Synthesis 1

1. Asymptotic Analysis

1.1 Big O (O)

Big O notation (Big O notation) is a mathematical notation used to describe the asymptotic behavior of a function. More precisely, it describes an asymptotic upper bound on the order of magnitude of a function in terms of another (usually simpler) function. In computer science, the Big O notation is usually used for the complexity analysis of algorithms. Therefore, it is the basis of learning algorithms.

Mathematical Definition:
f(x) = O (g(x)) , if there exists a positive number c and a real number x0 such that
|f(x)| <= c g(x) for all x >= x0;

From my perspective, introducing the Big O notation to computer science largely simplify the calculation of runtime and space complexity and help us better compare the time complexity among different algorithms. For example, if the time complexity of a program is, let’s say $n^3 + n^2 + n$. It looks a bit long and unnecessary, right? After applying Big O notation, we can simply rewrite the time complexity into $O(n^3)$ because the term with the highest power dominates the equation when n grows large enough. And in practical application, especially in Big Data Analytics, n can rise to trillion or even larger. In such a situation, $n^2 + n$ doesn’t matter at all.

Although it may seem fair enough to consider f = O(g) as f <= g, it is completely unjustifiable to make an analogy like this. This analogy fails due to the constant c and the condition x >= x0. For example, $5n^2$ = O(n2) , but $5n^2$ is smaller than n^2, and similarly, $n^2 = O(5n^2)$. However, if f(x) = n^2, g(x) = 2n+5, g(x) can not be larger than f(x) no matter how large the value of c is or how large x0 is. Why is that? To be short, this is simply because f(x) grows much faster than g(x) when x gets increasingly larger even if g(x) is bigger than f(x) in some low-x-value interval. From these two examples, we can conclude that whether f = O(g) is determined by the domination of the function on each side of the equation. if the domination of f is larger than g, then f ≠ O(g) but g = O(f) instead. Likewise, if the domination of f is smaller than g, then f = O(g). If the domination of f is equal to g, then f = O(g) and g = O(f) at the same time which also indicates that f = **ϴ**(g). (We will talk about theta notation in the follow-up section.)

As mentioned above, Big O can help us analyze the complexity of our programs. And here's a list of commonly used notations for analyzing the running time of multiple algorithms.

Notation		Example
O(1)	O(1) is often considered a constant, and the complexity of O(1) is often ignored when getting calculated.	Find a value in the hash set(in an ideal situation)
O(log n)	O (log n) is considered a logarithmic and it's very fast since the magnitude of a logarithmic is not very large.	Binary Seach
O(n)	O(n) is considered a linear solution. The linear solution is good because we don't waste too much time running the program	Traverse a list or an array

O(n log n)	Worse than linear but better than n^2. Still considered a doable algorithm with this complexity.	Merge sort
O(n^2)	O(n^2) is considered a very bad complexity because a quadratic function grows very faster as n gets larger and therefore significantly ruining the efficiency of the algorithm.	Bubble sort

Additionally, there are some other notations larger than O(n^2). Respectively, O(n^c) (power function), O(c^n) (exponential function), O(n !) (factorial function). Under normal circumstances, if our algorithm takes larger than O(n^2), such as O(n!), we'd better find other solutions to the problem because this approach is way too time-consuming.

According to the ranking of the different notations, we can derive an inequality like this: O(n!) > O(c^n) > O(n^2) > O(n log n) > O(n) > O(log n) > O(1). Note that for exponential function(O(c^n)), $(c+1)^n$ dominates c^n and n actually dominates $(\log n)^2$.

Exercise: In each of the following pairs, figure out whether f = O(g). If so, tick them, otherwise, cross it and give specific reasons.

1. f(n) = n! , g(n) = 2^n ✗

   Explanation: f(n) is dominated by 2n, which is a exponential function while g(n) is dominated by n!, which is a factorial function. In this case, no matter how big the constant c is, eventually f(n) would be smaller than g(n) when x > x0 somewhere.

2. f(n) = 2 * 3^n , g(n) = 4^n ✓

Explanation : Note that what is mentioned before, for exponential function, $(c+1)^n$ dominates c^n. Therefore, $2 * 3^n$ can never grow faster than 4^n.

