Algorithm Analysis

Definition 2.1
$T(N) = O(f(N))$ if there are positive *constants* c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

O(f(N)) indicates the growth rate of worst/upper bound on time taken as the given input size of N increases. f(N) here is the total cost of whole algorithm calculated for line by line execution and c * f(N) is a approximation of this cost that will always be a little worse or more upper bounded hence satisfying the overall use as a growth rate approximation for worst time only when the N is larger than n_0.

```
int sum(int n) {
    int partialSum;
    partialSum = 0;
    for (int i = 1; i <= n; ++i) {
        partialSum += i * i * i;
    }
    return partialSum;
}
```

Full Cost Breakdown

1. **Initialization of** `partialSum`:
    - **Cost:** 1 unit.
2. **Loop initialization (** `int i = 1;` **):**
    - **Cost:** 1 unit.
3. **Loop condition (** `i <= n` **):**
    - This is checked $N + 1$ times.
    - **Cost:** $N + 1$ units.
4. **Loop increment (** `++i` **):**
    - This increment happens N times.
    - **Cost:** N units.
5. **Loop body (** `partialSum += i * i * i;` **):**
    - Each iteration involves two multiplications, one addition, and one assignment.
    - **Cost per iteration:** 4 units.
    - **Total for** `N` **iterations:** $4N$ units.
6. **Return statement (** `return partialSum;` **):**
    - **Cost:** 1 unit.

The above function has a total worst time cost of O (6N + 4). 6N+4 will always be less than 7N for N larger than 4 giving c = 7 and n_0 = 4. c = 8 & n_0 = 2 also satisfies the above function but the tightest approximation is preferred for a good estimate of growth rate. There is a little-o notation, same as big O, but in that all c values must be satisfied. (6N -> o{N^2})

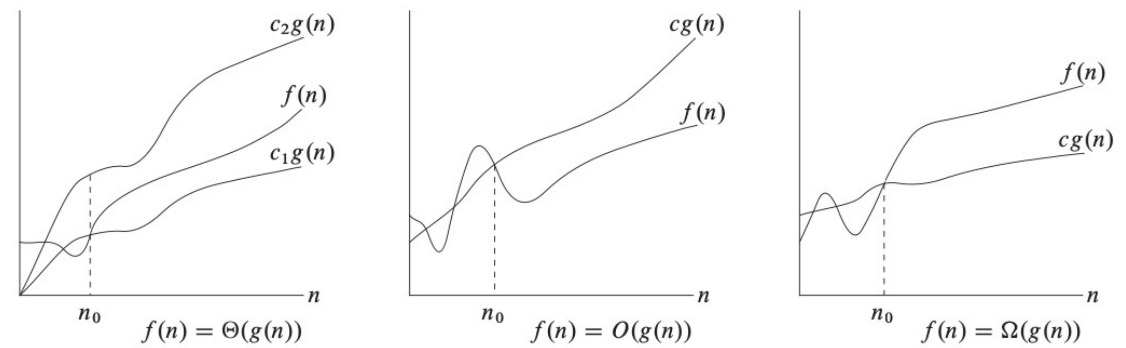
Ultimately insignificant terms and constants can be ignored as they are overwhelmed by parts in formula that will grow on a significantly faster speed, in this case whether our approximation is 7N or 8N, the final answer is O(N) as the 7 or 8 doesn't really matter in determining how fast the algorithm process time for worst case increase with increase in N.
Similarly O(N^2 + N) would just be O(N^2) due to N^2 growth being far more significant than N.

As for cost calculation, basic operations like assignment take constant O(1) cost and Loops usually take O(N^k) time based on k-nesting of loops present. If we are dividing the input size in k parts, the division itself takes O($\log_k N$) cost while whatever work is done at every level of division will be multiplied into O($\log_k N$). For example Merge Sort divides array in 2 parts endlessly until base case of one subpart is left for O($\log_2 N$) then sorts them at all levels for O(N) work, giving time complexity O($N\log_2 N$).

Definition 2.2

$T(N) = \Omega(g(N))$ if there are positive *constants* c and n_0 such that $T(N) \geq cg(N)$ when $N \geq n_0$.

Similarly we can approximate for Best Time case or lower bound time while The Average Time case will be between c1*g(N) and c2*g(N) for a certain n_0



$f(n) = \Theta(g(n))$ (a) $f(n) = O(g(n))$ (b) $f(n) = \Omega(g(n))$ (c)

Recurrence Tree & Master Theorem

Recurrence Tree

$$T(n) = 3T(n/4) + \Theta(n^2)$$

This problem means $T(n)$ is divided into subproblems of size $T(n/4)$ & 3 of these problems are further used

