CMSC 351: Algorithm Types

Justin Wyss-Gallifent

May 26, 2023

1 Introduction 2
2 Divide and Conquer 2
3 Decrease and Conquer 3
4 Dynamic Programming 3
5 Greedy Algorithms 4
6 Comparisons and Thoughts 4

1 Introduction

Before proceeding to look at some specific algorithms we will have a high-level discussion on different approaches to problems along with a simple example of each.

It's certainly worth noting that many algorithms use a combination of these approaches and some don't necessarily use them at all, so think of these as some broad-strokes approaches.

It should also be noted that these approaches overlap somewhat when we describe them so try not to think of them as overly rigid.

2 Divide and Conquer

In a divide and conquer approach we typically divide our problem into two or more sub-problems. We then solve each of those subproblems and combine the two or more solutions.

This is different from decrease and conquer in that we solve all of the subproblems.

Example 2.1. Here is a divide and conquer approach to finding the maximum element in a list. Note that this is not at all a practical approach, this is simply demonstrative.

If the list is length 1 then we return the single element, otherwise we split the list in half (using floor or ceiling to handle odd-length lists) and take the maximum of the maximums of each half.

In pseudocode this would be something like the following, where $A[x : y]$ includes $A[x]$ but not $A[y]$ as with Python:

```
function max(A):
  if len(A) == 1:
    return A[0]
  end if
  maxleft = max(A[0:floor(n/2)])
  maxright = max(A[floor(n/2):n])
  if maxleft > maxright:
    return maxleft
  end if
  return maxright
end function
```

Note 2.0.1. Algorithms such as binary search are often put into the divide and conquer category but this is starting to change. The reason is that binary search splits the list in half but does not examine both halves. The more modern name

for this is decrease and conquer.

3 Decrease and Conquer

In a decrease and conquer approach we typically reduce our problem to a single problem of a smaller size.

Example 3.1. Consider the calculus problem of approximating a root of a continuous function using the bisection method. Suppose we have a continuous function f and two x-values L and R with $f(L) < 0$ and $f(R) > 0$. Continuity guarantees some $x \in (L, R)$ with $f(x) = 0$. We wish to approximate the values of x to within ϵ of the actual value.

The bisection method approaches the problem by finding the midpoint $M = (L + R)/2$ and checking $f(M)$. If $f(M) < 0$ then the root is in (M, R) and so we set $L = M$ and repeat, whereas if $f(M) > 0$ then the root is in (L, M) and so we set $R = M$ and repeat. We do this until $R - L < 2\epsilon$ and then return $M = (L + R)/2$. At any stage if $f(M) = 0$ we simply return M.

Notice when we do this we cut the interval in half and then we isolate our next step to only one of those halves.

Observe that unlike divide and conquer we only look at one of the sublists.

Divide and conquer (as with decrease and conquer) does not require that we cut our problem in half. We could divide it into three or more subproblems and solve one or we could simply reduce the size of the problem somehow. We will see examples of these later.

4 Dynamic Programming

In a dynamic programming approach we build up the solution from previously calculated values.

Example 4.1. Consider the problem of calculating the n^{th} Fibonacci number. Recall they are defined as:

$$f_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f_{n-1} + f_{n-2} & \text{if } n \geq 2 \end{cases}$$

If we wish to calculate f_n for some arbitrary n we will need to calculate a succession of previous values.

Note: There is a closed formula for the Fibonacci numbers, so that can be used instead in theory, but it is problematic due to approximations which may occur.

5 Greedy Algorithms

In a greedy approach we typically make the best choice possible at each stage with the hope (but no guarantee) that this leads to an optimal result.

> **Example 5.1.** Suppose I am playing Scrabble. A greedy approach would be that every time it is my turn I simply calculate the most points I can get with my letters on the board and make that move.
>
> Note that this is not how professionals play Scrabble and is not optimal! Rather they often play moves to block their opponents and they often use just one or two letters from their own rack in hopes of getting replacement letters which will allow them to use all of their letters on their next move; this is called a bingo and earns a 50-point bonus.

6 Comparisons and Thoughts

Some cues that might help us see whether some of these might work are as follows:

- Divide and conquer: Can a quick check help us reduce the problem to two or more smaller problems which can be solved and their solutions combined?
- Decrease and conquer: Can a quick check help us reduce the problem to a significantly smaller problem?
- Dynamic programming: Is our goal to get a result as efficiently as possible?
- Greedy: Is our goal to get a (hopefully reasonably good) result as quickly as possible, even if we are willing to sacrifice some precision?