CMSC 351: Algorithm Design

Justin Wyss-Gallifent

February 23, 2021

1 Introduction 2
2 General Steps 2
3 Algorithm Design 4
4 Thoughts, Problems, Ideas 6

1 Introduction

This is not, per se, a course in algorithm design. In theory that is meant to be part of CMSC451. However it's worth introducing some elements of algorithm design which we can keep in mind and revisit as we discuss the algorithms in this course.

2 General Steps

Algorithm design can be broken down a variety of ways but here is one possible way that we can think about how we might design an algorithm to solve a certain problem.

1. Problem statement: Come up with a rigorous statement of what the problem is.

   Typically the first step is fairly straightforward but often edge cases can complicate things. If the problem involves a set of numbers we might ask how we should respond if the set is empty. If the problem involves returning the index of an entry in a database we might ask how we should respond if there are multiple results.

2. Model development: How can we represent the problem using a model which lends itself to analysis?

   The second step often takes some mathematical knowledge. Is our data best represented in a list? An array? A linked list? A heap? How? Note that this is less a computer science question than a mathematics one.

3. Algorithm specification: What should the algorithm do, explicitly and rigorously? It's at this stage where we nail down how the real-world problem manifests in the algorithm.

   This step is typically a mash-up of the first two since it involves rephrasing the first step in terms of the model we've chosen.

4. Algorithm design: Now we figure out how to actually design the algorithm.

   Typically this culminates in the pseudocode and to some degree is merged with the next step in the sense that checking correctness can be considered a part of design. Together these two steps are often the most challenging.

5. Checking correctness: Once we've designed an algorithm how can we check if it's correct?

   If it's not then we need to go back and fix our algorithm.

6. Algorithm analysis: Typical analysis include running time in big-$\mathcal{O}$ notation as well as memory usage.

   It's at this point where we might decide that our algorithm is not sufficient. Perhaps the time complexity is not good enough or the memory usage is too high. This could mean we need to go back to the algorithm specification state or even re-think the development of the model.

7. Algorithm implementation: Now we get to actually implement the algorithm in code.

   This is to some degree the "computer programming" stage rather than the "computer science" stage.

8. Program testing: Once we have code then we can do real-world tests not just to see the algorithm in action but to see if we've overlooked anything.

Example 2.1. Suppose we wish to write an algorithm to find the maximum of a set of numbers. Here's how we might go through the stages.

1. Problem statement: We have a set of numbers and we wish to find the maximum. These numbers coule be positive or negative and could be integers. For now we'll avoid real numbers and we'll assume the set is nonempty.

2. Model development: We'll store the numbers in an array because this lends itself well to computation. Thus our model is an array of numbers.

3. Algorithm specification: The algorithm needs to somehow examine all the numbers and figure out the largest.

4. Algorithm design: One idea - we'll go through the numbers one by one, keeping track of and updating a variable which will contain the largest so far. To jump-start the process we'll assign the largest so far to be the first one. Here's the pseudocode:

```
\\ PRE: The array A contains all the numbers.
max = A[0]
for i = 1 to len(A)-1
    if A[i] > max
        max = A[i]
    end
end
\\ POST: The variable max contains the maximum.
```

5. Checking correctness: We could introduce a loop invariant such as:

   `LI(i):` When `i` iterations have completed, `max` contains the maximum of `A[0,...,i]`.

   We would then check this loop invariant.

6. Algorithm analysis: If there are n numbers then the length of the array is n and the loop iterates $n-1$ times. Before the loop we have an assignment which takes c_1 time and inside the loop we have a check and possible assignment which takes c_2 time. Thus we have total time requirement $c_1 + c_2(n-1) = \mathcal{O}(n)$.

7. Algorithm implementation: In Python something like:

```
max = A[0]
for i in range(1,len(A)):
    if A[i] > max:
        max = A[i]
```

8. Program testing: We could, for example, stress-test by giving the algorithm thousands of sets of numbers to make sure it never has any crashes and to spot-check some of the output. If we have other reliable ways of finding the maximum we could compare our output against that output.

3 Algorithm Design

The algorithm design stage is very often the most challenging. Some classic notions that arise are as follows. This is not at all comprehensive and these are not mutually exclusive. To help digest the ideas we'll work through a real-world example.

1. Linear: Can we simply step through the data item by item?

   Example 3.1. You have a bunch of unlabeled boxes in your attic. You're looking for a bottle-opener so you have to check every box in turn.

2. Divide and Conquer: This is recursive and realted to decrease and conquer. Here we divide into smaller problems and tackle each of those and combine the results. Examples: Merge sort, quick sort.

   Example 3.2. You have two friends who are willing to help. Each of you takes one third of the boxes and looks through them one by one.

3. Decrease and Conquer: This is recursive and related to divide and conquer. Here we reduce the problem to a single smaller version of itself. Examples: Finding factorials, binary search.

   Example 3.3. You were smart enough to use red boxes for kitchen items so you can focus just on those.

4. Greedy Algorithms: Greedy algorithms make the best choice at each stage but may not make the best overall choice. Typically this approach to a problem is useful when best local solutions are good enough or when they do, in fact, lead to best global solutions.

Example 3.4. Open a box and find something that'll open a bottle. It might not be a bottle-opener but that's fine.

5. Brute Force: This almost always works but the time required can be prohibitive.

   Example 3.5. Look at every item carefully. Is it a bottle-opener?

6. Dynamic Programming: Remembering the past and applying it to the future.

   Example 3.6. Maybe at the start of all this you don't know what a bottle-opener is. After you try opening a bottle with whatever you find you start developing a better idea what a bottle-opener looks like and it makes subsequent attempts easier.