The division and merging of these 3 subproblems take cn^2 cost ($\Theta(n^2)$)

we will assumme n is some multiple of 4 for convininience

lets say $n = 4$

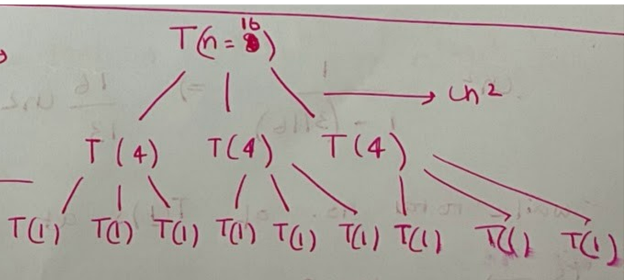
$T(n=4)$ $\quad cn^2$

$T(n/4 = 1)$ $\quad T(n/4 = 1)$ $\quad T(n/4 = 1)$

so total cost is cleary $3 \times T(1) + cn^2$

$\underbrace{3 \times T(1)}_{\text{Base case}} + \underbrace{cn^2}_{\text{merge}}$

for $n=16$ →



$c\left(\frac{n}{4}\right)^2 + c\left(\frac{n}{4}\right)^2 + c\left(\frac{n}{4}\right)^2$ ←

so total cost is $9 \times T(1) + cn^2 + \frac{3}{16} cn^2$

if we see the pattern

the latest divide / merge will take

$$\left(\frac{3}{16}\right)^{(\text{height}-1)} cn^2$$

where height for dividing by 4 uptil base case

→ $\log_4 n$

so total divide merge cost

$$cn^2 + \frac{3}{16} cn^2 + \frac{3}{16}^2 cn^2 \cdots \frac{3}{16}^{(\log_4 n - 1)} cn^2$$

$$cn^2 \left(\left(\frac{3}{16}\right)^0 + \left(\frac{3}{16}\right)^1 + \left(\frac{3}{16}\right)^2 \cdots \frac{3}{16}^{(\log_4 n - 1)}\right)$$

its okay to assume G.P will go to infinity

$$cn^2 \frac{1}{1-(3/16)} \Rightarrow \frac{16}{13} cn^2$$

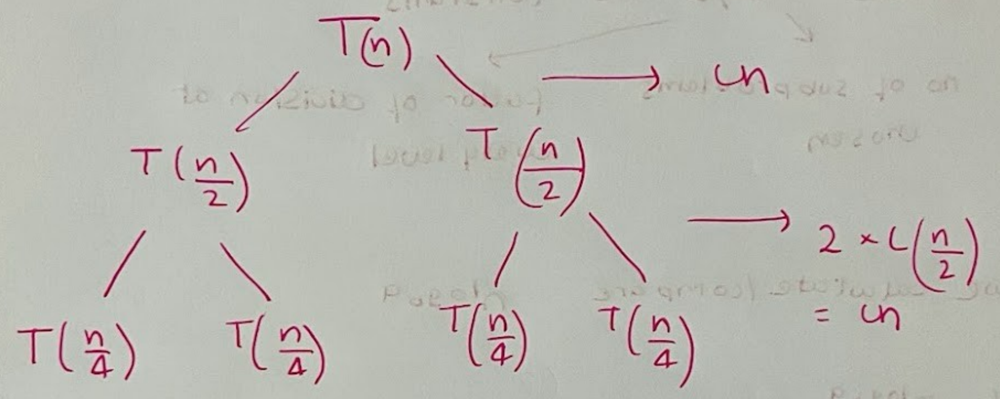
while total no. of $T(1)$s at last base case

is ⟶ $3^{\log_4 n}$ (via pattern & obvious)

Final answer = $\frac{16}{13} cn^2 + 3^{\log_4 n}$ or $\frac{16}{13} cn^2 + n^{\log_4 3}$

this is peculiar case where cost itself (for div/merge) changes every level we go down, as its a exponent of input size 'N'

if it was same for every level, when div/merge cost has exponent 'one' for 'N', like merge sort



so total div/merge cost

$cn + cn + cn \; .... \; (\log_2 n - 1 \text{ times})$

$cn \times (\log_2 n - 1)$

no of T(1)s at base level $\rightarrow$ $(2^{\log_2 n}$ or $n^{\log_2 2} = (n)$

The answer is roughly

$O(n \log_2 n) + O(n)$

merge/divide

→ highly insignificant to $(n \log n)$

Master Theorem

for recurrence relations of form

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

where a, b are constants

no of subproblems chosen

factor of division at every level

we calculate / compare $n^{\log_b a}$

if $n^{\log_b a}$ grows faster than $f(n)$

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

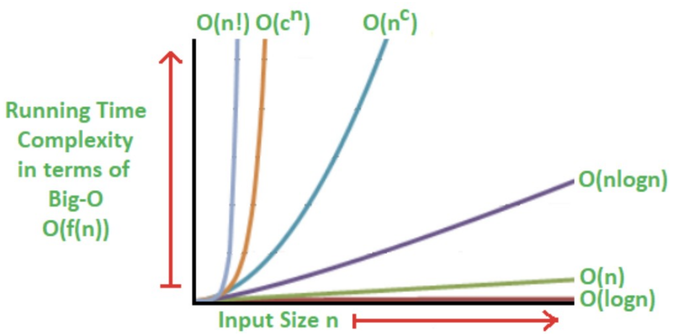
if $f(n)$ grows faster than $n^{\log_b a}$

$$T(n) = \Theta(f(n))$$

if equal growth rate

$$T(n) = \Theta\left(n^{\log_b a} \log n\right)$$

growth rates of some functions for order of significance in worst time calculation



