SCS1308 - Foundations of Algorithm

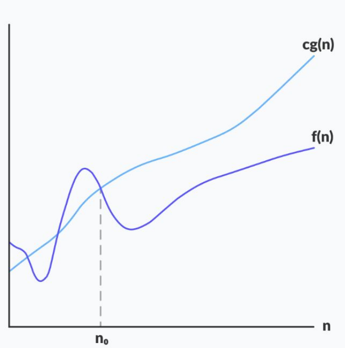
Tutorial - 01

Time Complexity & Recursion

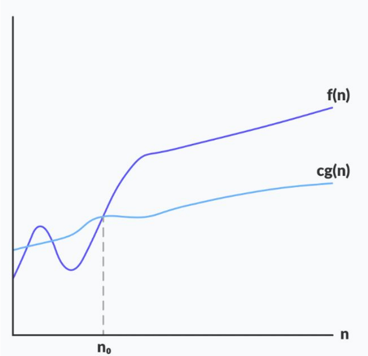
What is an Algorithm?

Algorithm Efficiency

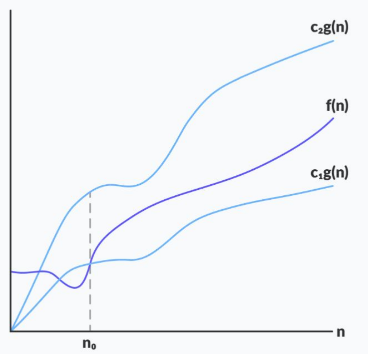
- ➔ Efficiency depends on time and space complexity.
- ➔ Types of analysis
  - Worst case
  - Best case
  - Average case
- ➔ Comparisons often focus on growth rates (Big-O, Omega, Theta).



f(n) = O(g(n))



f(n) = Ω(g(n))



f(n) = Θ(g(n))

Analyzing Iterative Algorithms

- Loops - Depends on the number of iterations
- Nested loops - Multiply the number of iterations of each loop
- Consecutive statements - If multiple independent statements execute sequentially, their complexities are added.
- If-then-else-statement - Complexity depends on the branch that is executed most frequently or the most expensive branch.
- Logarithmic Complexity - Algorithms exhibit logarithmic complexity when the input size is divided in each iteration.

Growth Rate Classes

We have seen that when we analyze functions asymptotically

- ➔ Constant: Θ(k), for example Θ(1)
- ➔ Linear: Θ(n)
- ➔ Logarithmic: Θ($\log_k$ n)
- ➔ nlogn: Θ(n$\log_k$ n)
- ➔ Quadratic: Θ(n^2)
- ➔ Polynomial: Θ(n^k)
- ➔ Exponential: Θ(k^n)

Only the leading term is important.

Constants don't make a significant difference.

The following inequalities hold asymptotically

$$c < \log n < \log 2^n < \sqrt{n} < n < n \log n < n^{(1.1)} < n^2 < n^3 < n^4 < 2^n$$

In other words, an algorithm that is Θ(nlog(n)) is more efficient than an algorithm that is Θ(n3).

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
0.6931	2	1.39	4	8	4
1.099	3	3.30	9	27	8
1.386	4	5.55	16	64	16
1.609	5	8.05	25	125	32
1.792	6	10.75	36	216	64
1.946	7	13.62	49	343	128
2.079	8	16.64	64	512	256
2.197	9	19.78	81	729	512
2.303	10	23.03	100	1000	1024
2.398	11	26.38	121	1331	2048
2.485	12	29.82	144	1728	4096
2.565	13	33.34	169	2197	8192
2.639	14	36.95	196	2744	16384
2.708	15	40.62	225	3375	32768
2.773	16	44.36	256	4096	65536
2.833	17	48.16	289	4913	131072
2.890	18	52.03	324	5832	262144
$\log \log m$	$\log m$				m

n	$100n$	n^2	$11n^2$	n^3	2^n
1	100	1	11	1	2
2	200	4	44	8	4
3	300	9	99	27	8
4	400	16	176	64	16
5	500	25	275	125	32
6	600	36	396	216	64
7	700	49	539	343	128
8	800	64	704	512	256
9	900	81	891	729	512
10	1000	100	1100	1000	1024
11	1100	121	1331	1331	2048
12	1200	144	1584	1728	4096
13	1300	169	1859	2197	8192
14	1400	196	2156	2744	16384
15	1500	225	2475	3375	32768
16	1600	256	2816	4096	65536
17	1700	289	3179	4913	131072
18	1800	324	3564	5832	262144
19	1900	361	3971	6859	524288

n	n^2	$n^2 - n$	$n^2 + 99$	n^3	$n^3 + 234$
2	4	2	103	8	242
6	36	30	135	216	450
10	100	90	199	1000	1234
14	196	182	295	2744	2978
18	324	306	423	5832	6066
22	484	462	583	10648	10882
26	676	650	775	17576	17810
30	900	870	999	27000	27234
34	1156	1122	1255	39304	39538
38	1444	1406	1543	54872	55106
42	1764	1722	1863	74088	74322
46	2116	2070	2215	97336	97570
50	2500	2450	2599	125000	125234
54	2916	2862	3015	157464	157698
58	3364	3306	3463	195112	195346
62	3844	3782	3943	238328	238562
66	4356	4290	4455	287496	287730
70	4900	4830	4999	343000	343234
74	5476	5402	5575	405224	405458

