Analysis

08 February 2021 06:00 PM

- Algorithms:

  Combination of sequence of finite states to solve a problem.

- properties of Algorithm:
  1. It should terminate within finite time.
  2. should produce atleast 1 output
  3. May take input may not take input
  4. Algorithm should be deterministic
  5. Every statement should have some use
  6. Independant from programming language

- steps required to desing an algorithm.
  1. Problem statement
  2. Select Design Technique.
  3. Flowchart
  4. Verification
  5. coding
  6. Analysis.

- Time complexity.

  T(program) = compile time (c) + Running time (R)

  depends on language of compiler — Based on CPU

Analysis	Analysis
① dependant on language of compiler & Type of hardware	① Independant of language of compiler & type of hardware
② Gives exact answer	② Gives approximate answer
③ Accuracy is more	③ Accuracy less
④ Different from system to system	④ Same across all systems
⑤ Takes time to calculate	⑤ Quick in calculation

Time Complexity Problems

17 March 2021 12:05 PM

– Apriori Analysis

Determination of order of magnitude of a statement.

①
```
main() {
    x = y + z;        ——— 1
}
```
→ Find out number of statements and times it executes

$\therefore \quad \Theta(1) = \Omega(1) = O(1)$

All are correct.

∵ Best case and worst case equal.

②
```
main() {
    x = y + z;                    ——— 1
    for (i = 1; i ≤ n; i++) {     .... n
        x = y + z;                ——— n
    }
}
```
→ total order of magnitude = $n+1$

$\therefore \quad T(P) = O(n)$

③
```
main() {
    x = y + z;                    ——— 1
    for (i = 1; i <= n; i++)      .... n   ← doesn't do any work
        x = y + z;                ——— 1 x n
    for (i = 1; i <= n; i++)      .... n
        for (j = 1; j <= n; j++)  .... n x n
            x = y + z             ——— 1 x n x n
}
```
$\Omega(n)$ ← Also correct

$= \quad 1 + n + n^2 = \Theta(n^2)$

$O(n^3)$ ← Also correct.

$T(P) = \Theta(n^2)$

Ω	: complexity smaller or equal than biggest function	$\Omega(n^2), \Omega(n), \Omega(1)$ All correct
Θ	: Biggest function should be equal	only $\Theta(n^2)$ correct.
O	: complexity bigger or equal to biggest function	$O(n^2), O(n^3)$ .... correct.

④
```
main() {
    i = 1                         ——— 1
    while (i < n) {               .... n-1
        i = i + 1                 ——— n-1
    }
}
```
$= 1 + n - 1 \quad = n$

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

⑤
```
main() {
    i = 1                         ——— (1)
    while (i <= n) {              ——— (n/2 + 1)
        i = i + 2;                ——— n/2
    }
}
```
$= \quad 1 + n/2 + 1 \; n/2$

$= \quad 2 + n$

$= n + 2$

$\therefore T(P) = \Theta(n)$

⑥
```
main() {
    while (n>1) {        — n
        n -= 1           — n-1
    }
}
```

$= n + n - 1$
$= 2n - 1$

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

⑦
```
main() {
    i = 1;
    while( i<n )
        i = i * 2;
}
```

loop will execute till $2^k < n$

$\therefore k = \log_2 n$

$T(P) = \log_2 n$

⑧
```
main() {
    i = n;
    while( i>1 )
        i = i/2          dec by 1/2
}                        ∴ log2 n
```

$T(n) = \log_2 n$

⑨
```
main() {
    i = 1;
    while( i < n)
        i = 10 * i;      increasing exponentially
        i = i/2
}
```

$\therefore T(P) = \log_5 n$

⑩
```
main() {
    i = 2;
    while (i < n)
        i = i^2
}
```

Always find k

$(i^2)^k < n$
$(2^2)^k < n$
$2^{(2k)} < n$
$2k \log_2 2 < \log_2 n$
$2k < \log n$ / $\log 2$
$2k < \log_2 n$
$k \log_2 2 < \log_2 \log_2 n$
$\therefore k < \log_2 \log_2 n$

$\therefore T(P) = \log_2 \log_2 n$

```
main() {
    i = 23;
    while (i < n) {
        i = i^27
        i = i^2
    }
```

```
        i = i^2
    }
}
```

$$(i^{27^2})^k < n$$
$$(23^{27^2})^k < n$$
$$\therefore \quad k < \log_2 \log_{27} \log_{23} n$$

(11)
```
main() {
    while(n > 2) {
        n = n^(1/2)
    }
}
```

find this value in condition $n = 2$

$$(n^{1/2})^k > n'$$
$$\left(\frac{1}{2}\right)^k \log 2 > \log n'$$
$$k \log(1/2) > \log_2 n'$$
$$k \log 2^{-1} > \log \log_2 n'$$
$$-k \log 2 >$$
$$-k > \log_2 \log_2 n'$$
$$k > -\log_2 \log_2 n'$$

$$\therefore \quad k > -\log_2 \log_2 n$$

$$T(P) = \log_2 \log_2 n$$

(12)
```
main() {
    i = 2;
    while (i<n) {
        i = i^2
        i = i^5
        i = i^7
    }
}
```

$$(i^{2\times5\times7})^k < n$$
$$(2^{70})^k < n$$
$$k < \log_{70} \log_2 n$$

(13)
```
main() {
    i = 5
    while (i < n) {
        i = i+2;
        i = i*2;
        i = i^2;
    }
}
```

$$\left(\left[(i+2).2\right]^2\right)^k < n$$
$$\left((2)\times2\right)^{2^k} < n$$
$$(4)^{2^k} < n$$
$$2^k < \log_4 n$$
$$k < \log_2 \log_4 n$$