Sorting Methods

Insertion Sort O(n^2) for nearly sorted arrays or small datasets

```
void insertionSort(std::vector<int>& arr, int size){
    for(int i=1; i<size; i++){
        int key = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > key){
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = key;
    }
}
```

Selection Sort O(n^2) for small arrays when memory is limited

```
using namespace std;
void selectionSort(vector<int>& arr,int size){
    for(int i=0;i<size-1;i++){
        int minval = i;
        for(int j=i+1;j<size;j++){
            if(arr[j]<arr[minval]){
            minval =j;
            }
        }

        if(minval!=i){
            arr[i]=arr[i]+arr[minval];
            arr[minval]=arr[i]-arr[minval];
            arr[i]=arr[i]-arr[minval];
        }
    }
}
```

Bubble Sort O(n^2) for teaching purpose only

```
using namespace std;
void bubbleSort(vector<int>& v) {
    int n = v.size();

    for (int i = 0; i < n - 1; i++) {

        for (int j = 0; j < n - i - 1; j++) {

            if (v[j] > v[j + 1])

                swap(v[j], v[j + 1]);
        }
    }
}
```

Merge Sort O(n logn) for large data sets

```cpp
#include <iostream>
#include <vector>

void merge(std::vector<int>& arr, int left, int mid, int right){
    int n1 = mid - left + 1;
    int n2 = right - mid;

    std::vector<int> L(n1), R(n2);

    for(int i=0; i<n1; i++){
        L[i]=arr[left+i];
    }
    for(int i=0; i<n2; i++){
        R[i]=arr[mid+i+1];
    }

    int i=0,j=0,k=left;

    while(i<n1 && j<n2){
        if(L[i]<=R[j]){arr[k]=L[i];i++;}
        else{arr[k]=R[j];j++;}
        k++;
    }

    while(i<n1){arr[k]=L[i];i++;k++;}
    while(j<n2){arr[k]=R[j];j++;k++;}

}

void mergeSort(std::vector<int>& arr, int left, int right){
    if (left >= right) return;
    int mid = left + (right-left)/2;


    mergeSort(arr, left, mid);
    mergeSort(arr, mid+1, right);
    merge(arr, left, mid, right);
}
```

Quick Sort $O(n^2)$ but on avg gives O(n logn),averagely faster than merge sort (better cache)

```cpp
#include <iostream>
#include <vector>

int partition(std::vector<int> &vec, int low, int high) {
    int pivot = vec[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (vec[j] <= pivot) {
            i++;
            std::swap(vec[i], vec[j]);
        }
    }

    std::swap(vec[i + 1], vec[high]);
    return (i + 1);
}

void quickSort(std::vector<int> &vec, int low, int high) {
    if (low < high) {
        int pi = partition(vec, low, high);
        quickSort(vec, low, pi - 1);
        quickSort(vec, pi + 1, high);
    }
}
```

Counting Sort O(n+range) good for sorting small ranged values

```cpp
#include <iostream>
#include <vector>
using namespace std;

void countingSort(vector<int>& arr) {
    if (arr.empty()) return;

    // Find the maximum value in the array
    int maxVal = *max_element(arr.begin(), arr.end());

    // Create a frequency array (size: maxVal + 1)
    vector<int> count(maxVal + 1, 0);

    // Count occurrences of each element
    for (int num : arr) {
        count[num]++;
    }

    // Reconstruct the sorted array
    int index = 0;
    for (int i = 0; i <= maxVal; i++) {
        while (count[i] > 0) {
            arr[index++] = i;
            count[i]--;
        }
    }
}
```

Searching Methods

- **Unsorted array?** → Linear Search
- **Sorted array?** → Binary Search
- **Finding first/last occurrence?** → Lower/Upper Bound Binary Search
- **Unimodal array (peak search)?** → Ternary Search
- **Large datasets?** → Jump Search or Fibonacci Search
- **Infinite array?** → Exponential Search
- **Uniformly distributed data?** → Interpolation Search

Linear Search

```cpp
#include <iostream>
p/ETUDIE/CODING MATERIALS/.vscode
#include <algorithm>

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5, 8, 9, 11};
    int key = 8;

    auto it = std::find(v.begin(), v.end(), key);

    if (it != v.end())
        std::cout << key << " Found at Position: " << it - v.begin() + 1;
    else
        std::cout << key << " NOT found.";

    return 0;
}
```

Iterative Binary Search

```cpp
#include <iostream>

int binarySearch(int arr[], int low, int high, int x) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == x) return mid;
        if (arr[mid] < x) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

Recursive Binary Search

```cpp
#include <iostream>

int binarySearch(int arr[], int low, int high, int x) {
    if (high >= low) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == x) return mid;
        if (arr[mid] > x) return binarySearch(arr, low, mid - 1, x);
        return binarySearch(arr, mid + 1, high, x);
    }
    return -1;
}
```