3. $f(n) = n^2 * 2^n$, $g(n) = 5^n$ ✓

Explanation: In this case, even though 2^n is multiplied by n^2, the whole f(n) is still dominated by 2^n. And 2n is inferior to 5^n, therefore f(n)=O(g(n)).

1.2 Big Omega (Ω)

Similar to the Big O notation, the Big Omega notation was originally created for mathematical use but then widely applied in computer science. In some sense, the Big Omega notation is the lower bound in Big O notation. If f = Ω(g), it means that the function f is superior to g under certain conditions.

Mathematically, there’s not much of a difference between the Big O notation and Big Omega notation. In computer science, the Big Omega notation is not as widely used as the Big O notation is. Why? probably due to the difference in the range of these two symbols. Big O notation can limit the range of complexity to “<= a certain number” while Big Omega notation could only limit the range of complexity to “>= a certain number” which would seem a little vague and inexact.

Mathematical Definition:

f(x) = Ω (g(x)) , if there exists a positive number c and a real number x0 such that |f(x)| >= c g(x) for all x >= x0;

Exercise: In each of the following pairs, figure out whether f = (g). If so, tick them, otherwise, cross it and give specific reasons.

1. $f(n) = n^{2.5}$ $g(n) = n^{1.2}$ ✓

   Explanation: In that f(n) and g(n) are both power function and the power of f(n) is larger than that of g(n), f(n) dominates g(n), indicating f(n) = Ω g(n).

2. $f(n) = 1.5 \log 2n$ $g(n) = 5 \log n^3$ ✗

   Explanation: g(n) can be simplified as 15 log n, g(n) = O(n log n). f(n) also equals O(n log n). In this case, it is undeniable that f(n) = Ω(g(n)). However, g(n) also equals Ω(f(n)). To represent this “two-way Ω”, we use a new notation Θ for this situation. And Θ will be discussed in the following section.

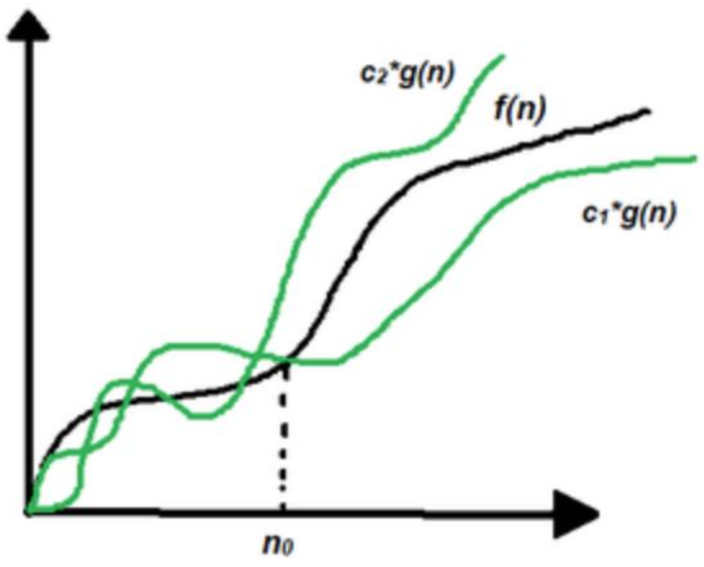
3. $f(n) = 2^n / n^2$, $g(n) = n^2 * (\log n)^2$ ✓

   Explanation: In f(n) , 2^n is divided by n^2 , but f(n) is still dominated by 2^n. In g(n), n^2 is multiplied by $(\log n)^2$, similarly, g(n) is dominated by n^2. Therefore, f(n) = Ω(g(n)).

1.3 Big Theta (ϴ)

In contrast to Big O and Big Omega notations, Big Theta notation is more like the “parents” of the two, because f(n) = ϴ(g(n)) if and only if f(n) = O(g) and f(n) = Ω(g). This may sound confusing, but the graph below shows how it actually works.

Mathematical Definition:
f(x) = ϴ (g(x)) , if there exists a positive number c and a real number x_0 such that |f(x)| >= c1 g(x) for all x >= x_0 and at the same time |f(x)| <= c2 g(x) for all x >= x_0.



