Transform And Conquer

Madiba Hudson-Quansah

Contents

Chapter 1

Introduction

Transform and conquer is a general algorithm design strategy that is based on the following steps:

Transform The problem instance is modified to make it more amenable to solution.

Conquer The transformed problem is solved.

There are three major forms of this strategy:

1. Transformation to a simpler / more convenient problem / Instance Simplification
2. Transformation to a different representation of the same problem instance / Representation Change
3. Transformation to an instance of a different problem for which a solution is known / Problem Reduction

Chapter 2

Instance Simplification

2.1 Presorting

Presorting stems from the observation that many problems are easier to solve if the input is sorted. The general approach is to sort the input and then solve the problem on the sorted input. For example the problem of determining element uniqueness in an array can be solved by sorting the array and then checking for duplicates:

Algorithm 1 CheckUniqueness (A)

```
▷
▷ Determines if all the elements in the array $A$ only appear once
▷ Input: An array $A$ of size $n$ of orderable elements
▷ Output: Returns true if $A$ has no duplicates and false otherwise

1: Sort $A$
2: for $i \leftarrow 0$ to $n - 2$ do
3:     if $A_i = A_{i+1}$ then
4:         return false
5:     end if
6: end for
7: return true
```

The running time of this algorithm is the sum of the time spent to sort the array and the time spent to check for duplicates.

2.2 Balanced Search Trees

Chapter 3

Representation Change

3.1 Heaps and Heap sort

Definition 3.1.1: Heap

A heap is a complete binary tree that satisfies the heap property, which is the value of each root node is more extreme or equal to the values of its child nodes, according to some total order. Technically these are conditions that should be met:

- The Shape property - A heap is a complete binary tree.
- The parental dominance / Heap property - The value of each node is more extreme than or equal to the value of its children.

The properties of a heap are:

- There exists exactly one essentially complete binary tree with n nodes, whose height is equal to $\lfloor \log_2 n \rfloor$
- The root of a heap always contains the most extreme element
- A node of heap considered with all its descendants is also a heap
- A heap can be implemented as an array, where:
  - The parent nodes will be in the first $\lfloor n/2 \rfloor$ positions of the array while the leaf nodes will be in the last $\lceil n/2 \rceil$ positions of the array.
  - The children of a node at position i, $(0 \leqslant i \leqslant \lfloor n/2 \rfloor - 1)$ are in positions $2i + 1$ and $2i + 2$, and the parent of a node at position i will be in the position $\lfloor (i-1)/2 \rfloor$

In constructing a heap from a list of given keys there are two main methods:

- Bottom-up construction - This method starts with a heap of size 1 and then adds elements one by one to the heap until all the elements are added. This method is $O(n)$
- Top down construction - This method starts with an empty heap and then adds elements one by one to the heap until all the elements are added. This method is $O(n \log n)$

3.1.1 Bottom-Up

Algorithm 2 HeapBottomUp $(H[0 \ldots n-1])$

```
▷
▷ Constructs a heap from a given array of elements
▷ Input: An array $H[0 \ldots n]$ of orderable items
▷ Output: A heap $H[0 \ldots n]$

for $i \leftarrow \lfloor (n-1)/2 \rfloor$ down to 0 do
    $k \leftarrow i; v \leftarrow H_k$
    heap ⟵ false
    while not heap and $2k+1 \leqslant n-1$ do
        $j \leftarrow 2k+1$
        if $j+1 < n$ then
            if $H_j < H_{j+1}$ then $j \leftarrow j+1$
            end if
        end if
        if $v \geqslant H_j$ then
            heap ⟵ true
        else
            $H_k \leftarrow H_j; k \leftarrow j$
        end if
    end while
    $H_k \leftarrow v$
end for
```

Examining the time complexity of the algorithm, assuming $n = 2^k - 1$ the largest possible number of nodes occurs on each level. The height of the tree is $\lfloor \log n \rfloor$ and more specifically in this case:

$$\begin{aligned} h &= \lfloor \log n \rfloor \\ &= \log \left(2^k - 1\right) \\ &= \lceil \log (n+1) \rceil - 1 && = k - 1 \end{aligned}$$

