Decrease-and-Conquer

Madiba Hudson-Quansah

Contents

Chapter 1

Introduction

The decrease and conquer technique, exploits the relationship between a solution to a given instance of a problem and a solution to its smaller instance. Once such a relationship is established, it can be exploited either top down or bottom up. The top down approach naturally leads to a recursive implementation, also known as divide and conquer. The bottom up variation is usually implemented iteratively, starting with the smallest solvable instance of a problem and building up to the whole problem. This is also known as the **incremental approach**. There are three major variants of decrease and conquer:

- Decrease by a constant.
- Decrease by a constant factor.
- Variable size decrease.

1.1 Decrease by a constant

The size of the problem instance is reduced by the same constant on each iteration of the algorithm, usually by one. Taking the example of the exponentiation problem, a^n where $a \neq 0$ and $n \in \mathbb{N}$, the relationship between a solution to a reduced problem instance, $n-1$ and n is obtained easily:

$$a^n = a \times a^{n-1}$$

This allows the function $f(n) = a^n$ to be computed top down, recursively, by:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ f(n-1) \times a & \text{if } n > 0 \end{cases}$$

Or bottom up, iteratively, by:

$$f(n) = \prod_{i=1}^{n} a$$

1.2 Decrease by a constant factor

The size of a problem instance is reduced by the same constant factor on each iteration of the algorithm, usually by a factor of two. Using the exponentiation problem again we can observe that the relationship between a solution to the reduced problem instance, $n/2$ and n is:

$$a^n = \left(a^{\frac{n}{2}}\right)^2$$

But considering only integer values of n, this does not hold for odd values of n. If n is odd, we have to compute for $n-1$ instead, i.e.

$$a^n = \left(a^{\frac{n-1}{2}}\right) \times a$$

This allows the function $f(n) = a^n$ to be computed top down, recursively, by:

$$a^n = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 & \text{if } n \text{ is even and positive} \\ \left(a^{\frac{n-1}{2}}\right) \times a & \text{if } n \text{ is odd} \\ 1 & \text{if } n = 0 \end{cases}$$

Or bottom up, iteratively, by:

$$a^n = \begin{cases} \prod_{i=1}^{n/2} a \times a & \text{if } n \text{ is even and positive} \\ \prod_{i=1}^{(n-1)/2} a \times a & \text{if } n \text{ is odd} \\ 1 & \text{if } n = 0 \end{cases}$$

In both cases the number of operations done each step reduces by a factor of two, making the algorithm run in $\Theta(\log n)$ time.

1.3 Variable size decrease

The size of the problem instance is reduced by a variable amount on each iteration of the algorithm. An example of this is Euclid's algorithm for finding the greatest common divisor of two numbers:

$$\gcd(m, n) = \gcd(n, m \mod n)$$

In this algorithm the reduction in the size of the problem instance depends on the size of the numbers m and n

Chapter 2

Insertion Sort

Definition 2.0.1: Insertion Sort

For an array of n elements $A\,[0\ldots n-1]$, we assume that the smaller problem instance of $A\,[0\ldots n-2]$ has already been solved, giving us a sorted array of size $n-1$, $A_0 \leqslant \ldots \leqslant A_{n-2}$. Taking advantage of this, we just have A_{n-1} left unsorted. We then scan the sorted subarray from the right to find the correct position for A_{n-1} and insert it there. This is repeated for each increasing subarray until the entire array is sorted, i.e.:

$$A_0 \leqslant \ldots \leqslant A_{n-i} \mid A_i,\, A_{n-1}$$

Where i is the number of passes through the array so far. Although presented as a top down approach, it is more efficiently implemented as a bottom up approach, i.e. iteratively, starting with A_1, and building up to A_{n-1}.