7. Randomized Algorithm: There may be values which need to be assigned as part of the algorithm. It's not uncommon to make random choices at this stage, if only because non-random choices are often biased in some way.

   Example 3.7. Just grab whatever, at random, out of any box. Why not? Could be a bottle-opener.

4 Thoughts, Problems, Ideas

1. Each of the following algorithm descriptions is lacking at least one aspect of rigor or clarification. Identify.
    (a) An algorithm which finds the index of the maximum number in a list of numbers.
    (b) An algorithm which sorts a list of numbers.
    (c) An algorithm which rotates a square array by $90°$.
    (d) An algorithm that takes three positive integers a, b, n and finds the first n digits of the decimal expansion of a/b, returning them in a list.

2. Which of the following models could work for the problem given. Model choices include:
    - An n-dimensional array (specify n)
    - An undirected unweighted graph
    - A directed weighted graph
    - An undirected weighted graph
    - A directed weighted graph
    - some combination thereof if it seems reasonable

    Justify.

    (a) An algorithm which finds the shortest route in a highway network.
    (b) An algorithm which finds the longest period of daily increase in TSLA stock.
    (c) An algorithm which finds patterns in temperature data throughout a city.
    (d) An algorithm which finds weaknesses in a small computer network.

3. Suppose the only storage mechanism you have is three stacks `S`, `T`, and `U`, so no arrays, lists, etc. For a stack `S` you can `S.push(x)` and `x=S.pop`, and `S.pop` will return `NULL` if a stack is empty. Suppose you have a list of distinct numbers contained in a stack `S` and you wish to sort them into `U`. Explain how you could do this.

4. Repeat the previous question but now you have only two stacks `S` and `T` and you want the numbers sorted back into `S`.

5. This algorithm should find the spread (max-min) of a list but something is wrong. What?

```
\\ PRE: A[0,...,n-1] is a list.
spread = -inf
for i = 0 to n-2
    for j = i+1 to n-1
        if A[i] - A[j] > spread
            spread = A[i] - A[j]
        end
    end
end
\\ POST: spread found.
```

6. Suppose an algorithm needs to take a value n and return the list:

$$\{0!, 1!, 2!, 3!, ..., (n-1)!\}$$

However you don't have access to a factorial function so you need to calculate those manually. How is this problem related to dynamic programming?

7. Suppose $f(n)$ is defined for all $0 \leq n \leq 1000000$. Suppose an algorithm needs to find a local maximum of $f(n)$. In other words it needs to find an n_0 such that $f(n_0 - 1) < f(n_0)$ and $f(n_0 + 1) < f(n_0)$. How could a greedy approach aid in this?

8. Suppose in the previous question "local maximum" is replaced with "maximum". How could some degree of randomization be introduced in order to get a reasonable (although perhaps not perfect) result?

9. In what way could your local post office be considered as doing a divide-and-conquer approach?

10. Often conditionals involving too many conjunctions and/or negations can be confusing. In such cases some rewriting can simplify the situation. Rewrite each of the following as a simple conditional with an expression of the form `if X COMPARISON Y` where `comparison` is one of $\geq$, $\leq$, $>$, $<$, $==$ or $! =$ and where `X` and `Y` are basic arithmetic expressions. The first one has been done for you.

    (a) `if A and B have opposite signs`
    Solution: `if A*B < 0`

    (b) `if A and B have the same sign`

    (c) `if at least one of A,B is zero`

    (d) `if neither A nor B is zero`

    (e) `if both A and B are zero`

    (f) `if at least one of A and B is nonzero`

    (g) `if (A+B>0 and C-D>0) or (A+B<0 and C-D<0)`

    (h) `if (A+B>0 and C-D<0) or (A+B<0 and C-D>0)`

11. Often access to data is restricted by architecture. For example when you pull data from a database via an API you can only retrieve it in chunks and for purposes of storage and bandwidth the size of those chunks might be restricted. As such you may need to plan your algorithm around this.

    For each of the following assume you have a list `A` whose length `n` is a multiple of 5. You cannot access the entries in `A` directly. Instead you have a separate working memory `R[0,...,4]` which you can access directly. You also have the commands `in(i)` which copies `A[i,...,i+4]` to `R[0,...,4]` and `out(i)` which copies `R[0,...,4]` to `A[i,...,i+4]`. This is called a *block transfer*. In general you want to minimize the number of block transfers you make and you must be careful since neither command called on an index `i` may spill over the end of `A`.

    Using this restriction as well as $\Theta(1)$ auxiliary space, write pseudocode which does each of the following. The first is done for you.

    (a) Prints each element in the list.
    Solution: One possible solution:

    ```
    blockcount = n/5
    for i = 0 to blockcount-1
        in(5*i)
        for j = 0 to 4
            print(R[j])
        end
    end
    ```

    (b) Finds the minimum of the list.

    (c) Counts the number of occurrences of a specified value.

    (d) Passes through the list once from left to write and increases each value by 1.

    (e) Replaces all occurrences of a specified value by another specified value.

    (f) Passes through the list once from left to right and interchanges any adjacent entries which are out of order.

    (g) Reverses the list. Do this first with the assumption that `n` is a multiple of 10.

12. Repeat question 11 under the slightly different constraints:

    - The functions `in` and `out` take a second value `k` between 1 and 5 inclusive so that `in(i,k)` copies `A[i,...,i+k-1]` into `R[0,...,k-1]` and `out(i,k)` does the reverse.
    - The list length is not necessarily a multiple of 5.

13. Repeat question 11 under the slightly different constraint:

    - The list length is not necessarily a multiple of 5 but is at least 5.