Each key on level i is compared with the key on level $i+1$, and in the worst case will travel to the leaf level h. As moving keys down requires two comparisons, one to find the larger child then another to compare with the parent, the total number of comparisons involving a key on level i is $2(h-i)$, i.e. the total number of key comparisons is:

$$\begin{aligned} T(n) &= \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) \\ &= \sum_{i=0}^{h-1} 2(h-i) \times 2^i \\ &= 2(n - \log(n+1)) \\ &= 2n - 2\log(n+1) \end{aligned}$$

Therefore the time complexity of the algorithm is $O(n)$

3.1.2 Top-Down

Algorithm 3 HeapTopDown $(H[0 \dots n-1])$

```
                                                            ▷
▷ Constructs a heap from a given array of elements
▷ Input: An array $H[0 \dots n]$ of orderable items
▷ Output: A heap $H[0 \dots n]$

$L \leftarrow []$                                           ▷ The heap to be constructed
for each element $e$ in $H$ do
    $L_k \leftarrow e$                                      ▷ Where $k$ is the current size of the heap
    $i \leftarrow k$
    while $i > 0$ do
        $j \leftarrow \lfloor (i-1)/2 \rfloor$
        if $L_j \geqslant L_i$ then
            break
        else
            swap $L_i$ and $L_j$
            $i \leftarrow j$
        end if
    end while
end for
return $L$
```

The top down approach involves successive insertions of each element in the array into an already constructed heap. This is less efficient then the bottom up approach as the each insertion is done $\log n$ time and each element of the n elements in the array need to be inserted. Therefore the time complexity of the top down approach is $O(n \log n)$

3.1.3 Extreme Key Deletion

Due to the characteristics of a heap, the most extreme key is always the root of the heap. Therefore to delete the most extreme elements we go through the following steps:

1. Exchange the root with the last element K of the heap
2. Decrease the heap size by one
3. Heapify the new tree by sifting K down the tree until the heap property is restored

Since this operation is done down the height of the tree, the time complexity of this operation is $O(\log n)$

3.1.4 Heap Sort

The heap sort algorithm uses the properties of a heap to sort an array of elements. The algorithm is in two parts:

Heap Construction : Construct a heap from the array of elements

Extreme Deletions : Delete the most extreme element from the heap for $n-1$ times

This results in the elements of the array being deleted in a sorted order. The time complexity of the algorithm is also comprised of these two parts, constructing the heap using the most efficient method takes n and removing all the keys from the constructed heap takes $n \log n$ resulting in a time complexity of $O(n \log n)$

3.2 Horner's Rule and Binary Exponentiation

Chapter 4

Problem Reduction

4.1 Introduction

The main idea behind problem reduction is reducing an problem with an unknown solution to another problem with a known solution.

4.2 Computing the Least Common Multiple

The least common multiple of two numbers a and b is defined as the smallest integer that is divisible by a and b. This problem can be reduced by finding the GCD of the numbers a and b which would obviously be a divisor of both number, then multiplying the two numbers to find the largest common multiple of the two numbers. By dividing the GCD of the two numbers from the product of the two numbers we get the least common multiple of the two numbers.

$$\operatorname{lcm}(a, b) = \frac{a \times b}{\gcd(a, b)}$$

Where the GCD can be found in $O(\log \min(a, b))$ time using the Euclidean algorithm.

4.3 Counting Paths in a Graph

The problem of counting the number of paths between two nodes in a graph can be solved with problem reduction by recognizing that the number of paths between the node i and the node j is equals the (i, j)th element of A^k, where A is the adjacency matrix of the graph and k is the number of edges between the two nodes. This reduces the problem to matrix multiplication which can be solved in $O(n^2.81)$ time.
For example given the adjacency matrix of a graph:

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

To get all the counts of the number of paths between the nodes of length 3 we can calculate A^3:

$$A^3 = \begin{bmatrix} 2 & 3 & 4 & 4 \\ 3 & 0 & 1 & 1 \\ 4 & 1 & 2 & 3 \\ 4 & 1 & 3 & 2 \end{bmatrix}$$