Algorithm 1 InsertionSort (A, n)

```
▹
▹ Sorts a given array by insertion sort
▹ Input: An array $A$ of $n$ orderable elements
▹ Output: Array $A$ sorted in nondecreasing order

for $i \leftarrow 1$ to $n-1$ do
    $v \leftarrow A_i$
    $j \leftarrow i-1$
    while $j \geqslant 0$ and $A_j > v$ do
        $A_{j+1} \leftarrow A_j$
        $j \leftarrow j-1$
    end while
    $A_{j+1} \leftarrow v$
end for
```

The basic operation of this algorithm is the comparison $A_j > v$, and the number of comparisons done depends on the nature of the input, i.e. if the array is already sorted the number of comparisons is $n-1$ as the while loop will never iterate but the outer for loop will always run the whole length of the array. In the worst case, where the array is sorted in descending order, the number of comparisons done is the largest amount possible, i.e. for every element encountered the while loop will iterate j times, where j is the index of the element in the array. Calculating this:

$$
\begin{aligned}
C(n) &= \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 6 \\
&= 6 \sum_{i=1}^{n-1} (i-1) + 1 \\
&= 6 \sum_{i=1}^{n-1} i \\
&= 6 \left(\frac{n(n-1)}{2} \right) + 1 \\
&= 3n^2 - 3n + 1
\end{aligned}
$$

Therefore the worst case time complexity of the insertion sort algorithm is $\Theta\left(n^2\right)$.

Chapter 3

Topological Sorting

Definition 3.0.1: Topological Sort

Given a directed acyclic graph $G = (V, E)$, a topological sort is a linear ordering of the vertices such that for every directed edge (u, v) from vertex u to vertex v, u comes before v in the ordering. This is useful in scheduling tasks that have dependencies on each other.

Topological sorting can be done in a decrease and conquer manner by the source removal algorithm. This algorithm works by identifying in a remaining digraph a **source** vertex, i.e. a vertex with an in-degree of 0 and delete it with the edges originating from it. This is repeated until all vertices have been removed. The order these vertices are removed in is the topological sort of a digraph.

Chapter 4

Generating Combinatorial Objects

4.1 Generating Permutations

In generating permutations we assume that the underlying set whose elements we want to permute is the set of integers from 1 to n, i.e., they can be represented as the indices of elements in an n-element set $\{a_1, \dots, a_n\}$.
In breaking the task of generating all $n!$ permutation of the set $\{1, \dots, n\}$, we can consider all $(n-1)!$ permutations have been generated. Assuming this we can relate the task of generating all $n!$ permutations to augmenting this set of $(n-1)!$ permutations with the nth element in all possible positions, i.e.:

$$n! = (n-1)! \times n$$

We can insert n in the previously generated permutations either left to right or right to left. It is more efficient to start with inserting n into $1, 2, \dots (n-1)$ by moving right to left, i.e. inserting n at the end of each permutation and switching direction every time a new permutation of $\{1, \dots, n-1\}$ needs to be processed, i.e. for $n = 3$:

start	1		
insert 2 right to left	12	21	
insert 3 right to left	312	132	213
insert 3 left to right	321	231	213

It is also possible to get the same ordering of permutations of n without explicitly generating all permutations of for smaller values of n. This is done by associating a direction with each element k in a permutation. This element k is said to be **mobile**, if its direction points to a smaller element adjacent to it. This defines the **Johnson-Trotter** algorithm for generating permutations:

Algorithm 2 JohnsonTrotter (n)

```
                                                                                    ▹
▹ Implements Johnson-Trotter algorithm for generating permutations
▹ Input: A positive integer $n$
▹ Output: A list of all permutations of $\{1, \dots, n\}$

1: Initialize the first permutation with $1, 2, \dots, n$ with all directions pointing left
2: while the last permutation has a mobile element do
3:     find its largest mobile element $k$
4:     swap $k$ with the adjacent element it is pointing to
5:     reverse the direction of all the elements greater than $k$
6:     add the new permutation to the list of permutations
7: end while
```

4.2 Generating Subsets

The task of generating all subsets of a set $A = \{a_1, \ldots, a_n\}$, can be done in a decrease and conquer manner by dividing all the subsets of A into two groups:: those that do not contain a_n and those that do. The former group is all the subsets of $\{a_1, \ldots, a_{n-1}\}$, while each and every element of the latter can be obtained by adding a_n to a subset of $\{a_1, \ldots, a_n\}$.

