

Overview

1. Background
2. General Algorithm
3. Examples

Background

Divide and Conquer is an approach to designing algorithms. It's a *top-down* approach. This means a large problem is repeatedly split into smaller, more manageable problems. It often uses recursion to split the problems.

Examples:

- Binary Search
- Merge Sort
- Quick sort
- Strassen's Algorithm
- Karatsuba's Algorithm

Many algorithms divide the input into smaller parts, repeatedly, until the solution of the smaller problems can be found easily. To analyze recursive divide and conquer algorithms we need a mathematical tool, recurrence relations. These relations can be used to characterize run-times.

General Algorithm






Divide and Conquer algorithms are generally recursive. The recursion 'bottoms out' when reaching a base case. A General Divide and Conquer algorithm has three steps.

1. Divide the problem into subproblems
2. Conquer subproblems by solving them recursively
3. Combine the solutions from the subproblem(s).

If $f(n)$ is the number of operations required to solve an initial problem, then a Divide and Conquer Recurrence Relation will look like

$$f(n) = \underbrace{a}_{\text{subproblems}} \overbrace{f\left(\frac{n}{b}\right)}^{\text{size}} + g(n) \qquad \text{where } n = b^k$$

With problem size n split into a subproblems of size n/b every time the problem is divided. b is referred to as the *branching factor*. A Binary tree would have a branching factor of 2 because every parent node splits into two child nodes. For simplicity, suppose $b \mid n$.

Level	Size of Problem	Subproblems	Work
0		a^0	$f(n)$
1		a^1	$af\left(\frac{n}{b}\right) + g(n)$
i		a^i	$a^{i-1}[af\left(\frac{n}{b^i}\right) + g(n)] + \sum_{j=0}^{i-1} a^j g\left(\frac{n}{b^j}\right)$
\vdots			
k		a^k	$a^k f(1) + \sum_{j=0}^{k-1} a^j g\left(\frac{n}{b^j}\right)$

Theorem: Suppose $f \nearrow$ (is strictly increasing), $b \mid n$, $a \geq 1$, $b \geq 1$, $c \in \mathbb{R}^+$:

$$f(n) = af\left(\frac{n}{b}\right) + g(n) \quad \text{then,}$$

$$f(n) \in \begin{cases} O(n^{\log_b a}) & \text{if } a > 1 \\ O(\log_b n) & \text{if } a = 1 \end{cases} \quad \text{Rule } n^{\log_b a} = a^{\log_b n}$$

Proof:

$$\begin{aligned}
 f(n) &= af\left(\frac{n}{b}\right) + g(n) && \text{where } n = b^k \\
 &= a \left[af\left(\frac{n}{b^2}\right) + g\left(\frac{n}{b}\right) \right] + g(n) \\
 &= a^2 f\left(\frac{n}{b^2}\right) + ag\left(\frac{n}{b}\right) + g(n) \\
 &= a^2 \left[af\left(\frac{n}{b^3}\right) + g\left(\frac{n}{b^2}\right) \right] + ag\left(\frac{n}{b}\right) + g(n) \\
 &= a^3 f\left(\frac{n}{b^3}\right) + a^2 g\left(\frac{n}{b^2}\right) + ag\left(\frac{n}{b}\right) + g(n) \\
 &\vdots \\
 &= a^k f\left(\frac{n}{b^k}\right) + a^{k-1} g\left(\frac{n}{b^{k-1}}\right) + a^{k-2} g\left(\frac{n}{b^{k-2}}\right) + \cdots + a^0 g\left(\frac{n}{b^0}\right) \\
 n = b^k &\implies f\left(\frac{n}{b^k}\right) = f(1) \\
 \therefore &= a^k f(1) + \sum_{j=0}^{k-1} a^j g\left(\frac{n}{b^j}\right)
 \end{aligned}$$

We have four cases because $a \geq 1$

Case 1: $a = 1$ & $n = b^k$

Case 3: $a > 1$ & $n = b^k$

Case 2: $a = 1$ & $n \neq b^k$

Case 4: $a > 1$ & $n \neq b^k$

Case 1: $a = 1$ & $n = b^k \implies k = \log_b n$,

$$\begin{aligned} f(n) &= f(1) + ck \\ &= f(1) + c \log_b n && \text{since } b^k = n \\ \therefore f(n) &\in O(\log_b n) \end{aligned}$$

