do we want to improve?

- better **time efficiency** (most common goal)
- better space efficiency
- avoid randomization
- avoid amortization
- extra feature e.g. sort is in-place
- simplicity; elegance; easier to implement/understand
- better constant factors

(non-research contexts): goal may be dictated to you

4. Design

Now, design an algorithm, hopefully achieving your goal.

Start with verbal discussion; informal sketches; snippets of pseudocode, prose, equations

As ideas develop and become more concrete, move to pseudocode.

Finally write complete, clear pseudocode for the entire algorithm.

"Devil is in the details:" resist urge to avoid a tricky part, often that is the key to meeting your goal.

5. Inspiration

Step 4. Design...

1. *might* come naturally, devise an algorithm effortlessly
2. *more likely:* not immediately apparent how to proceed; stuck at first

Case 2 is

- more likely
- a learned skill
- requires practice, effort, experience
- point of class exercises
- why algorithm design is an in-demand skill

Sources of Inspiration: algorithm design patterns

- greedy
- divide-and-conquer
- randomization
- reduction to another algorithm (sorting) or data structure (hash table, search tree, heap, etc.)
- dynamic programming

Run through the list; can you think of how to use any of these patterns to solve the problem at hand?

If a pattern doesn't seem to work, why is that? Might give a hint about what would work instead.

Sources of Inspiration: Identify Bottleneck

Review the analysis of your baseline algorithm (either from the literature, or what you just devised).

Identify the dominating term in the efficiency analysis.

Trace that term backwards to a part of the baseline algorithm; probably a particular loop or data structure operation.

How can we do less work there?

- preprocessing (ex. maximum subarray)
- use an appropriate data structure (ex. heap sort)
- reuse work instead of repeating work (ex. Dijkstra's alg.)
- dynamic programming

Sources of Inpiration: Other Algorithms

One reason we study specific algorithms in detail is to learn about clever "tricks" other designers have used, that we might be able to use in novel circumstances.

Examples

- define invariants to keep yourself organized (selection sort)
- break down problem into simpler phases (heap sort)
- use pointers so you can change many paths w/ one assignment; redirection (search trees)
- use an array instead of pointers (heapsort, open addressing)
- master theorem insights to refactor work out of dominating term (selection)
- when almost all candidate solutions are clearly right/wrong, randomize (quicksort, universal hashing)
- compute word-at-a-time (hashing, radix sort)

6. Analyze

Finally analyze your algorithm.

Does it meet your goal? (time efficiency, space efficiency, randomization, etc.)

If yes, obviously done.

If not,

- Is your algorithm still an improvement over your baseline?
- Is more effort justified? (Pareto principle.)
- If so go back to prior steps.