Recursion Basics

- Recursive algorithms consist of base cases and recursive cases.
- Whenever we analyze the run time of a recursive algorithm, we will first get a recurrence relation
- To get the actual run time, we need to solve the recurrence relation
- Recurrence relations express the overall time complexity.
- $T(n) = T(n/2)+1$ is an example of a recurrence relation
- A Recurrence Relation is any equation for a function T, where T appears on both the left and right sides of the equation.
- We always want to “solve” these recurrence relation by getting an equation for T, where T appears on just the left side of the equation

Methods to Solve Recurrences

1. Iteration Method
2. Substitution Method
3. Recursion Tree
4. Master's Theorem

Substitution Method

Steps:

1. Guess the form of T(n) based on the recurrence.
2. Prove the guess is correct by substituting it back into the recurrence relation.
3. Use induction to verify the solution holds for all n.

What is Induction?

Induction is a mathematical proof technique with two key steps:

1. Base Case: Prove the statement holds for the smallest input size (e.g., N=1).
2. Inductive Step: Assume the statement holds for a smaller input size (e.g., N=k) and then prove it holds for the next size (e.g., N=k+1).

For solving a recurrence, the induction step ensures that the guessed solution works for all input sizes

Example for Substitution method

Find the time complexity of the following recurrence relation : $T(N) = T(N-1) + c$

- Guess the form of solution: $T(N)=T(1)+c(N-1)$

- Prove Using Induction

  - ➔ For N=1 substitute into the guessed solution: $T(1) = T(1)$ (So the base case holds.)
  - ➔ Assume the solution holds for N=k : $T(k)=T(1)+c(k-1)$

Inductive Step:

- ➔ Prove the solution holds for N=k+1 Starting with the recurrence:

$$T(k+1)=T(k)+c$$

- ➔ Simplify by substituting T(k)

$$T(k+1)=T(1)+c(k-1)+c$$

$$T(k+1)=T(1)+c(k)$$

Ignoring constants T(1) and c, the dominant term is:

$$T(N)=\Theta(N)$$

Iteration Method

We keep on substituting the smaller terms again and again until we reach the base condition and find a pattern from it. Thus the base term can be replaced by its value, and we get the value of the expression.

T(N) = T(N-1) + c, Here T(0) = c and c is constant

<u>Solution:</u>

T(N) = T(N-1) + c, ------- 1

T(N-1) = T(N-2) + c ------ 2

T(N-2) = T(N-3) + c -------3

Applying 2 to 1

T(N) = (N-2) + c + c

T(N) = (N-3) + c + c + c

| | |

T(N) = (N-N) + c + c .... + c

T(N) = T(0) + N*c

$$\left.\begin{aligned} T(N) &= T(N-1) + c \\ T(N-1) &= T(N-2) + c \\ T(N-2) &= T(N-3) + c \\ &\ \ | \\ &\ \ | \\ T(1) &= T(0) + c \end{aligned}\right\} \text{N times c}$$

T(N) = 1 + N*c

Now, as c is constant T(N) = O(N).

Thus T(N) = O(N) using this method.

Activity 01

Find the time complexity for the following recurrence relation using iteration method

1. $T(N) = T(N-1) + N$
2. $T(N) = T(N-1) + N^2$
3. $T(N) = T(N-1) + \log N$

Recursion Tree Method

Steps:

1. Build the tree

2. Compute TC per level

3. Compute number of levels

   (find last level as a function of N)

4. Compute total over levels.

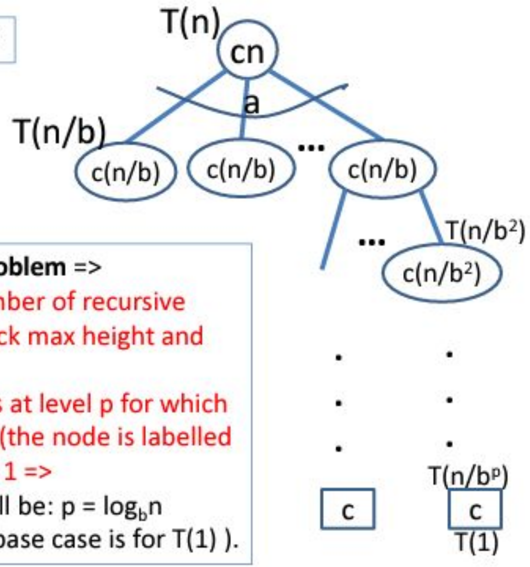
   * Find closed form of that summation.