Giving f(n) = Θ (g(n)), it means that the upper bound of f(n) is c2 * g(n), and the lower bound of f(n) is c1 * g(n). When we are using the Θ notation, it is fair to say that we are having an asymptotically tight bound of our algorithm — the running time of our algorithm fluctuate within the asymptotically tight bound.

The introduction of Big Theta, Big O, and Big Omega notation have one thing in common: we don't need to give a very exact running time of a certain algorithm by using these notations. For example, let's say the running time of a certain algorithm is $100n^2 + 20n +600$. it looks very long and redundant, right? But if we use Θ or O notations, we can narrow down the running time to just $\Theta(n^2)$. In this way, the running time of the algorithm looks more simple and easy to understand.

Exercise: In each of the following pairs, figure out whether f = (g). If so, tick them, otherwise, cross it and give specific reasons.

1. $f(n) = 3^n$ $g(n) = 2 * 3^{(n+1)}$ ✓

   Explanation: When c1 is a very small number(larger than 0, smaller than 1), f(n) = Ω g(n). When c2 >=1 , f(n) always grows slower than g(n) and therefore f(n) = O g(n), and f(n) = Θ g(n).

2. f(n) = 3n g(n) = 4n ✓

Explanation: When c1 is a relatively small number(larger than 0, smaller than 1), f(n) = Ω g(n). When c2 >=1 , f(n) always grows slower than g(n) and therefore f(n) = O g(n), and f(n) = ϴ g(n).

2. Divide and Conquer

2.1 technique definition

In computer science, we prefer to use the divide-and-conquer strategy to solve problems with complicated steps. Theoretically, the divide and conquer strategy contains mainly 3 steps:

1. Break up the complicated problem into small pieces of sub-problems.
2. Solve sub-problems individually
3. Combine the answers to the sub-problems and create an answer for the original problem.

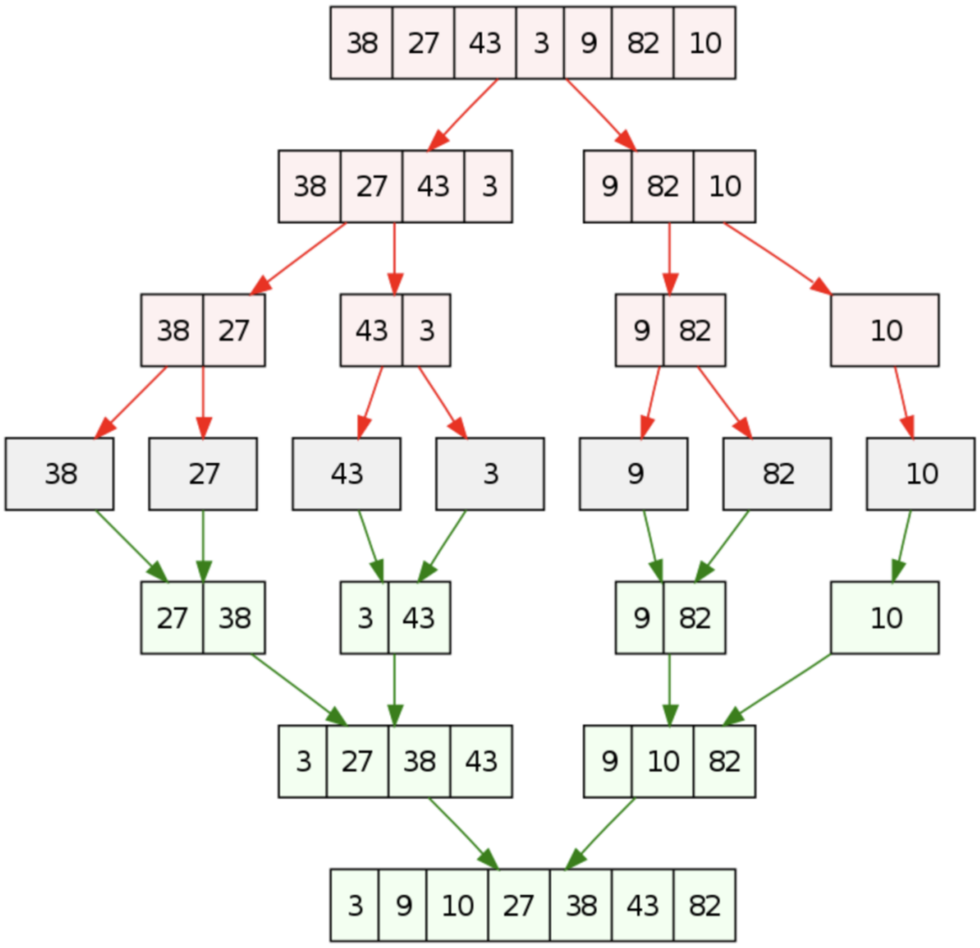
The core idea of the divide and conquer strategy is partitioning and combining, that is, the original problem is divided into n smaller sub-problems, and the structure of the original problem is similar, recursively solve these sub-problems, and then merge the results to get the solution of the original problem.