This simplification is diffucult

Hence find the updation that affects value more

$i = i^2$

$T(P) = \log_2 \log_2 n$

(14)
```
main(){
  for (i=1; i<=n^2; i++)                    ——— n^2
    for (j=n; j>1; j=j-2)                   ——— n/2
      for (k=5; k<n^7; k=k^3)               ——— log_3 log_5 log n^7
        x = y+z;
}
```

$= n^2 \cdot \frac{n}{2} \cdot \log_3 \log_5 n^7$

$= \frac{n^3}{2} \log_3 (7 \cdot \log_5 n)$

$= \frac{n^3}{2} [\log_3 7 + \log_3 \log_5 n]$

$= \frac{n^3}{2} \log_3 7 + \frac{n^3}{2} \log_3 \log_5 n$

$= \Theta(n^3 \log 7)$

(15)
```
main(){
  for (i=1; i<=n; i++)                      n
    for (j=1; j<=i; j++)                    n(n+1)/2   ← dependant
      x = y+z;
}
```

$T(P) = \frac{n(n+1)}{2}$

$= \Theta(n^2)$

(16)
```
main(){
  for (i=1; i<=n; i++)                      n.
    for (j=1; j<=n; j=j+i)                  n/i   ← Always Adding i
      x = y+z
}
```

$T(P) = \frac{n}{1} + \frac{n}{2} + \frac{n}{3} \cdots$

$= n \left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} \cdots \right)$

$= n \cdot \log_2 n$

$T(P) = \Theta(n \log n)$

(17)
```
main(){
  p=0 : q=0
  for (i=1; i<=n; i=2*i)                    log n
    p++
    for (j=1; i<p; j=2*j)                   log p
      q++;
}
```

$p = \log n$

$q = \log \log n$

$\therefore$

$T(P) = \log n + \log p$

$= \log n + \log \log n$

$T(P) = \Theta(\log n)$

(18)
```
main() {
    int i;
    for (i=1; i<=n; i++)          ——— ⊥
        for (i=1; i<n^2; i++)     ——— ⊥
            for (i=1; i<=n^3; i++) ——— n^3
                x = y+z
}
```
All i loops. Fails. Finally $i = n^3+1$

$T(P) = n^3$

(19)
```
main() {
    i=1, j=1;
    while (i<=n) {
        j++;
        i = i+j;        ← adds 1, 2, 3, 4...
    }
}
```
$\frac{k^2+k}{k^2+k} < n$

$k^2 < 2n$

$k < \sqrt{2n}$

$\therefore k < \sqrt{n}$

$\therefore T(P) = \Theta(\sqrt{n})$

(20)
```
main() {
    for (i=1; i<=n; i++) {        ← prime.
        if (n % i == 0)           ——— 2
            for (j=1; j<=n; j++)  ——— 2n
                x = y z;
    }
}
```
$T(n) = n \mid 2n$

$= 3n$

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

(21)
```
main() {
    for (i=1; i<=n; i++) {        ——— n
        if (n % i == 0)           — min 2, max n/2
            for (j=1; j<=n; j++)
                x = y+z;
    }
}
```
worst case $T(n) \quad \frac{n \cdot n}{2} = \frac{n^2}{2} = \Theta(n^2)$

Best case $T(n) = n \mid 2 = \Omega(n)$

Always $\geq$ LHS

$T(P) = O(n^2)$ $O(n^3)$ Also right

$\Omega(n)$ $\Omega(1)$ Also right

Always $\leq$ LHS

$T(P)$ → n or n^2

(22)
```
main() {
    for (i=2; i<=√n; i++) {       ——— √n
        if (n % i == 0) {
            print not prime
            return 0;
```

```
    }
  }
}
```

Best case if n is even

$T(P) = 1$

Worst case if n is prime

$T(P) = \sqrt{n}$

(23)

```
main() {
  for( i=1 ; i≤n ; i++)          ——— n
    for( j=1 ; j≤i ; j++)        ——— 1 + 2 + 3 + .... n
      for( k=1 ; k≤j ; k++)      ——— 1 + (1+2) + (1+2+3) + .... n
        x = y + z
}
```

i, j, k not updating by 1, 2, 3 ∴ ans ≠ loglogn

$$T(P) = 1 + (1+2) + (1+2+3) \cdots$$

$$= \sum_{i=1}^{n} \frac{i(i+1)}{2}$$

$$= \frac{1}{2} \sum_{i=1}^{n} (i^2 + i)$$

$$= \frac{1}{2} \left[\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i \right] \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$= \frac{1}{2} \left[\frac{n(n+1)(2n+1)}{2} + \frac{(n+1)n}{2} \right] \qquad \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$= \frac{1}{4} (n^2+n)(2n+1) + \frac{n^2+n}{4}$$

$$= \frac{1}{4} (2n^3 + n^2 + 2n^2 + n + n + n^2)$$

$\therefore T(n) = \Theta(n^3)$

(24)

```
for (i=n , j=1 ; i≤n ; i/=2 ; j+=i)
```

Infinite loop

Not Algo.

