CPSC 535: Advanced Algorithms

Instructor: Dr. Doina Bein

General Computational Problems

- A problem has an input and an output
- A solution to the problem can be an algorithm that takes the input and produces the output after a finite number of steps
- Def: Given an alphabet Σ, a *computational problem* is a function mapping strings defined on Σ to set of strings on Σ,
- Computational problem ≠ decision problem
- Any particular input to a problem is called an *instance* of the problem.

Categories

- All the computational problems discussed in this course fit into the definition, but in real-world they are stated as questions or tasks, not functions
- Based on their task, we divide them into:
  - Function problems
  - Search problems
  - Optimization problems
  - Threshold problems
- Each computational problem may have an associated *decision problem*

Function Problems

- A *function problem* is characterized by a function f(I) that returns a single string as output, where I is the input string, and is defined as follows:

  Given input string I, compute f(I) (or return "no" if f(I) is not defined).

- The output of a *function problem* is always a singleton string (we consider "no" to be a singleton string, too).

Example 1: Multiplication à la russe

- Read more on Internet about it, also called “Ethiopian multiplication” or “Peasant multiplication” (https://en.wikipedia.org/wiki/Ancient_Egyptian_multiplication , https://www.wikihow.com/Multiply-Using-the-Russian-Peasant-Method)
- The way most people learn to multiply large numbers looks something like this:

```
  86
 x 57
------
  602
+ 4300
------
 4902
```

- This "long multiplication" is quick and relatively simple.
- For the Ethiopian / à la russe / Russian peasant algorithm, one does not need multiplication; one only needs to double numbers, cut them in half, and add them up, but it will take longer to get the result.

How it works?

- **Follow the rules**:
- Write each number at the head of a column, it does not matter which column; let's consider that the first number is in the left column and the second number in the right column.
- Halve the number in the left column and double the number in the right column as follows:
  - If the number in the left column is odd, divide it by 2 and drop the remainder.
  - If the number in the left column is even, cross out that entire row.
- Keep halving, doubling, and crossing out rows until the number in the left column is 1.
- Add up the remaining (uncrossed) numbers in the right column.
- The total is the product of your original numbers.

Why it works?

- Let m and n be the two numbers

- If m is even, then the first row will contain

  m n
- And the second row will contain

  m/2 2*n
- Note that m*n=(m/2)*(2*n).
- The first row will be crossed out and the result will be computed based on second (possibly) and the subsequent rows.

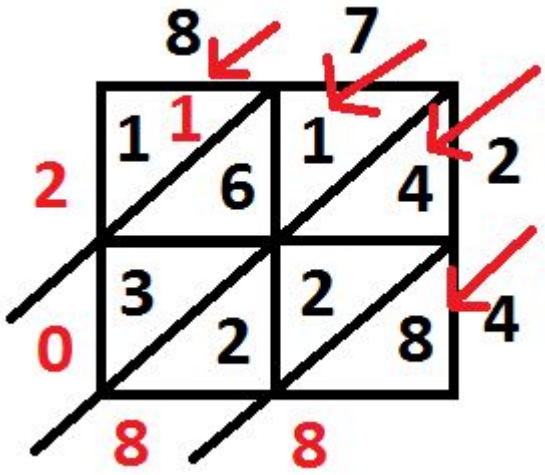
- If m is odd, then the first row will contain

  m n
- And the second row will contain

  (m-1)/2 2*n
- Note that m*n = m+((m-1)/2)*(2*n).
- The first row will be kept and the result will be computed based on first, second (possibly) and the subsequent rows.

Example 2: Lattice Multiplication

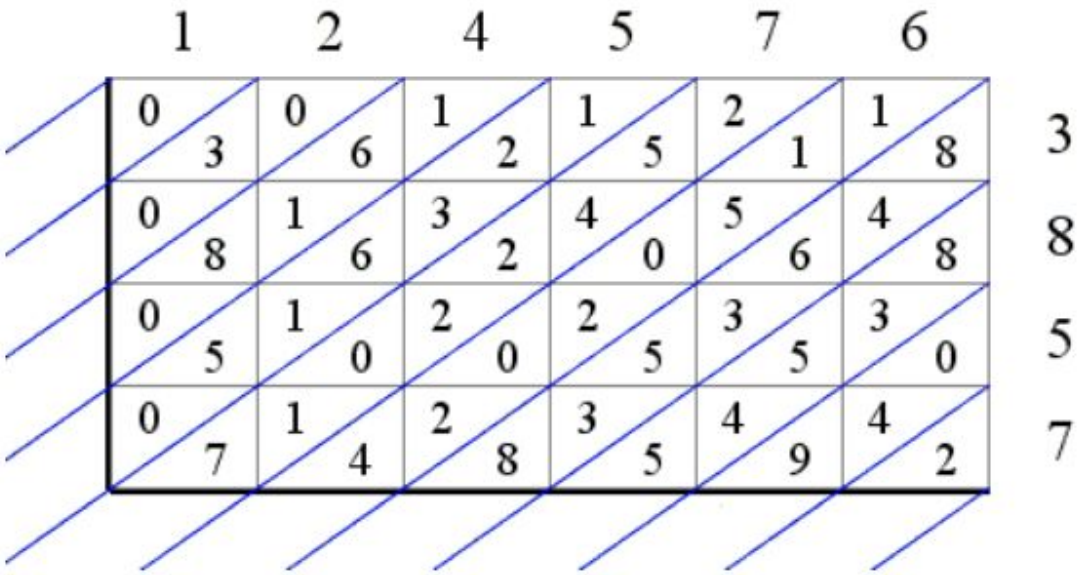
- Read more on Internet about it, known under many names, most common “lattice multiplication” (https://en.wikipedia.org/wiki/Lattice_multiplication)