It looks a bit like recursion, but it is important to understand that divide and conquer strategy is a way of thinking about the problem, and recursion is a programming technique. It looks like it is because partitioning is generally better implemented with recursion.

Conditions of application of divide and conquer strategy：

1. The original problem and the subproblem have the same pattern between them
2. The divided sub-problems can be solved independently, and there is no correlation among the sub-problems.
3. The dividing process has a termination condition, i.e. when the problem is small enough to be solved directly.
4. The sub-problems can be merged into the original problem, but the complexity of the merging operation should not be too high, otherwise, the effect of reducing the overall complexity of the algorithm will not be achieved.

The merge sort can be the most classical implementation of the divide and conquer method: for an unsorted sequence with n elements, it is continuously separated into two subsequences from the middle, eventually forming n subsequences with only 1 element each. The subsequence with only 1 element is naturally ordered, and then merge two by two, sort the subsequences in the merging process, and finally return a merged sequence of n elements that are ordered. The process is shown in the figure below.



2.2 divide and conquer multiplication

The great mathematician Carl Friedrich Gauss devised a new method to calculate two complex numbers. Originally, we calculate complex numbers in the way of the formula below:

$$(a + bi)(c + di) = a*c - b * d + (b * c + a * d) i$$

Since bc + ad = (a + b) (c + d) - ac - bd, Gauss realized that the number of multiplications in the original equation can be reduced to 3. Although only one multiplication is reduced, the efficiency of many algorithms can be boosted due to recursion.

After knowing about this particular method, let’s apply this strategy to a practical problem.

Let’s say we want to calculate 2 n-bit integers. In order to perform the multiplication more easily, we divide each number into 2 parts, shown in the graph below.

$$x = \boxed{\quad x_L \quad}\boxed{\quad x_R \quad} = 2^{n/2}x_L + x_R$$
$$y = \boxed{\quad y_L \quad}\boxed{\quad y_R \quad} = 2^{n/2}y_L + y_R.$$

With this special strategy, the multiplication of x and y can be written as:

$$xy = (2^{n/2}\ x_L + x_R)(2^{n/2}\ y_L + y_R) = 2^n x_L\ y_L + 2^{n/2}\ (x_L\ y_R + x_R\ y_L) + x_R y_R.$$

If we do the multiplication in the traditional way, it may take 4 multiplications. But don’t forget we mentioned Gauss’s method. By applying Gauss’s method, we can reduce the multiplications to 3.

$$x_L\ y_R + x_R\ y_L = (x_L + x_R)(y_L + y_R) - x_L\ y_L - x_R y_R$$

And finally, we can apply the divide and conquer strategy to multiplication.

```
function multiply($x, y$)
Input:  Positive integers $x$ and $y$, in binary
Output:  Their product

$n =$ max(size of $x$, size of $y$)
if $n = 1$:  return $xy$

$x_L$, $x_R =$ leftmost $\lceil n/2 \rceil$, rightmost $\lfloor n/2 \rfloor$ bits of $x$
$y_L$, $y_R =$ leftmost $\lceil n/2 \rceil$, rightmost $\lfloor n/2 \rfloor$ bits of $y$

$P_1 =$ multiply$(x_L, y_L)$
$P_2 =$ multiply$(x_R, y_R)$
$P_3 =$ multiply$(x_L + x_R, y_L + y_R)$
return $P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2$
```

We should note that we are flooring the value of n/2 in the return statement.