Case 2: $a = 1$ & $n \neq b^k \implies b^k < n < b^{k+1} \implies f(b^k) < f(n) \leq f(b^{k+1})$, since $f \nearrow$

$$\begin{aligned} f(b^{k+1}) &= f(1) + c(k+1) && \geq f(n) \\ &= (f(1) + c) + ck \\ &\leq (f(1) + c) + c \log_b n && \text{since } b^k < n \\ \therefore f(n) &\in O(\log_b n) \end{aligned}$$

Case 3: $a > 1$ & $n = b^k \implies k = \log_b n$,

$$\begin{aligned} f(n) &= a^k f(1) + c \sum_{j=0}^{k-1} a^j \\ &= a^k f(1) + c \frac{a^k - 1}{a - 1} \\ &= a^k f(1) + \frac{c a^k}{a - 1} - \frac{c}{a - 1} \\ &= a^k \left[f(1) + \frac{c}{a - 1} \right] - \frac{c}{a - 1} \end{aligned}$$

Note $a^k = a^{\log_b n} = n^{\log_b a}$. The continued equation gives

$$f(n) = C_1 n^{\log_b a} + C_2$$

Where $C_1 = f(1) + \frac{c}{a-1}$ and $C_2 = -\frac{c}{a-1} \implies f(n) \in O(n^{\log_b a})$

Case 4: $a > 1$ & $n \neq b^k \implies b^k < n < b^{k+1} \implies f(b^k) < f(n) \leq f(b^{k+1})$, since $f \nearrow$

We can reduce this equation to the same one seen above, where

$$f(n) = C_1 n^{\log_b a} + C_2$$

Since $f(n) \leq f(b^{k+1})$

$$\begin{aligned} f(b^{k+1}) &= C_1 a^{k+1} + C_2 \\ &= (C_1 a) n^{\log_b a} + C_2 \\ f(n) &\leq (C_1 a) n^{\log_b a} + C_2 \\ \therefore f(n) &\in O(n^{\log_b a}) \end{aligned}$$

$$\therefore f(n) \in \begin{cases} O(n^{\log_b a}) & \text{if } a > 1 \\ O(\log_b n) & \text{if } a = 1 \end{cases}$$

Recall that a = number of subproblems and b = *branching factor*. Therefore, if there is only one subproblem in an algorithm, like in Binary Search, then it will be in $O(\log_b n)$ where b is the amount of branches in the recursion tree. If there is more than one subproblem, like in Merge Sort, then it will be in $O(n^{\log_b n})$.

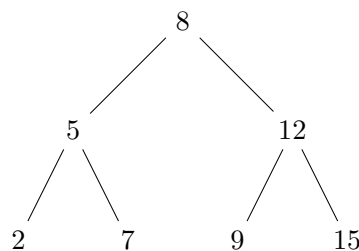
Examples

Binary Search

Problem: Find the index of a element in a sorted list.

2	5	7	8	9	12	15
---	---	---	---	---	----	----

The Binary Search algorithm is used to find items in a *sorted* list in logarithmic time. Instead of using the ‘brute-force’ method for searching, which is just a linear search, it is possible to find items faster by arranging them in a tree structure.



With this method, half of the remaining list is eliminated with each level of the tree traversed. This results in finding the element in $O(\log_2 n)$ time. For a list this size, $n = 7$, so the maximum operations to find an element would be $\lceil \log_2 7 \rceil = 3$, which can be confirmed by analyzing the tree above.

The algorithm for Binary Search is as follows.

Algorithm 1 Binary Search

Inputs:

$A \leftarrow []$

▷ Array with values to search through; **Sorted**

$x \leftarrow$ value to search for

Function binSearch(A, x) = $y \rightarrow$ index of x

$n \leftarrow \text{len}(A)$

if $x == A \left(\left\lfloor \frac{n+1}{2} \right\rfloor \right)$ **then**

return $A \left(\left\lfloor \frac{n+1}{2} \right\rfloor \right)$

else if $x < A \left(\left\lfloor \frac{n+1}{2} \right\rfloor \right)$ **then**

return binSearch($A[1:\text{mid}-1], x$)

else if $x > A \left(\left\lfloor \frac{n+1}{2} \right\rfloor \right)$ **then**

return binSearch($A[\text{mid}+1:n], x$)

else

return -1

▷ If value is not found

end if

Every-Case Time Analysis

The Binary Search Algorithm produces the following recurrence relation. We can assume $b \mid n$

$$T(n) = 1 T\left(\frac{n}{2}\right) + O(1).$$

Then expand the relation using substitution.