Asymptotic notation

16 April 2021 06:56 AM

1. Big Oh Notation (O)
2. Omega Notation (Ω)
3. Theta Notation (Θ)

Big Oh (O)

16 April 2021 09:00 AM

① Big-oh Notation

$f(n)$ & $g(n)$ two positive functions

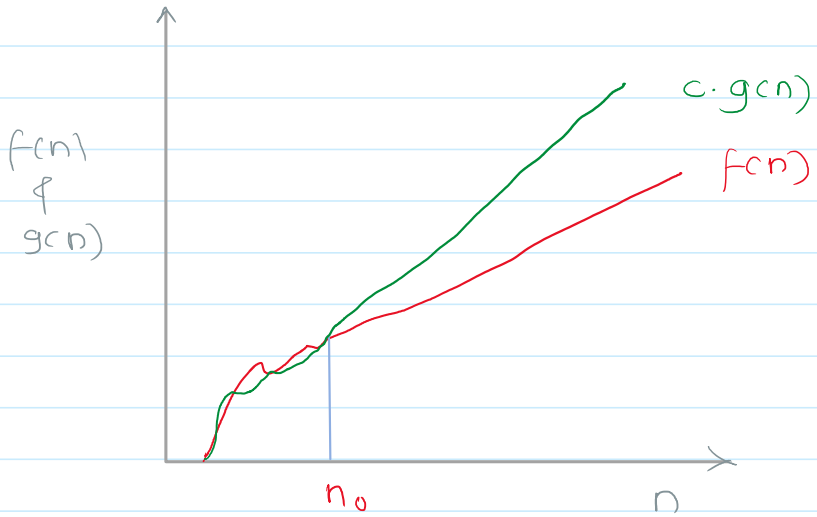
$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$, $\forall n$, $n \geq n_0$

such that $\exists$ 2 positive constants

For some $c > 0$ and $n_0 \geq 1$

RHS greater or equal for all

for c, n_0 fractions also allowed but for simplicity avoid fractions



$n^2 = O(n^2)$ ← Tightest upper bound

$= O(n^3)$

$= O(n^4)$

$= O(n^{100})$

Non-tightest upperbound

Upper bounds.

small-oh (o) (o, <)

small-oh means strictly less than

Here no tightest upperbound exists.

only non-tight upper bounds.

$n^2 = o(n^3)$

$= o(n^7)$

$n^2 = O(n^3)$
$= O(n^5)$
$= O(n^{100})$

Ex.

1. $f(n) = n + 10$
$g(n) = n$

We have to prove
$f(n) \leq c.g(n)$
$\therefore n + 10 \leq c.n$

$c = 2$ ← smallest value that satisfies the above equation

$\therefore n + 10 \leq 2n$

if $n = 1$,
$f(n) \leq c \cdot g(n)$ fails

so the minimum value from which the above equation holds true till infinite.
$n_0 = 10$

for $n = 1$ to 10 above eqn fails
but from n_0 till infinite it holds true

$n + 10 = O(n)$

Big-oh means proving $g(n)$ is bigger from n_0 onwards after taking constant

Big-oh means proving g(n) is bigger from n_0 onwards after taking constant help c.

2. $f(n) = n$
$g(n) = n+10$

proving

$f(n) \leq c \cdot g(n)$

$n \leq c \cdot (n+10)$

for $c = 1$ the above function holds true.

put $n = 1$

$1 \leq 11$ ← True

From $n = 1$ till infinity it satisfies

$\therefore n_0 = 1$

$\therefore n = O(n+10)$

3. $f(n) = n^2 + n + 10$
$g(n) = n^2$

proving

$f(n) \leq c \cdot g(n)$

$n^2 + n + 10 \leq c \cdot n^2$

Consider $c = 2$

$\therefore$ for finding n_0

put $n = 1$

$1 + 1 + 10 \leq 1 \cdot 1$

$12 \leq 1$ ← fails

put $n = 2$

$4 + 2 + 10 \leq 2 \times 4$

$16 \leq 8$ ← fails but its getting closer

put $n = 4$

$16 + 4 + 10 \leq 2 \times 16$

$32 \leq 32$ ← passed

∴ For $c = 2$ from $n_0 = 4$ till infinity above equation holds true

Generally if you increase value of c then value of n_0 will drop.

3. $f(n) = n^2$

$g(n) = n$

proving

$f(n) \leq c \cdot g(n)$

For proving the above statement value of c should be minimum 'n'. ← consider n to be infinite we have to put c = constant

But c can't be variable.

∴ No constant c where the above equation holds true

∴ $n^2 \neq O(n)$

Omega Notation (Ω)

16 April 2021 08:27

- Omega Notation (Ω, $\geq$)

$f(n)$ & $g(n)$ two positive functions

$f(n) = \Omega(g(n))$ → RHS smaller or equal

iff $f(n) \geq c \cdot g(n)$, $\forall n$, $n \geq n_0$

such that $\exists$ 2 positive constants

$c > 0$ and $n_0 \geq 1$

Ex.

1. $f(n) = n$

$g(n) = n + 10$

prove,

$f(n) \geq c \cdot g(n)$

$n \geq c \cdot (n+10)$

for $c = 1/2$, from $n_0 = 10$ the above equation holds true

$\therefore n = \Omega(n+10)$

2. $f(n) = n + 10$

$g(n) = n$

proving.