Why does this one reflect the idea of divide and conquer? First, we divide the 2 inputting numbers into 4 parts and recursively apply them to the multiply function. After we recursively apply the multiply function and finally reach the base —--- that is:

> when n = 1,
> the function directly returns xy and returns each value repeatedly.

Exercise: Given 2 numbers X and Y below, calculate X*Y using the divide and conquer multiplication algorithm.

X = 01100101
Y = 01001000

Solution:

First, we are supposed to split each number into the left and right parts:

XL = 0110

XR = 0101

YL = 0100

YR = 1000

Then we can have:

P1 = XL * YL (0110 * 0100)

P2 = XR * YR (0101 * 1000)

P3 = (XL + XR) * (YL + YR) = 1011 * 1100

Since we don’t know the result of P1, P2, P3, we have to jump into next recursion.

Next recursion:

P1 = XL * YL = (0110 * 0100)

Split the XL and YL into corresponding parts:

XLL= 01

XLR = 10

YLL = 01

YLR = 00

Then we can have:

P1 = XLL * YLL (01 * 01)

P2= XLR * YLR(10 * 00)

P3 = (XLL+ XLR) * (YLL + YLR) = (11 * 01)

P2 = XR * YR(0101 * 1000)

Split them into parts:

XRL = 01

XRR = 01

YRL = 10

YRR = 00

Then we have:

P1 = XRL * YRL(01 *10)

P2 = XRR * YRR(01 * 00)

P3 = (XRL + XRR) * (YRL + YRR) = (10 * 10)

P3 = (XL + XR) * (YL + YR) = 1011 * 1100

XL = 10

XR = 11

YL = 11

YR = 00

Then we have:

P1 = XL * YL (10 * 11)

P2 = XR * YR (11 * 00)

P3 = (XL + XR) * (YL + YR) = (101 * 011)

Next, we need to calculate **XLL * YLL(01 * 01) in P1**

XLLL = 0

XLLR = 1

YLLL = 0

YLLR = 1

Then we have:

P1 = XLLL * YLLL = 0

P2 = XLLR * YLLR = 1 * 1 = 1

P3 = (XLLL + XLLR) * (YLLL + YLLR) = 1 * 1 = 1

XLL * YLL = P1 * 2 ^n + (P3 - P2 - P1) * 2 + 1 = 0 + 0 + 1 = 1

XLR * YLR(10 * 00)

XLRL = 1

XLRR = 0

YLRL = 0

YLRR = 0

Then we have:

P1 = XLRL * YLRL = 1 * 0 = 0

P2 = XLRR * YLRR = 0 * 0 = 0

P3 = (XLRL + XLRR) * (YLRL + YLRR) = 1 * 0 = 0

(XLL+ XLR) * (YLL + YLR) = (11 * 01)

Split:

1L = 1

1R = 1

2L = 0

2R = 1

Then we have:

P1 = 1L * 2L = 0

P2 = 1R * 2 R = 1

P3 = (1L + 1R) * (2L + 2R) = 10

11 * 01 = 0 + 1 * 2 + 1 = 11

XRL * YRL(01 *10)

Split:

XRLL = 0

XRLR = 1

YRLL = 1

YRLR = 0

Then we have:

P1 = XRLL * YRLL = 0

P2 = XRLR * YRLR = 0

P3 = (XRLL + XRLR) * (YRLL + YRLR) = (0 + 1) * (1 + 0) = 1

XRR * YRR(01 * 00)

Split:

XRRL = 0

XRRR = 1

YRRL = 0

YRRR = 0

Then we have:

P1 = XRRL * YRRL = 0 * 0 = 0

P2 = XRRR * YRRR = 1 * 0 = 0

P3 = (XRRL + XRRR) * (YRRL + YRRR) = 1 * 0 = 0

(XRL + XRR) * (YRL + YRR) = (10 * 10)

Split:

1L = 1

1R = 0

2L = 1

2R = 0

Then we have :