How it works?

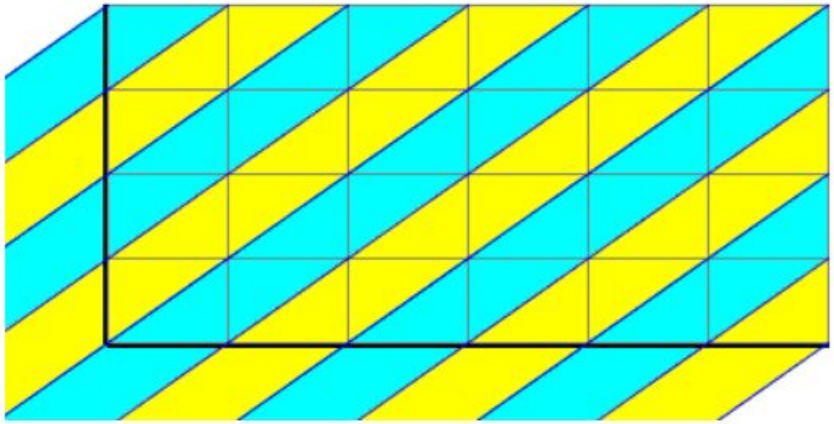
(https://www.cut-the-knot.org/Curriculum/Arithmetic/LatticeMultiplication.shtml)

- The number with the smallest number of digits goes on the row and the number with the largest number of digits goes on the column, one digit per cell.
- A grid is drawn up, and each cell is split diagonally.
- The two multiplicands of the product to be calculated are written along the top and right side of the lattice, as below:

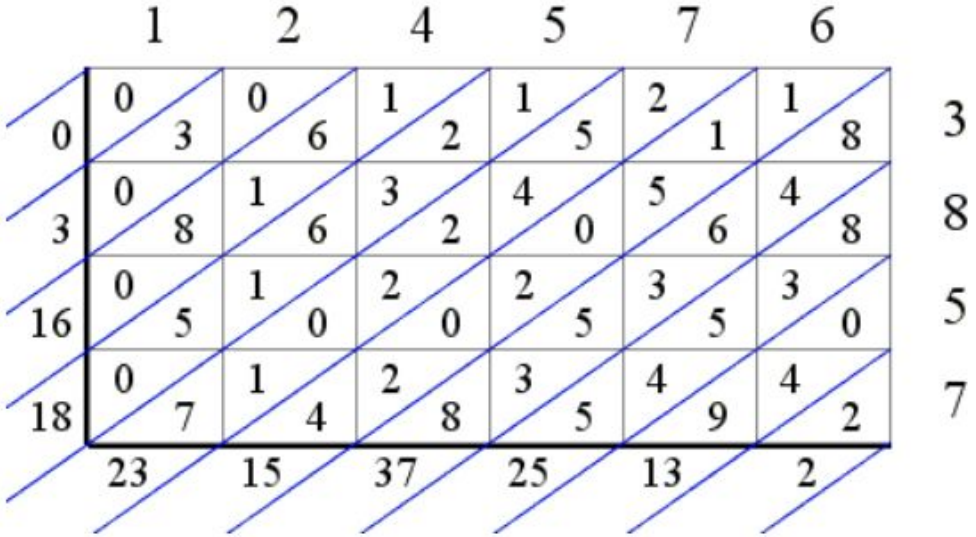


(contd.)

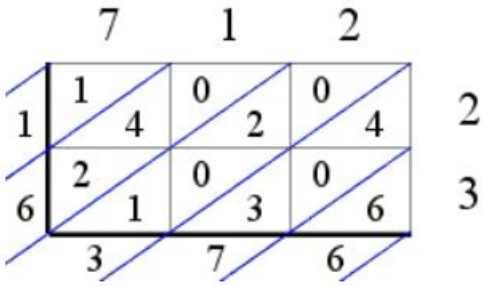
- Note that the diagonals split the lattice into (diagonal) bands:



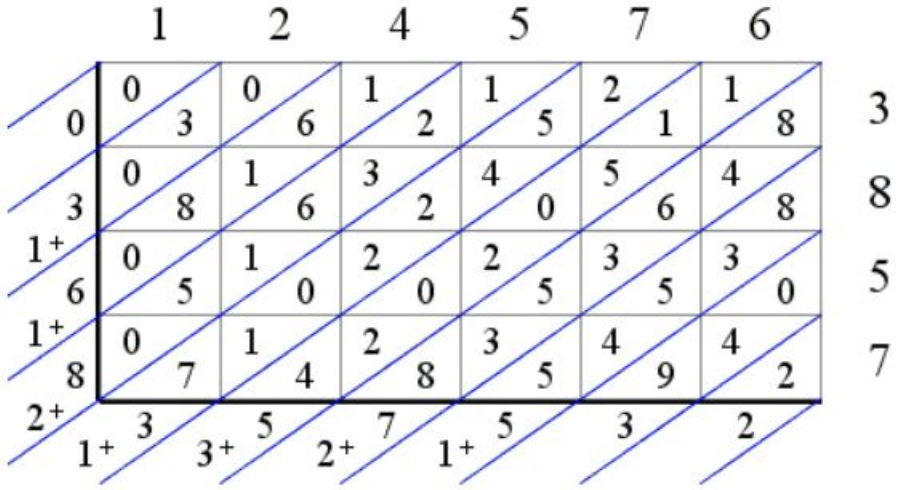
- The algorithm requires to compute the sums of all the digits in a band and place the result next to the lattice, to the left or below the bottom, as the case may be.
- The product 124576×3857 leads to the following diagram:



- The sums are thought of as being ordered around the lattice, first from the top to the bottom and then left to right.
- If all the sums are single digit numbers, the product of the two multiplicands can be read right away by following the sums around the lattice *counterclockwise*, so 712×23 = 16376:



- The 2-digit sums cause a complication which is resolved by carrying their first digit to the previous band (counterclockwise) and adding it to what remains there of the sum placed there previously.
- The product 124576×3857 leads to the following diagram:



- The third intermediate sum 2 + 8 = 10 takes two digits leading to an additional carry of 1 which is added to the preceding sum 1 + 6 making it 1 + 6 + 1 = 8.
- Once all the sums are single digit numbers, the product of the two multiplicands can be read right away by following the sums around the lattice counterclockwise.
- We obtain 124576×3857 = 480489632.

Search Problems

- A predicate is a function that returns true or false.
- A search problem is characterized by a predicate Q(I,S) where I is the input string and S is some string, as follows:

  Find a string S such that Q(I,S) is true (or return "no" if no such S exists).

Optimization Problems

- A *numerical function* is a function that returns numbers, usually integer-valued.
- An *optimization problem* is characterized by a numerical function V(I,S) where I is the input string and S is some string, as follows:

  Find a string S such that V(I,S) is minimized (or maximized).

Threshold Problems

- A *threshold problem* is characterized by a numerical function V(I,S) and a numerical threshold K, where I is the input string, K is an input numerical value, and S is some string, and is defined as follows:

  Given I and K, find a string S such that V(I,S) ≤ K (or ≥, or =).

- Another variant asks a decision version of the same question:

  Given I and K, does there exist an S such that V(I,S) ≤ K (or ≥, or =)?

Decision Problems

- A *decision problem* is a function problem with exactly two possible solutions, “yes” or “no”, i.e. it is characterized by a function f(I) that returns “yes” or “no”, where I is the input string, and is defined as follows:

  Given the string I, returns “yes” if f(I) exists or “no” if f(I) is not defined.
- Are rarely used in practical applications, but they are relatively easy to state and prove theorems about it.
- Most general computational problems have a closely related decision problem that captures the same fundamental idea, and vice-versa.

Strategies to Design Algorithms

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

- Why?
  - novel scholarly research
  - industrial software development (algorithm engineering)
  - technical interviews
  - class exercises
  - exam questions
- Follow a checklist (next)

Checklist

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

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

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

- 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: Word Find [1]

Problem description: A popular diversion in the United States, “word find” (or “word search”) puzzles ask the player to find each of a given set of words in a square table filled with single letters. A word can read horizontally (left or right), vertically (up or down), or along a 45 degree diagonal (in any of the four directions) formed by consecutively adjacent cells of the table; it may wrap around the table’s boundaries, but it must read in the same direction with no zigzagging. The same cell of the table may be used in different words, but, in a given word, the same cell may be used no more than once. Write an algorithm for solving this puzzle.

Quiz:

- Data types of input and output, constraints on these data types (if any)
- Problem instance, i.e. example of input and output
- Are there any special cases, so called “Corner cases”?
- Use case, i.e. where solving this problem can be useful
- Similar to anything familiar?

[1] Introduction to the Design and Analysis of Algorithms. By Anand Levitin, ex. 10, page 107

2. Baseline

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

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 naive 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
  - → 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 is possibly a very inefficient solution to your problem.

- might be acceptable 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

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

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

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

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

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

Step 4. Design...

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

Inspiration:

- 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

Patterns taught in CPSC 335:

- 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 Inspiration: 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

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

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 yes, go back to prior steps.