$n + 10 \geq c \cdot n$

for $c = 1$ Above equation holds true from $n_0 = 1$

3. $f(n) = n$

$g(n) = n^2$

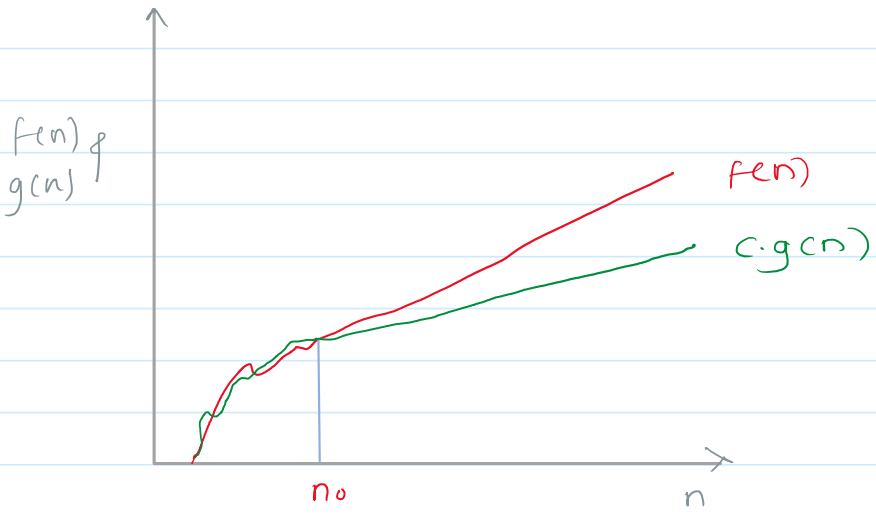
proving

$f(n) \geq c \cdot g(n)$

proving above equation true
we have to take $c = 1/n$

but c can't be variable

$\therefore \quad n \neq \Omega(n^2)$



$= \Omega(n^4)$ ✗ Not possible

$n^3 = \Omega(n^3)$ ✓ ← Tightest lower bound

$= \Omega(n^2)$ ✓

$= \Omega(n)$ ✓ } Non-tightest lower bounds

$= \Omega(1)$ ✓

} Lower bounds

small omega (ω) ($\omega, >$)

$A = \omega(B)$

where B is Non-tightest lower bound only.

Theta Notation (Θ)

16 April 2021 08:47

- Theta Notation (Θ) (O , ≤)

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

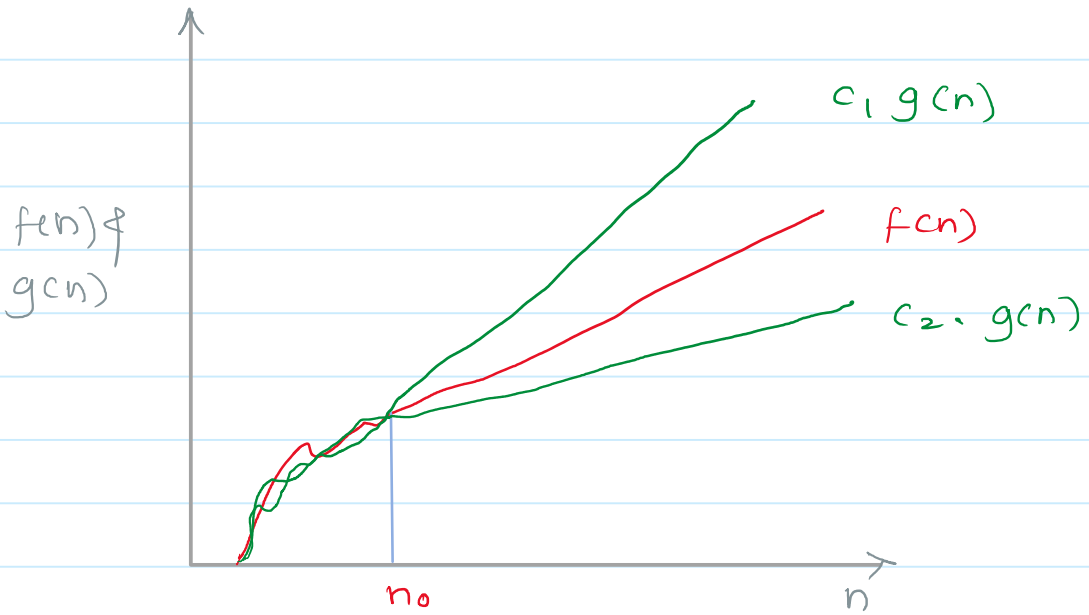
Iff ,

$f(n) \leq c_1 . g(n)$

$\forall\ n > n_0$

$f(n) \geq c_2 . g(n)$

c_1 & c_2 are constants where n_0 onwards above equations hold true.



$n^2 = O(n^2)$ ✓

$= \Omega(n^2)$ ✓

When both are possible then $n^2 = \Theta(n^2)$

Ex.

1. $f(n) = n + 10$

1. $f(n) = n + 10$
$g(n) = n$

proving
$f(n) \geq c_1 . g(n)$
$f(n) \leq c_2 . g(n)$

For $c_1 = 2$ & $c_2 = 1/2$
Above equations hold true
$n_0 = 10$ onwards

$\therefore$ $n + 10 = \Theta(n)$

In short, if Big-oh (O) and Big Omega (Ω) both are possible Then only Theta (Θ) possible

