Video Notes

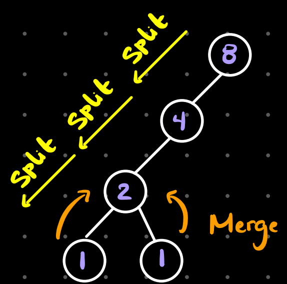
- Idea behind Merge Sort
  - Split array until we have subarrays of size one (base case)
  - After, merge these subarrays back together until you have your final sorted array

- Made up of 2 functions:

  ① Split
  ② Merge

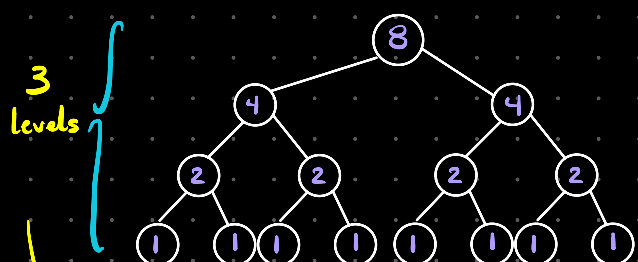
Split Subroutine

Suppose n = 8 (size of array)



This is how it recursively does the splitting, doesn't go level by level

Complete Tree

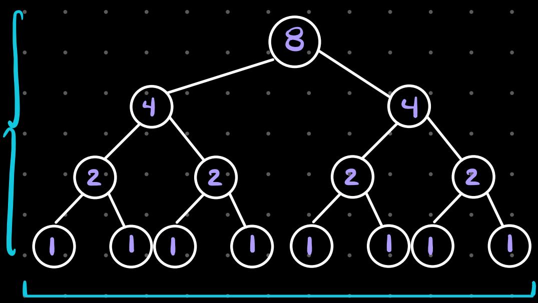


Pay close attention to the number of splits that ocurred, in relation to our input n

How do I turn this to a function of n ?

- Each step, we split the array in $\boxed{\text{half}}$

$$\log_2(8) \rightarrow \log_2(n)$$



log(n)
(# of levels we can split down)

8 leaves = n # of leaves

$2^x = 8$

$2^3 = 8$ → # of levels

Here, we start to see ideas that lead to Merge Sort being:

$O(n * \log(n))$

Let's look at mergeSort() code implementation

```
//this function will continue to split the arrays into halves until we try to perform
//mergeSort on a subarray of size 1 (left == right)
void mergeSort(int arr[], int begin, int end)
{
    if(begin < end){
        //midpoint is where we split the array into two subarrays
        int midpoint = begin + (end - begin) / 2;

        mergeSort(arr, begin, midpoint);
        mergeSort(arr, midpoint + 1, end);

        //merge sorted subarrays
        merge(arr, begin, midpoint, end);
    }
}
```

→ split()

→ merge()

① Idea is that mergeSort() will recursively call itself on each left and right subarray until we're reached the base case

② Merging occurs after we have a sorted left and right subarray

---

Merge Subroutine

- It's important that we understand how the merge function works as well, we'll go over a worst case scenario when merging two sorted subarrays, as it will be important for the Recurrence Relation as well

List 1 | 1 | 2 | 3 |

List 2 | -1 | 0 | 6 |

n = length of both lists combined = 6

Total Comparisons : 1

We compare 1 and -1, lesser value wins and gets inserted to new array (comparisons increments by 1)

-1 __ __ __ __ __

List 1 | 1 | 2 | 3 |

List 2 | -1 | 0 | 6 |

n = length of both lists combined = 6

Total Comparisons : 2

-1 0 __ __ __ __

List 1 | 1 | 2 | 3 |

List 2 | -1 | 0 | 6 |

n = length of both lists combined = 6

Total Comparisons : 3

-1 0 1 __ __ __

List 1 | 1 | 2 | 3 |

List 2 | -1 | 0 | 6 |

n = length of both lists combined = 6

Total Comparisons : 4

-1 0 1 2 __ __

List 1 | 1 | 2 | 3 |

List 2 | -1 | 0 | 6 |

n = length of both lists combined = 6

Total Comparisons : 5

-1 0 1 2 3 __

↓

List 1 | 1 | 2 | 3 |

List 2 | -1 | 0 | 6 |

n = length of both lists combined = 6

Total Comparisons : 5

We've exhausted all elements from "List 1", so we don't compare 6 to anything, just insert to array

-1 0 1 2 3 6

↓

List 1 | 1 | 2 | 3 |

List 2 | -1 | 0 | 6 |

n = length of both lists combined = 6

Total Comparisons : 5

END!!!

