# Divide & Conquer Big-O Runtime

**Topics:**

- Master Theorem

**Resources:**

- `Node.h`
- `Array.h`
- `main01.cpp`
- `main02.cpp`

## Introduction

Calculating the big-O runtime time for sequential functions are typically straightforward; whereas calculating the big-O runtime for recursive functions may be more challenging, especially for divide and conquer functions.

## Master Theorem

The recursive section of divide and conquer recursive functions are, normally, of the form

$$T(n) = aT(n/b) + f(n)$$

where $a$ is the number of recursive callers in the section, $n/b$ (or $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$ if $b$ does not divide $n$ evenly) is the size of the subcollection, and $f(n)$ represents the runtime of the remainder of the section. To determine (actually, approximate) the big-O runtime of these function, we can use the *master theorem*. The following definition of the master theorem is a paraphrase given that you may not be familar with theta and omega notation, which states

> Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers, $\mathbb{W}$, by the recurrence
>
> $$T(n) = aT(n/b) + f(n)$$
>
> .Then $T(n)$ can be bounded asymptotically as follows
>
> 1. If $f(n) < O(n^{\log_b a - \epsilon})$ , then $T(n) = O(n^{\log_b a})$
> 2. If $f(n) = O(n^{\log_b a})$, then $T(n) = O(n^{\log_b a} \lg n)$
> 3. If $f(n) > O(n^{\log_b a + \epsilon})$ and if $af(n/b) \leq cf(n)$ for some constant $c > 1$ when n is sufficiently large, then $T(n) = O(f(n))$

where $\epsilon > 0$ is a constant and $\lg n = \log_2 n$. Likewise, $\epsilon$ is not intended to exceed 1 (i.e $0 < \epsilon \leq 1$); ultimately, it should be used to get $\log_b a$ to the nearest whole number.

Let us examine the master theorem with a few examples. First, we will look at the `MergeSort()` function, which is defined as follows

```
template<typename T>
void MergeSort(Array<T>& data,int p,int q)
{
 if(p < q)
 {
  int r = (p + q) / 2;
  MergeSort(data,p,r);
  MergeSort(data,r+1,q);
  Merge(data,p,q,r);
 }
}
```

where `Merge()` is defined as

```
template<typename T>
void Merge(Array<T>& data,int p,int q,int r)
{
 int i, lc = 0, rc = 0, ln = r - p + 1, rn = q - r;
 Array<T> L(ln), R(rn);

 for(i = 0;i < ln;i += 1)
 {
  L[i] = data[p+i];
 }

 for(i = 0;i < rn;i += 1)
 {
  R[i] = data[r+1+i];
 }

 for(i = p;lc < ln && rc < rn;i += 1)
 {
  if(L[lc] <= R[rc])
  {
   data[i] = L[lc];
   lc += 1;
  }
  else
  {
   data[i] = R[rc];
   rc += 1;
  }
 }

 while(lc < ln)
 {
  data[i] = L[lc];
```

```
  i += 1;
  lc += 1;
 }

 while(rc < rn)
 {
  data[i] = R[rc];
  i += 1;
  rc += 1;
 }
}
```

The worst-case scenario runtime of the `Merge()` function given that conditions of control structures cost 1 while everything else cost 0 is

$$T(n) = 3n + 4$$

where $n$ represents the distance between $p$ and $q$. Hence, its big-O runtime would be O(n). Now consider the `MergeSort` function. It has two calls to itself in its recursive section; thus, $a = 2$. Likewise, since $r$ is equal to the midpoint of $p$ and $q$, $b = 2$. Therefore, recurrence function is

$$T(n) = 2T(n/2) + O(n)$$

So $O(n^{\log_2 2}) = O(n^1) = O(n)$. This means the `MergeSort()` function falls into case 2 of the master theorem. Therefore, the big-O runtime of the function

$$\begin{aligned} T(n) &= O(n^{\log_2 2} \lg n) \\ &= O(n \lg n) \end{aligned}$$

as required.

For the next example we will be looking at the function `Contains()` which returns the size of a binary tree. Its definition is as follows

```
template<typename>
bool Contains(Node<T>* rt,const T& itm)
{
 if(rt == NULL)
 {
  return false;
 }
 else if(rt->data == itm)
 {
  return true;
 }
 else
 {
  return (Contains(rt->left,itm) || Contains(rt->right,itm));
 }
}
```

For this function, the recursive section calls the function twice and return the or operation of the calls; thus $f(n) = 1$ and $a = 2$. Since, we are dealing with a binary tree, traversing a subtree reduces the content by half (assuming that the tree is full), hence, $b = 2$. Thus the recurrence function is

$$T(n) = 2T(n/2) + 1$$

This means $O(n^{\log_2 2}) = O(n^1) = O(n)$. Thus the function falls into case 1. Therefore, the big-O runtime of `Contains()` is

$$\begin{aligned} T(n) &= O(n^{\log_2 2}) \\ &= O(n) \end{aligned}$$

For the last example, we will not look at an actual function; instead, we will look at the recurrence function which is as follows

$$T(n) = 2T(n/4) + n^2$$

In this example, $O(n^{\log_4 2}) = O(n^{0.5}) \approx O(n^1)$ when $\epsilon = 0.5$. Hence, this example falls into case 3. Therefore, the big-O runtime will be

$$T(n) = O(n^2)$$