$$\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + O(1) \\
T\left(\frac{n}{2}\right) &= T\left(\frac{n}{4}\right) + O(1) \\
T\left(\frac{n}{4}\right) &= T\left(\frac{n}{8}\right) + O(1) \\
T\left(\frac{n}{8}\right) &= T\left(\frac{n}{16}\right) + O(1) \\
&\vdots \\
T\left(\frac{n}{2^k}\right) &= T(1) + O(1) \implies k = \lg n \\
\therefore T(n) &= T\left(\frac{n}{4}\right) + O(1) + O(1) \\
&= T\left(\frac{n}{8}\right) + O(1) + O(1) + O(1) \\
&\vdots \\
&= T(1) + \sum_{k=1}^{\lg n} O(1) \\
&= T(1) + \lg n \cdot O(1) \\
T(n) &\in O(\lg n)
\end{aligned}$$

It is also possible to analyze the recurrence relation through the theorem shown above, where

$$f(n) \in \begin{cases} O(n^{\log_b a}) & \text{if } a > 1 \\ O(\log_b n) & \text{if } a = 1 \end{cases}$$

The relation generated by the Divide and Conquer algorithm has $a = 1$.
Using the theorem, $f(n) \in O(\lg n)$

Merge Sort

Problem: Sort an unsorted list in ascending order.

Algorithm 2 Merge Sort

Inputs:

$A \leftarrow []$

▷ Array with values to sort

$\text{low} \leftarrow 0$

$\text{high} \leftarrow \text{len}(A)$

Function mergeSort(low, high, A) = B → sorted list

if low < high **then**

$\text{mid} \leftarrow \lfloor (\text{low} + \text{high})/2 \rfloor$;

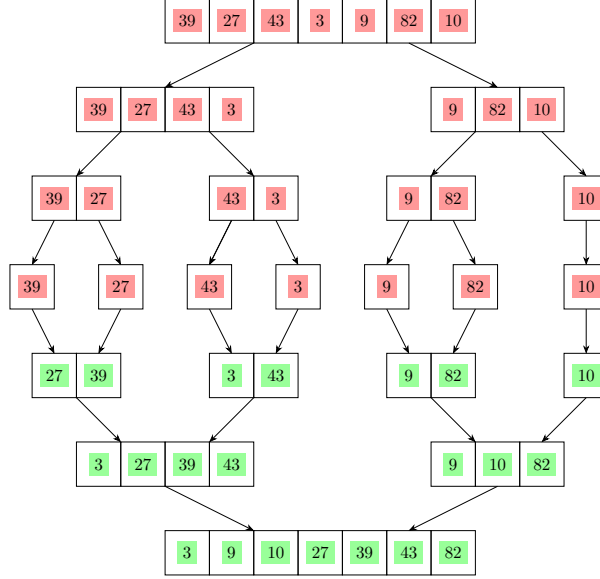
 mergeSort(low, mid);

 mergeSort(mid+1, high);

 merge(low, mid, high);

end if

Merge sort solves the problem by splitting the list into smaller lists until they are sorted. Then, the small, sorted lists are merged back together to make one sorted list. Every time the list is split, it is



split in half. This means the branching factor is 2. Since we still need to deal with both halves of the list, there are also two sub-problems. This means $a = 2$ and $b = 2$.

Merging two lists takes $O(n)$ time as both lists are traversed linearly. This produces the following recurrence relation

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Every-Case Time Analysis

Using substitution it is possible to expand the equation to analyze the time complexity.

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + O(n) \\
&= 2\left[2T\left(\frac{n}{4}\right) + O\left(\frac{n}{2}\right)\right] + O(n) \\
&= 4T\left(\frac{n}{4}\right) + 2O(n) \\
&= 2\left[4T\left(\frac{n}{8}\right) + O\left(\frac{n}{2}\right)\right] + 2O(n) \\
&= 8T\left(\frac{n}{8}\right) + 3O(n) \\
&\vdots \\
&= 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=1}^k O(n) \implies k = \lg n \\
&= \lg n T(1) + \sum_{i=1}^{\lg n} O(n) \\
&= \lg n T(1) + \lg n O(n) \\
T(n) &\in O(n \lg n)
\end{aligned}$$