2. $f(n) = n$
$g(n) = n$

proving
$f(n) \geq c_1 . g(n)$
$f(n) \leq c_2 . g(n)$

$\therefore$ for $c_1 = c_2 = 1$ above equation holds true $n_0 = 1$ onwards

$\therefore$ $n = \Theta(n)$

3. $f(n) = n$
$g(n) = n^2$

proving
$n \leq c_1 n^2$: $n = O(n^2)$

proving

$n \leq c_1 n^2$: $n = O(n^2)$

$n \geq c_2 n^2$: $n \neq \Omega(n^2)$

$\therefore \quad n \neq \Theta(n^2)$

Complexity Classes

16 April 2021 05:11 PM

All are positive functions, in increasing order

① Decreasing functions

e.g. $\frac{1}{n}, \frac{1}{n^2}, \frac{1}{n^3} \ldots \quad c^n, 0 < c < 1$

i.e. as n increases, value decreases

② Constant functions

e.g. $1, 10, \log 15, 10.\log_{10} 5$

Constants never change

③ Logarithmic functions

e.g. $\log n, \log n^2, (\log n)^8$

$\log\log\log n < \log\log n < \log n$

④ Linear functions

e.g. n ← power of 1

⑤ Quadratic functions

e.g. n^2 ← powers of 2

⑥ Cubic functions

e.g. n^3 ← powers of 3

⑦ Polynomial functions $\sqrt{n} = n^{1/2}$

e.g. $n^{0.1}, n^1, n^2, \ldots$ ← powers constant

n^c, where c = constant & $c > 0$

$\log\log n < \log n < n^{1/2} < n < n\log n$

⑧ Exponential functions

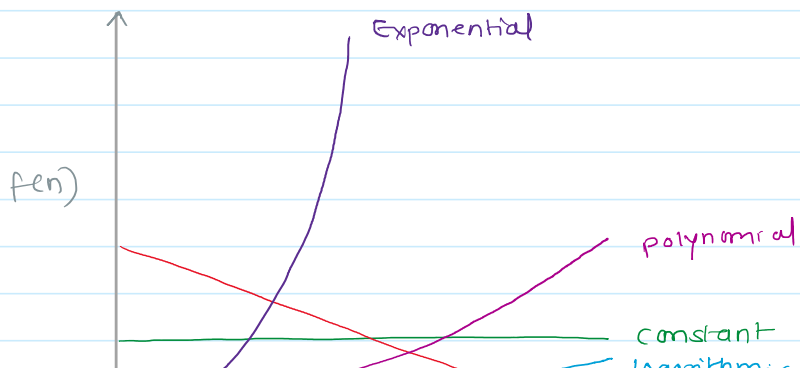
e.g. $3^n, 4^n \ldots$ ← powers of n

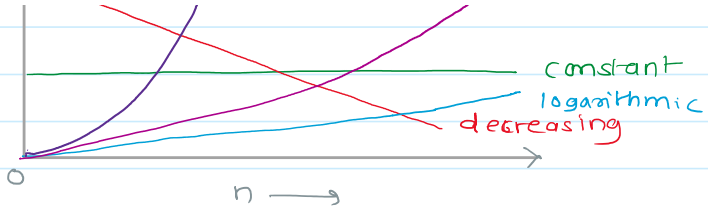
c^n ← 1^n ∉ exponential

c is constant $c > 1$

if $c < 1$ then

c^n decreasing function





Q1 ~~A.~~ ✓ $100\, n \log n = O\left(\frac{n \log n}{100}\right)$

~~B.~~ ✗ $\sqrt{\log n} = O(\log \log n)$

~~C.~~ ✓ If $0 < x < y$ then $n^x = O(n^y)$

~~D.~~ ✗ $n^c \neq O(c^n)$, where c is constant & $c > 1$

A. $100 n . \log n = c\, \frac{n . \log n}{100}$

put $c = 1000 \therefore n_0 = 1$

B. $(\log n)^{1/2} = O(\log \log n)$

$(\log n)^{1/2} = c \log \log n$

put $\log n = x$

$\therefore x^{1/2} = c . \log x$

polynomial — logarithmic

$\therefore x^{1/2} > c . \log x$

$(\log n)^{1/2} :: \log \log n$

Apply log.

$\frac{1}{2} \log \log n > \log \log \log n$

$\therefore (\log n)^{1/2} > \log \log n$

$\therefore$ Big-oh not possible

C. $0 < x < y$

$n^x = c . n^y$

$x < y$

$\therefore n^x < c\, n^y$

Big oh possible.

$\therefore$ True.

D. $n^c \neq O(c^n)$

polynomial — Exponential

polynomial Expnential

$n^c \leq c^n$

$\therefore n^c = O(c^n)$

↑ equal

Asymptotic Notation Problems

17 February 2021 11:48 PM

- State True / False.

① $2^{n+1} = O(2^n)$

$2^{n+1} = 2 \cdot 2^n$ $\quad c \cdot 2^n$

$2^{n+1} = O(2^n)$ True

② $\frac{4^n}{2^n} = O(2^n)$

$\frac{4^n}{2^n} = \left(\frac{4}{2}\right)^n$

$= 2^n$

$= O(2^n)$ True.

③ $2^{2n} = O(2^n)$

$2^{2n} = 2^n \cdot 2^n$ cannot be converted to $c \cdot 2^n$

← Not constant.