$T(1) = c$

Problem size

The local TC at the node

$T(n) = a*T(n/b) + cn$



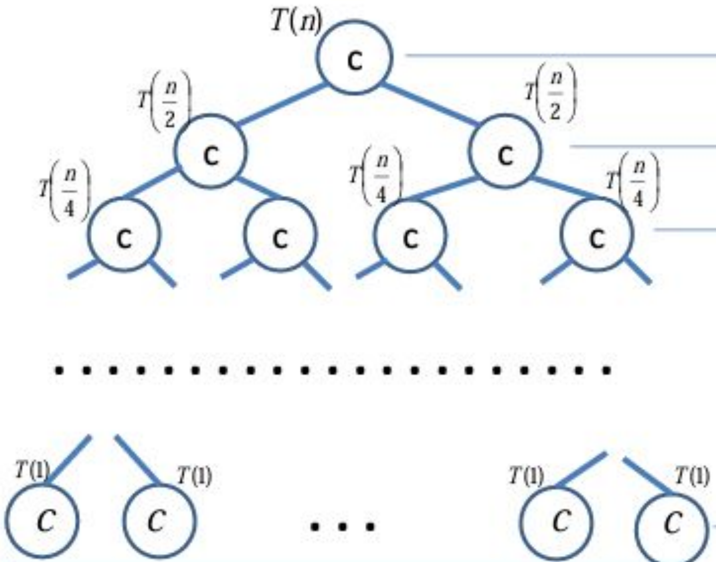
Number of subproblems => Number of children of a node in the recursion tree. => Affects the number of nodes per level. At level i there will be a^i nodes. Affects the level TC.

Size of a subproblem => Affects the number of recursive calls (frame stack max height and tree height) Recursion stops at level p for which the pb size is 1 (the node is labelled T(1)) => $n/b^p = 1$ => Last level, p, will be: $p = \log_b n$ (assuming the base case is for T(1)).

TC = time complexity

Recursion Tree for: $T(n) = 2T(n/2)+c$

Base case: $T(1) = c$



Level	Arg/ pb size	TC of 1 node	Nodes per level	Level TC
0	n	c	1	c
1	n/2	c	2	2c
2	n/4	c	4	4c
...				
i	$n/2^i$	c	2^i	2^ic
...				
p=lgn	1 ($=n/2^p$)	c	2^p ($=n$)	2^kc

Stop at level p, when the subtree is T(1).
=> The problem size is 1, but the general formula for the problem size, at level p is:
$n/2^p \Rightarrow n/2^p = 1 \Rightarrow p = \lg n$

Tree TC $= c(1+2+2^2+2^3+\ldots+2^i+\ldots+2^p) = c2^{p+1}/(2-1)$
$= 2c2^p = 2cn = \Theta(n)$

Master's Theorem

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n). \quad \text{Here} \text{ ----> } \quad f(n) = O(n^k \log^p n)$$

where $a \geq 1$ and $b > 1$ are constants and f(n) is an asymptotically positive function.

Case 1 : if $\log_b a > k$

Case 2 : if $\log_b a = k$

Case 3 : if $\log_b a < k$

1) If $a > b^k$, then $T(n) = \Theta\left(n^{\log_b^a}\right)$
2) If $a = b^k$
   a. If $p > -1$, then $T(n) = \Theta\left(n^{\log_b^a} \log^{p+1} n\right)$
   b. If $p = -1$, then $T(n) = \Theta\left(n^{\log_b^a} \log\log n\right)$
   c. If $p < -1$, then $T(n) = \Theta\left(n^{\log_b^a}\right)$
3) If $a < b^k$
   a. If $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

Think about the Master Theorem for the following form

T(n) = aT(n-b) + f(n).

$$General\ Form:\ T(n) = a\,T(n-b) + f(n)$$

$$where,\ a > 0,\ \ b > 0,\ k \geq\ 0\ and\ f(n) = \theta(n^k).$$

$$Case1:\ a = 1,\ O(n * f(n))$$

$$Case2:\ a > 1,\ O(n^k\ a^{n/b})$$

$$Case3:\ a < 1,\ O(n^k)$$

Activity 02

For each of the following recurrences, give an expression for the runtime T (n) if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

1. $T(n) = 3T(n/2) + n^2$
2. $T(n) = 4T(n/2) + n^2$
3. $T(n) = T(n/2) + 2^n$
4. $T(n) = 2^n T(n/2) + n^n$
5. $T(n) = 16T(n/4) + n$

Solutions

1. $T(n) = 3T(n/2) + n^2 \implies T(n) = \Theta(n^2)$ (Case 3)
2. $T(n) = 4T(n/2) + n^2 \implies T(n) = \Theta(n^2 \log n)$ (Case 2)
3. $T(n) = T(n/2) + 2^n \implies \Theta(2^n)$ (Case 3)
4. $T(n) = 2^n T(n/2) + n^n \implies$ Does not apply (a is not constant)
5. $T(n) = 16T(n/4) + n \implies T(n) = \Theta(n^2)$ (Case 1)

Thank you