P1 = 1L * 2L = 1

P2 = 1R * 2R = 0

P3 = (1L + 1R) * (2L + 2R) = 1

10 * 10 = 1 * 2^2 + 0 + 0 = 100

XL * YL (10 * 11)

Split:

XLL = 1

XLR = 0

YLL = 1

YLR = 1

Then we have:

P1 = XLL * YLL = 1

P2 = XLR * YLR = 0

P3 = (XLL + XLR) * (YLL + YLR) = 01 * 10 = 10

XL * YL = 1 * 2 ^2   + 1 * 2 + 0 = 110

XR * YR (11 * 00)

XRL = 1

XRR = 1

YRL = 0

YRR = 0

Then we have:

P1 = XRL * YRL = 0

P2 = XRR * YRR = 0

P3 = (XRL + YRL) * (YRL + YRR) = 0

XR * YR = 0 + 0 + 0 = 0

(XL + XR) * (YL + YR) = (101 * 011)

split :

1L = 10

1R = 1

2L = 1

2R = 1

P1 = 1L * 2L = 10

P2 = 1R * 2R = 1

P3 = (1L + 1R) * (2L + 2R) = 11 * 10

11 * 10

1L = 1

1R = 1

2L = 1

2R = 0

P1 = 1L * 2L = 1

P2 = 1R * 2R = 0

P3 = (1L + 1R) * (2L + 2R) = 10 * 1 = 10

11 * 10 return

P1 * 2 ^n + (P3 - P1 -P2) * 2^floor(n/2) + P2 = 1*4 + 1 * 2 + 0= 110

Therefore, the value of 11 * 10 = 110 and we go back to the previous iteration

P1 = 1L * 2L = 10

P2 = 1R * 2R = 1

P3 = (1L + 1R) * (2L + 2R) = 11 * 10 = 110

return 1111 , and (101 * 011) = 1111

Then we can go back to the previous recursion

XLL * YLL = P1 * 2 ^n + (P3 - P2 - P1) * 2 + 1 = 0 + 0 + 1 = 1

XLR * YLR = 0 + 0 + 0 = 0

11 * 01 = 0 + 1 * 2 + 1 = 11

XRL * YRL = 0 + 1 * 2 + 0 = 10

XRR * YRR = 0 + 0 + 0 = 0

10 * 10 = 1 * 2^2 + 0 + 0 = 100

XL * YL = 1 * 2 ^2 + 1 * 2 + 0 = 110

XR * YR = 0 + 0 + 0 = 0

return P1 * 2^n + (P3-P2-P1) * 2floor$^{n/2}$ + P2

P1 = XL * YL = (0110 * 0100) = 1 * 16 +(11 - 0 - 1) * 4 + 0 = 111000

P2 = XR * YR(0101 * 1000) = 10 * 2^4 + (100 - 0 - 10) * 2^2 + 0 = 101000

P3 = (XL + XR) * (YL + YR) = 1011 * 1100 = 110 * 2^4 + (1111 - 0 - 110) * 2^2 + 0 = 10000100

X * Y = 111000 * 2^8 + (10000100 - 101000 - 111000) * 2^4 + 101000 = 1110000010100

2.2 divide and conquer algorithmic design

The basic steps for the design of a divide and conquer algorithm:

step1: Dividing
Divide the original problem into several smaller, independent sub problems of the same form as the original problem.

step2: Solve
if the sub-problem is small and easy to be solved then solve it directly, otherwise solve each sub-problem recursively

step3: Merge
Combine the solutions of each sub-problem and generate the solution of the original problem.

In computer science, many algorithms are utilizing the methodology of the divide and conquer algorithmic design pattern, such as merge sort, quick sort…