④ $4^{\log_2 n} = O(n^{10})$ $\quad A^{\log B} = B^{\log A}$

$4^{\log_2 n} = n^{\log_2 64}$

$= n^{\log_2 2^6}$

$= n^{6 \log 2}$

$= n^6$

$n^6 = O(n^{10})$ True

⑤ $f(n) = O((f(n))^2)$ → False ∵ if you put $n \to \frac{1}{n}$ value decreases

$f(n) = O\left(\frac{f(n)}{2}\right)$ → True ∵ you can ignore constants

$f(n) = O\left(f\left(\frac{n}{2}\right)\right)$ → false ∵ $f(n) \Rightarrow 2^{2n} > f(n/2) = 2^n$

LHS > RHS

If $f(n) = O(g(n))$ then → False ∵ $f(x) = n \quad g(n) = n/2$

$2^{f(n)} = O(2^{g(n)})$ $\quad f(x) = O(g(n)) = n$

$\therefore 2^n \neq O(2^{n/2})$

⑥ Arrange in asymptotic increasing order

$f_1 = 2^n$ $\quad f_3 = n^3$

$f_2 = n^{\log n}$ $\quad f_4 = n!$

1. n^3
2. $n!$
3. 2^n
4. $n^{\log n}$

⑦ $\log_2 n = O(\log_3 n)$ T/F?

$\log_2 n = \left(\frac{\log_2 n}{\log_2 3}\right) \cdot \log_2 3$ ← constant

∴ TRUE

8. True or False:

① $10000 = O(c)$

$c = 10000$ $\quad - O(1)$ $\quad \therefore$ True

② $\frac{1}{n} = \Theta(1)$

$1 = O(n) \neq \Theta(n)$ RHS small

$\therefore$ false

③ $1000 = \Theta(1)$

1000 is constant

$\therefore 1 - \Theta(1)$

True

④ $1000 - O\left(\frac{1}{n}\right)$

$n = O(1)$

$\therefore$ false.

9. Let $f(n)$, $g(n)$ and $h(n)$ be 3 funtions defined beow

① $f(n) = O(g(n))$ & $g(n) \neq O(f(n))$

$f(n) < g(n)$

② $g(n) = O(h(n))$ & $h(n) = O(g(n))$

$h(n) = g(n)$

A	$f(n) \cdot g(n) = O(g(n) \cdot h(n))$	TRUE	
B	$f(n) + g(n) = O(h(n))$	TRUE	
C.	$h(n) = \Theta(f(n))$	FALSE	(should be Ω)
D	$g(n) \cdot h(n) = \Theta(h(n) \cdot h(n))$	TRUE	

10. $T_1(n) = O(f(n))$ $\quad T_1(n) \leq f(n)$

$T_2(n) = O(f(n))$ $\quad T_2(n) \leq f(n)$

A	$T_1(n) + T_2(n) = O(f(n))$	TRUE	
B	$T_1(n) = O(T_2(n))$	Not comparable	
C	$T_1(n) = \Omega(T_2(n))$	Not comparable	
D	$T_1(n) = \Theta(T_2(n))$	Not comparable	

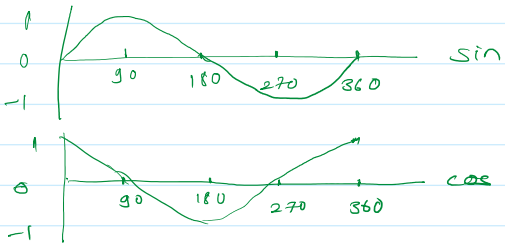
12. $f(n) = \begin{cases} n^3 & 0 < n < 100 \\ n^5 & n \geq 100 \end{cases}$

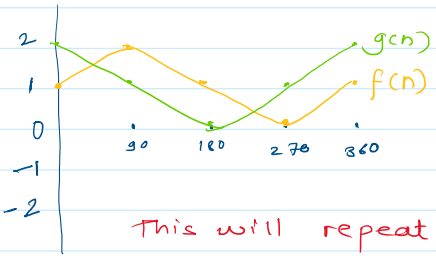
$g(n) = \begin{cases} n^7 & \text{if } 0 < n < 10000 \\ n^2 & \text{if } n > 10000 \end{cases}$

$$g(n) = \begin{cases} n^7 & \text{if } 0 < n < 10000 \\ n^2 & \text{if } n \geq 10000 \end{cases}$$

$$g(n) = O(f(n)) \qquad n \geq 10000$$

13. $f(n) = n^{1+\sin(n)}$
$g(n) = n^{1+\cos(n)}$
relation between $f(n)$ & $g(n)$

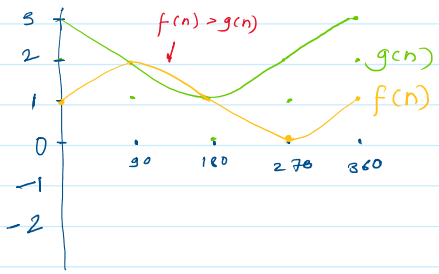




This will repeat again

∴ We can't compare $f(n)$ & $g(n)$.

(14) $f(n) = n^{1+\sin(n)}$
$g(n) = n^{2+\cos(n)}$

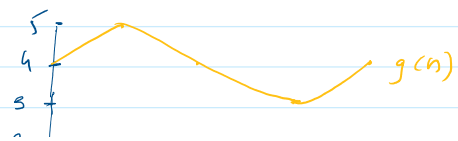


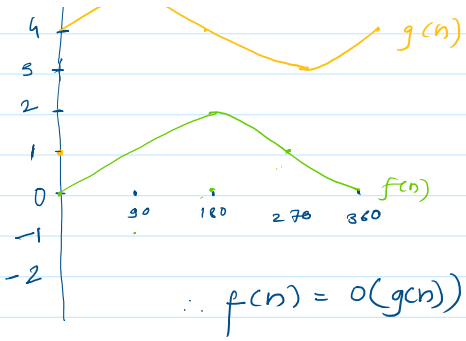
∴ We can't compare

(15) $f(n) = n^{1-\cos n}$ ← substract cos not ⊥
$g(n) = n^{2+\sin n}$

∴ Not comparable

(16) $f(n) = n^{1-\cos n}$ 0 – 2
$g(n) = n^{4+\sin n}$ 3 – 5





$\therefore f(n) = O(g(n))$

(17)