Chapter 5

Decrease-by-a-Constant-Factor

5.1 Binary Search

Definition 5.1.1: Binary Search

In a sorted array A of n elements we first compare the search key K with the middle element of the array A_m. If they match, the algorithm stops, otherwise, the same operation is repeated recursively for the first half of the array if $K < A_m$, and for the second half if $K > A_m$:

$$A_0 \ldots A_{m-1}\, A_m\, A_{m+1} \ldots A_{n-1}$$

Algorithm 3 BinarySearch (A, n, K)

```
▷
▷ Implements nonrecursive binary search
▷ Input: An array $A$ of $n$ elements sorted in ascending order and a search key $K$
▷ Output: An index of the array's element that is equal to $K$ or $-1$ if there is no such element

$l \leftarrow 0$
$h \leftarrow n - 1$
while $l \leq h$ do
    $m \leftarrow \lfloor (h + l) \rfloor / 2$
    if $A_m = K$ then
        return $m$
    else if $A_m > K$ then
        $l \leftarrow m + 1$
    else
        $r \leftarrow m - 1$
    end if
end while
return -1
```

The analysis of binary search is focused on the number of times its basic operation, the comparison of the value and the search key, is done. This number varies depending on the nature of the input. In the worst case $C_{\text{worst}}(n)$, the input array does not contain the search key, or the search key is the last element of the array. Since after one comparison the algorithm is left with an array of half size we get the following recurrence relation:

$$C_{\text{worst}}(n) = C_{\text{worst}}(\lfloor n/2 \rfloor) + 1 \text{ for } n > 1,\ C_{\text{worst}}(1) = 1$$

We can solve this using the master theorem:

$$O(\log n)$$

5.2 Fake-Coin Problem

Definition 5.2.1: Fake-Coin Problem

Among n identical-looking coins, one is fake. With a balance scale, we can compare any two sets of coins, i.e. by tipping to the left to the right, or stating even, the balance scale will tell whether the sets weigh the same or which of the sets is heavier than the other but not by how much.

We can solve this problem in a Decrease-by-a-Constant-Factor manner by:

- Dividing n coinst into two piles of $\lfloor n/2 \rfloor$ coins each , leaving one extra coin aside if n is odd.
- Put the two pile on the scale. If the piles weigh the same, the coin put aside must be fake, otherwise, we can proceed in the same manner with the lighter pile which must contain the fake coin.

This gives us the following recurrence relation:

$$W(n) = W(\lfloor n/2 \rfloor) + 1 \text{ for } n > 1,\ W(1) = 0$$

This is identical to the binary search cost relation since both algorithms are based on the same Decrease-by-a-Constant-Factor technique of decreasing by a factor of 2. This leads to the close form of this recurrence also being:

$$W(n) = \lfloor \log n \rfloor$$

5.3 Russian Peasant Multiplication

Definition 5.3.1: Russian Peasant Multiplication

Let n and m be positive integers whose product we want to compute, and let us measure the instance size by the value of n. If n is even, and instance of half the problem size has to deal with $n/2$. This gives us this formula relating the smaller $n/2$ multiplication to the larger problem instance

$$n \cdot m = \frac{n}{2} \cdot 2m$$

If n is odd, we need to account for that , i.e.:

$$n \cdot m = \frac{n-1}{2} \cdot 2m + m$$

Thus our base case becomes $n = 1$, i.e. $1 \cdot m = m$

Chapter 6

Variable-Size-Decrease

6.1 Computing a Median and the Selection Problem

Definition 6.1.1: Selection problem

The problem of finding the kth smallest element in an array of n elements. This number is also known as the kth order statistic.

The median is a special case of the selection problem where $\lceil n/2 \rceil$, which asks to find an element divides the array into two equal halves. This can be done by sorting the array and selecting the middle element, but this is not efficient as it takes $\Theta(n \log n)$ time.
We can instead use the idea of partitioning a given list around some value p, for example the first element. Then we can identify all the elements less than p and greater than p to determine it's position in the sorted list. Using the notion of **Lomuto partitioning**, we think of the subarray, $A[l \ldots r]$, as being divided into three parts:

$$p \mid \text{less than } p \mid \text{greater than } p \mid \text{unprocessed}$$

I.e. a segment with elements known to be less than p, a segment with elements known to be greater than p, and a segment of elements yet to be compared to p.