Lecture 6

Recap:

i) Derived and solved Recurrence Relation from Binary Search Algorithm
ii) Analysed Worst Case Complexity of Binary Search Algorithm

Average comparisons lower bound

To get a lower bound, simply omit all nodes on the last level. So we compute comparisons for a total of L = flr(log(n)) levels. Using the same formula, we get

total comparisons

$\geq$ Comparisons(flr(log(n)))

$= (\text{flr}(\log(n)) - 1) * 2^{\text{flr}(\log n)} + 1$

The flr(log(n)) expression truncates the decimal part of log(n) and so it will subtract away less than 1 from log(n), i.e.,
flr(log(n)) $\geq$ log(n)–1

Replacing flr(log n) by log(n)–1 and doing the algebra, we get
total comparisons

$\geq (\log(n) - 2) * 2^{\log(n) - 1} + 1$

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

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

Dividing this total by n gives the average, i.e., average comparisons

$\geq \frac{1}{2} * \log(n) - 1 + 1/n$

$> \frac{1}{2} * \log(n) - 1$

$> \frac{1}{4} * \log(n)$ (for sufficiently large n)

From this we can say:

The average number of comparisons is $\Omega(\log(n))$.

We also know that average number must be less that the worst, which we know is worst case comparisons $\leq \log(n) + 1$

Therefore, the average number of comparisons for any n is somewhere between: $\frac{1}{2}*\log(n) - 1$ and $\mathbf{1}*\log(n) + 1$

Qn 2

```
MergeSort(A,l,r)
if l < r then
 m := (⌈(l+r)/2⌉ )
MergeSort(A,l,m)
MergeSort(A,m+1,r)
Merge(A,l,m,r)
```

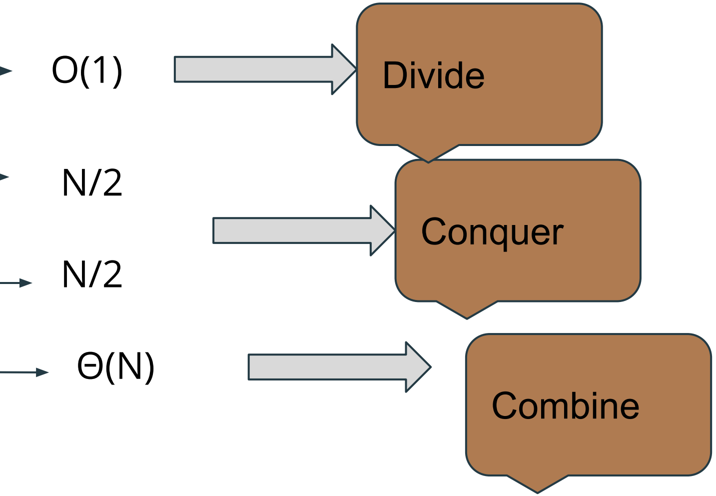
Find the recurrence relation for the pseudocode.

```
MergeSort(A,l,r)
if l < r then
 m := (⌈(l+r)/2⌉ )        → O(1)
MergeSort(A,l,m)          → N/2
MergeSort(A,m+1,r)        → N/2
Merge(A,l,m,r)            → Θ(N)
```

Summing up time required to perform all steps:

$T(n)=2T(n/2)+ \Theta(N)$ $n>1$
and $T(n)= \Theta(1)$ if $n=1$

```
MergeSort(A,l,r)
if l < r then
 m := (⌈(l+r)/2⌉ )
MergeSort(A,l,m)
MergeSort(A,m+1,r)
Merge(A,l,m,r)
```



Summing up time required to perform all steps:

$T(n)=2T(n/2)+\Theta(n)$ $n>1$
and $T(n)=\Theta(1)$ if $n=1$

Solving by iteration:

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

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

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

$= 2^2 (2T(n/8) + n/4) + 2n$

$= 2^3T(n/8) + 3n$

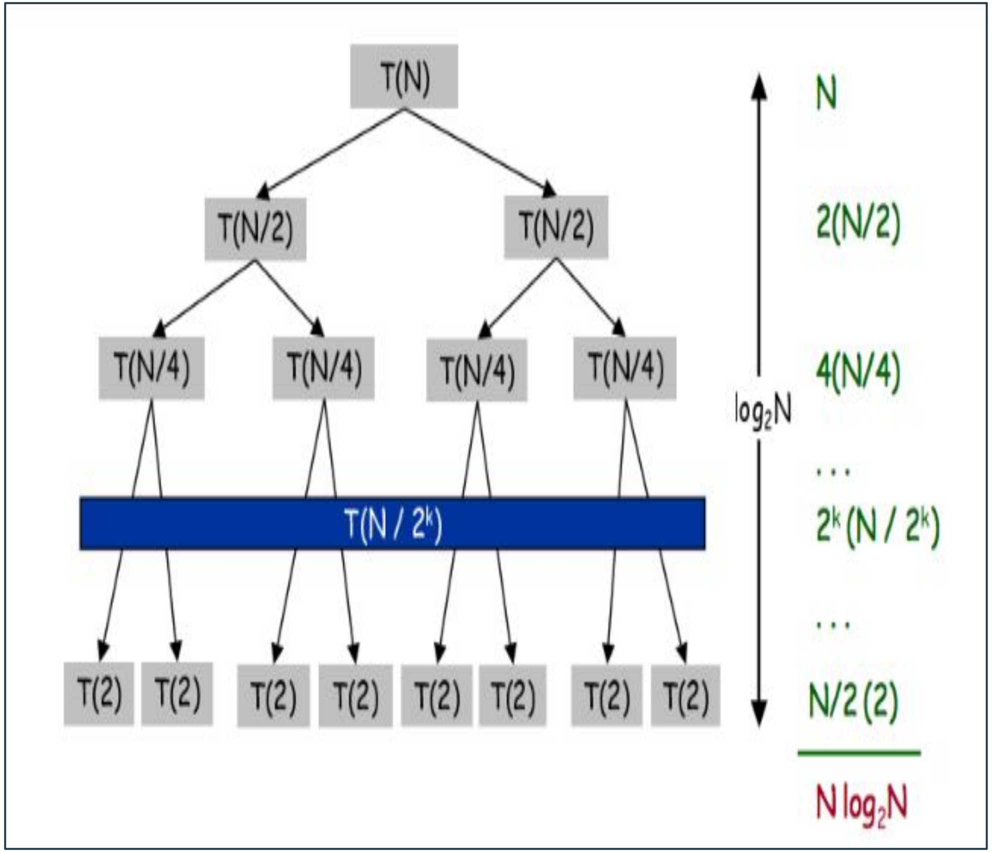
$T(n) = 2^kT(n/2^k) + k n$

let us assume that n is a power of 2, i.e $n = 2^k$ for some k.

$= 2^{\log n}T(n/n) + n \log n$

$= n + n \log n$

Recurrence Tree Method:



Analysis of Merge Sort

To justify that the running time of the merge-sort algorithm is O(n log n). Let the function t(n) denote the worst-case running time of merge-sort on an input sequence of size n.

Since merge-sort is recursive, we can characterize function t(n) by means of the following equalities, where function t(n) is recursively expressed in terms of itself, as follows:

$$t(n) = \begin{cases} b & \text{if } n = 1 \text{ or } n = 0 \\ t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + cn & \text{otherwise} \end{cases}$$

where b > 0 and c > 0 are constants.

In this case, we can simplify the definition of t(n) as follows:

$$t(n) = \begin{cases} b & \text{if } n = 1 \\ 2t(n/2) + cn & \text{otherwise.} \end{cases}$$

We must still try to characterize this recurrence equation in a closed- form way. One way to do this is to iteratively apply this equation, assuming n is relatively large. For example, after one more application of this equation, we can write a new recurrence for t(n) as follows:

$$\begin{aligned} t(n) &= 2\left(2t\left(n/2^2\right) + (cn/2)\right) + cn \\ &= 2^2 t\left(n/2^2\right) + 2cn. \end{aligned}$$

By Iteration we get the equation as:

$$t(n) \quad = \quad 2^i t\left(n/2^i\right) + icn.$$

To see when to stop, recall that we switch to the closed form t(n) = b when n = 1, which occurs when 2^i = n. In other words, this will occur when i = log n. Making this substitution yields :

$$\begin{aligned} t(n) &= 2^{\log n} t\left(n/2^{\log n}\right) + (\log n)cn \\ &= nt(1) + cn\log n \\ &= nb + cn\log n. \end{aligned}$$

We get an alternative justification of the fact that t(n) is O(n log n).