- $f_1 = n^{\log n}$
- $f_2 = (\log n)^{\log n}$
- $f_3 = n!$
- $f_4 = (\log n)!$
- $f_5 = (\log n!)$
- $f_6 = \sqrt{n}$
- $f_7 = (\log(\log n))!$
- $f_8 = 2^n$
- $f_9 = n^{\sqrt{n}}$

Increasing asymptotic order

$f_3 = n!$
$f_8 = 2^n$
$f_9 = n^{\sqrt{n}}$
$f_1 = n^{\log n}$
$f_2 = (\log n)^{\log n}$
$f_4 = (\log n)!$
$f_5 = \log(n!)$
$f_6 = \sqrt{n}$
$f_7 = (\log(\log n))!$

$n! < n^n$

But, $\log(n!) = \Theta(n \log n)$

According to stirling Appro.

$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

If two equations are strictly not equal (greater / smaller), after applying log there is a chance of equations to be asymptotically equal

$n^2 < n^3$
$\log n^2 < \log n^3$
$2 \log n < 3 \log n$
$\Theta(\log n) = \Theta(\log n)$

Properties of Asymptotic Notations

16 March 2021 11:44 PM

- Properties of Asymptotic Notations

① Reflexive property.

$f(n) = O(f(n))$

If Equal.

$\Omega, \Theta, O \Rightarrow$ satisfies

② Symmetric property.

If, $f(n) = \Theta(g(n))$
then,

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

$O, \Omega \Rightarrow$ fail
$\Theta \Rightarrow$ satisfies

③ Transitive property,

If $f(n) = O(g(n))$ &
$g(n) = O(h(n))$

then $f(n) = O(h(n))$

$\Omega, \Theta, O \Rightarrow$ satisfies.

④

If $f(n) = O(g(n))$
$d(n) = O(e(n))$

then $f(n) + d(n) = O(g(n) + e(n))$

Recursion

17 March 2021 11:57 AM

A function calling itself is Recursion

Ex.

```
int factorial (int n) {
    if (n <= 1)
        return 1;
    return n * factorial (--n);
}
```

Recursion

(1) solving big problems in terms of small problems.

(2) Every recursive problem will have two steps.

① When you will stop

then, ② If you don't stop, what you'll do further (recursion)

(3) For recursive programs stack data structure is used

(4) If recursion is infinite, it will terminate after some time with stack overflow error message.

∴ Every Recursive program should have Time complexity

(5) In recursion from one function to another, only parameter values change.

(6) Recursion will take more stack space than non-recursive programs

(7) Time complexity of a program imple-

-mented recursively or non-recursively will be asymptotically same.

(8) For every recursive program equiva-lent non-recursive program is there and vice-versa.

(9) While recursive program running, the number of stack units we use is the depth of the stack

- Recurrence relation

An equation which recursively defines a sequence where next term is a function of previous term.

for factorial

$$f(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n \cdot f(n-1) & \text{if } n > 1 \end{cases}$$

- Recurrence relation solving.

1. substitution method.
2. Recursive tree method
3. master's theorem.

Substitution Method

16 March 2021 11:03 PM

1. Substitution method:

Substituting the given function repeatedly until given function removed.

① $T(n) = \begin{cases} 1 & \dots n = 1 \\ T(n-1) + n & \dots n > 1 \end{cases}$

Without program given you don't know the purpose of above reccurrence relation ie. value or time.

$\Rightarrow T(n) = T(n-1) + n$ After every substitution

$= T(n-2) + (n-1) + n$ ⊥ non-recursive term will

$= T(n-3) + (n-2) + (n-1) + n$ be added to equation

This will go until $n - k = 1$

$= T(n-k) + (n-(k-1)) + (n-(k-2)) \dots + n$

$\therefore n - k = 1$

$k = n - 1$

substitute.

$= T(n-(n-1)) + (n-(n-2)) + (n-(n-3)) \dots n$

$= T(1) + 2 + 3 \dots + n$

$= 1 + 2 + 3 \dots + n$

$= \frac{n.(n+1)}{2}$

If,

Time complexity = $O(n^2)$ ⇒ Best case / worst case can't be judged just by upper / lower bound.

Value = $\frac{n.(n+1)}{2}$

② $T(n) = T(n-1).n$ if $n > 1$ | $T(n) = 1$ if $n = 1$

$= T(n-1).n$

$= T(n-2).(n-1).n$

$= T(n-3).(n-2)(n-1).n$

$T(1) = 1$

let $T(n-k) = T(1)$

$\therefore n - k = 1$

$k = n - 1$

substitute.