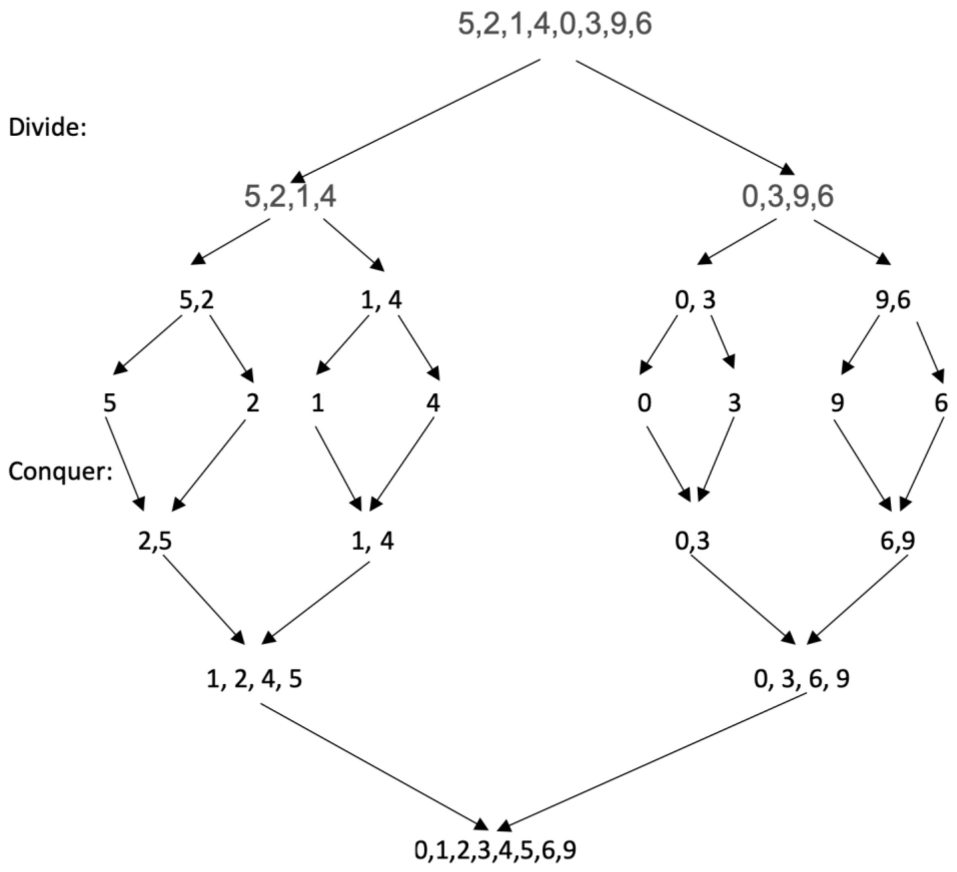
Its general algorithmic design pattern is as follows.

Divide and conquer(P)

1. if P <= a certain number
2. then return a certain value
3. Divide P into smaller subproblems P1 ,P2 ,... ,Pk
4. for i = 1 to k
5. do yi ← Divide-and-Conquer(Pi)
6. T ← MERGE(y1,y2,...,yk) △
7. return(T)

Exercise:
Have a walkthrough of the merge sort process.
The original array: 5,2,1,4,0,3,9,6

Solution:



the time complexity of a merge sort: O(n log n)
the space complexity of a merge sort: O(n)

3. Recurrences and Master Theorem

3.1 Recurrences

Recurrences are generally applied to the divide and conquer algorithm. During the dividing process, recurrences are repeatedly applied to each recursion. And also during the merging process, recurrences are applied to merge all the sub-problems to the original problem. And generally, we tend to write the pattern of a recurrence in the following way:

$$T(n) = kT(n/b) + O(n)$$

In practical application, the recurrence relation can be used for calculating the running time complexity of a divide and conquer or an algorithm with recursive calls.

Exercise:
Given a recurrence: $T(n) = 4T(n/2) + O(n)$, what is the kth general term in this case? And what value of k should be plugged in to get the answer?

Solution:

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

Think of $O(n)$ as being $<= cn$ for some constant c

Then we can have:

$T(n) <= 4T(n/2) + cn$

$\quad <= 4[\ 4T(n/4) + cn/2] + cn = 16T(n/4) + 3cn$

$\quad <= 16[4T(n/8) + cn/4] + 3cn = 64T(n/8) + 7cn$

$\quad <= 64[4T(n/16) + cn/8] + 7cn = 256T(n/16) + 15cn$

In general, T(n) can be written as:

$T(n) <= 4^kT(n/2^k) + 2^k - 1$

Then if we substitute $k = \log_2 n$, we can have:

$T(n) <= 4^{\log_2 n}T(1) + 2^{\log_2 n} - 1$

$\quad <= n^2T(1) + n - 1$

Since T(1) = O(1)

Then we can have:

$T(n) <= n^2 + n - 1$

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

3.2 Master Theorem

What is the master theorem?