5 comparisons... n = 6

(n - 1) comparisons

-1 0 1 2 3 6

We can declare this!

In the worst case, if we traverse through both lists exhaustively, where n is the combined Length of both lists, will do:

(n-1) comparisons

Recurrence Relation

- Recursively define what the answer to the subproblem is

$n = 8$

8 → 4, 4

Cut in half, $n/2$

$$T(n) = \underbrace{T(n/2)}_{\text{Left split}} + \underbrace{T(n/2)}_{\text{right split}} + \underbrace{(n-1)}_{\text{merging (worst case comparisons)}}$$

$T(1) = 0$ Base Case (if we have one element in subarray, no comparisons done)

Simplify!

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

Example:

$T(8) = 2T(8/2) + (8-1)$

$= 2T(4) + 7$ What is $T(4)$? Recursively call it!

$T(4)$: $2T(4/2) + (4-1)$

$= 2T(2) + 3$ Solve for $T(2)$

$T(2)$: $2T(2/2) + (2-1)$

$= 2T(1) + (1)$ $T(1) = 0$ Base Case!

$= 2(0) + 1$ We can now work our way back up

$= 1$ $\longrightarrow$ If $n = 2$, we do 1 comparison

$T(4) = 2T(2) + 3$

$= 2(1) + 3$

$T(4) = 5$ $\longrightarrow$ $T(4) = 5$ comparisons in worst case

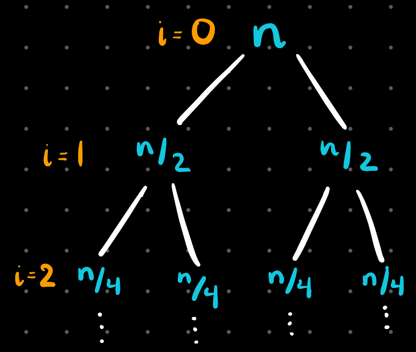
$T(8) = 2T(4) + 7$

$= 2(5) + 7$

$= 17$ $\longrightarrow$ $T(8) = 17$ comparisons in worst case

We will later solidify this into a solid function of n instead of a recursive relationship

Investigation of Work Done at Every Level



$1(n-1)$ — 1 subproblem, $(n-1)$ comparisons

$2(n/2-1)$ — 2 subproblems, n is now split in half

$4(n/4-1)$

Find a pattern!

i = level

$1(n-1) \rightarrow 2^0(n/2^0-1)$

$2(n/2-1) \rightarrow 2^1(n/2^1-1)$

$4(n/4-1) \rightarrow 2^2(n/2^2-1)$

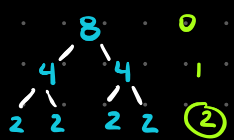
$2^i(n/2^i-1)$ Generic Equation for amount of work at the i^{th} level

Find worst case work to convert Recurrence Relation to a solid mathematical formula

$\rightarrow 2^i(n/2^i-1)$

If I wanted to find work done through all levels, we just take the sum

Summation Building



last level excluded since they contribute 0 comparisons (base case)

$\log(8) = 3$ (# of splits)

$3-1 = 2$ (up to 2 levels, in terms of indexing off zero)

Figuring out max. # of levels that we'll have (upper bound)

$\log(n)-1$

$$\sum_{i=0}^{\log(n)-1} 2^i\left(\frac{n}{2^i}-1\right) \rightarrow \sum\left(\frac{\cancel{2^i}(n)}{\cancel{2^i}} - 2^i\right) \rightarrow \sum n - 2^i \rightarrow \sum n - \sum 2^i$$

$$\sum_{i=0}^{\log(n)-1} n - \sum_{i=0}^{\log(n)-1} 2^i$$

$$((\log(n)-1)-0)+1 \longrightarrow n((\log(n)-1)-0)+1$$

$$= n\log(n)$$

$$1+2+4+\cdots+2^{\log(n)-1}$$

$$= \frac{2^{((\log(n)-1)+1)}-1}{(2)-1}$$

$$= 2^{\log_2(n)}-1$$

$$= n-1$$

$n\log(n) - n + 1$ comparisons (worst case, exact computation)

For Big(O), we only care about leading factors

Merge Sort : $O(n\log(n))$