In algorithmic analysis, the master theorem provides a way to represent many recursive relations obtained by divide and conquer with asymptotic notations.

What does the master theorem do?

in short, it is a method for computing the time complexity of recursions.

For example, if the recurrence relation is as follows:

$T(n) = aT(n/b) + O(n^d)$
where n is the size of the problem, a is the number of recursive problems, n/b is the size of the sub-problem.

When $a < b^d$, $\log_b^a < d$, $T(n) = O(n^d)$
When $a > b^d$, $\log_b^a > d$, $T(n) = O(n^{\log b\ a})$
When $a = b^d$, $\log_b^a = d$, $T(n) = O(n^{\log b\ a} * \log n)$

Exercise: Using the master theorem to solve the following recurrence relations

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

Solution:

$a = 2$, $b = 2$, $d = 1$

Since $a = b^d$, according to the master theorem:

$T(n) = O(n \log n)$

(b) $T(n) = 3 T(n/2) + 1$

Solution:

$a = 3$, $b = 2$, $d = 0$

Since $a > b^d$, according to the master theorem:

$T(n) = O(n \log_b{}^a) = O(n \log_2{}^3)$

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

Solution:

a = 4, b = 3, d = 1

Since a > b ^{d}, according to the master theorem:

T(n) = O(n log $_{b}$ ^{a}) = O(n log$_{3}$^{4})

(d) T(n) = 2T(n / 5) + n^{2}

Solution:

a = 2, b = 5, d = 2

Since a < b^{d}, according to the master theorem:

T(n) = O(n^{d}) = n^{2}

(e) T(n) = 4T(n/2) + n^{3}

Solution:

a = 4, b = 2, d = 3

Since a < b^{d}, according to the master theorem:

T(n) = O(n^{d}) = n^{3}

3.3 Using Master Theorem to analyze a recursive algorithm

When we want to analyze the running time complexity of a recurrence relation, let's say T(n) = a(n/b) + n^{d} , this is more like a standard pattern of the master theorem. And in this case, applying the master theorem to the recurrence relation is good. However, what if the pattern becomes T(n) = T(n-1) + n^{d}? In this case, the pattern of T(n) doesn't fit in the standard master theorem. To analyze its time complexity, we have to derive a more general term for the T(n). and substitute n with k, then we can obtain the time complexity of the recurrence relation. But for this section, we will not discuss the methodology of deriving a more general term for the T(n), what we discuss in this section is all standard patterns of the master theorem.

If we want to apply the master theorem, the form of T(n) must be strictly identical to the standard format of the master theorem. applying the master theorem to compute the time complexity of a relation makes the whole process much easier compared with other approaches because all you have to do is get the a, b, and d from the recurrence relation. To analyze a recursive algorithm using the master theorem, we have to figure out the number of each sub-problem after each division(a), the size of each sub-problem(n/b), and the time required to process each sub-problem(n^d).

Exercise: Analyze the time complexity of the merge sort using the master theorem.

First, in order to apply the master theorem to a merge sort, we have to derive the recurrence relation of the merge sort.

From the previous sections, we know that merge sort is the application of divide and conquer algorithm. And we know that the original array is divided into 2 sub-arrays during each division process. And the size of each sub-array is actually half of the original array. Therefore we can derive that

$$T(n) = 2\ T(n/2) + n^d.$$

Next, we need to obtain the d in order to apply the master theorem. In the merge sort process, n^d corresponds to the merging process after the division process. In the worst case, we need to compare each element in the 2 sub-arrays and put the sorted elements in the merged array. Therefore we can derive that the time complexity for each merging process is O(n) (n is the size of the sorted merging array).

Finally, we can write the recurrence relation of a merge sort as :
$T(n) = 2\ T(n/2) + n$

And we can have $a = 2$, $b = 2$, $d = 1$
and $a = b^{d}$.
Then we can derive the time complexity of the merge sort as : $O(n \log_b^a \log n) = O(n \log n)$