$= T(n-n+1)(n-n+2)(n-n+3) \dots n$

$= T(1).2.3.4 \dots n$

$= 1.2.3.4 \dots n$

$= n!$

Time complexity = $O(n^n) = \Omega(2^n)$

Value = $n!$

③ $T(n) = 2$ if $n = 0$

$= T(n-2) + \log(n)$ if $n > 0$

$T(n) = T(n-2) + \log(n)$

$= T(n-4) + \log(n-2) + \log(n)$

$= T(n-6) + \log(n-4) + \log(n-2) + \log(n)$

$= T(n-2k) + \log(n-(2k-2)) + \log(n-(2k-4)) \dots$

As 2k should always be even to get T(0) after (-2)

$\therefore n - 2k = 0$

$k = n/2$

$= T\left(n - 2\cdot\frac{n}{2}\right) + \log(n-n+2) + \log(n-n+4)\ldots$

$= T(0) + \log(2) + \log(4) \ldots + \log(n)$

$= 2 + \frac{n}{2}\log 2 + \log 1 + \log 2 \ldots + \log\frac{n}{2}$

$= 2 + \frac{n}{2}\cdot 1 + (\log 1 + \log 2 + \log 3 \cdots \log\frac{n}{2})$

$= 2 + \frac{n}{2} + \log\left(\frac{n}{2}!\right)$

Time complexity $= O\left(\log\left(\frac{n}{2}!\right)\right) = O\left(\log\left[\left(\frac{n}{2}\right)^{n/2}\right]\right) = O\left(\frac{n}{2}\log\left(\frac{n}{2}\right)\right)$

Value $= 2 + \frac{n}{2} + \log\left(\frac{n}{2}!\right)$

④ $T(n) = \begin{cases} 2 & \text{if } n = 0 \\ T(n-2) + n^2 & \text{if } n > 0 \end{cases}$

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

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

$= T(n-6) + (n-4)^2 + (n-2)^2 + n^2$

$= T(n-2k) + (n-2k+2)^2 + (n-2k+4)^2 \cdots n^2$

$T(n-2k) = T(0)$

$n - 2k = 0$

$k = \frac{n}{2}$

substitute

$= T\left(n - \frac{2n}{2}\right) + (n-n+2)^2 + (n-n+4)^2 \cdots n^2$

$= 2 + 2^2 + 4^2 + 6^2 \cdots n^2$

$= 2 + 2^2(1^2 + 2^2 + 3^2 + 4^2 \ldots (\frac{n}{2})^2)$

sum of squares of n natural numbers $= \frac{n\cdot(n+1)(2n+1)}{6}$

$= 2 + 2\left(\frac{n/2(n/2+1)\cdot(n+1)}{3}\right)$

$= 2 + \frac{n(n/2+1)(n+1)}{3}$

$= 2 + \frac{(n^2/2 + 1)(n+1)}{3}$

$= 2 + \frac{n^3}{6} + \frac{n^2}{6} + \frac{n}{3} + \frac{1}{3}$

Time complexity $= O(n^3)$

⑤ $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + c & n > 1 \end{cases}$

⑥ $T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + n & \text{if } n > 1 \end{cases}$

(7) $T(n) = \begin{cases} 1 & n=1 \\ 7 \cdot T(n/2) + n^2 & \text{if } n>1 \end{cases}$

(8) $T(n) = \begin{cases} 2 & n=2 \\ 2T(n/2) + n\log n & \text{if } n>2 \end{cases}$

(9) $T(n) = \begin{cases} 2 & n=2 \\ \sqrt{n} \cdot T(\sqrt{n}) + n & n>2 \end{cases}$

(10) $\begin{cases} 1 & n=1 \\ 2(T-1) + n & n>1 \end{cases}$

(5) $T(n) = \begin{cases} 1 & n=1 \\ T(n/2) + c & n>1 \end{cases}$

$$T(n) = T\left(\frac{n}{2}\right) + c$$
$$= T\left(\frac{n}{4}\right) + c + c$$
$$= T\left(\frac{n}{8}\right) + c + c + c$$
$$= T\left(\frac{n}{2^k}\right) + kc$$

$\therefore \frac{n}{2^k} = 1$

$n = 2^k$

$\log n = k \log 2$

$\therefore k = \log_2 n$

Substitute

$$T(n) = T\left(\frac{n}{2^{\log_2 n}}\right) + (\log_2 n) \cdot c$$
$$= T\left(\frac{n}{n^{\log_2 2}}\right) + (\log_2 n) c$$
$$= T(1) + (\log_2 n) \cdot c$$
$$= 1 + (\log_2 n) \cdot c$$

Time complexity = $O(\log_2 n)$

(6) $T(n) = \begin{cases} 1 & n=1 \\ T(n/2) + n & n>1 \end{cases}$

$$T(n) = T(n/2) + n$$
$$= T\left(\frac{n}{2.2}\right) + \frac{n}{2} + n$$
$$= T\left(\frac{n}{2.2.2}\right) + \frac{n}{4} + \frac{n}{2} + n$$
$$= T\left(\frac{n}{2^k}\right) + \frac{n}{2^{k-1}} + \frac{n}{2^{k-2}} \cdots n$$

$\frac{n}{2^k} = 1$

$k = \log_2 n$

$= T\left(\frac{n}{2^{\log_2 n}}\right) + \frac{n}{2^{\log_2 n - 1}} + \frac{n}{2^{(\log_2 n)-2}} \cdots n$

$= 1 + n\left[\left(\frac{1}{2}\right)^{\log_2 n} + \left(\frac{1}{2}\right)^{(\log_2 n)-1} + \left(\frac{1}{2}\right)^{(\log_2 n)-2} + \left(\frac{1}{2}\right)^{0}\right]$

(r = $\frac{1}{2}$)

This is G.P. series

sum of n terms in GP = $\left[\frac{a(1-r^n)}{1-r}\right]$ ⇓ formula

cannot exceed 1 because r^n cannot be more than 1 because it will make Time complexity negative

$= 1 + n\left[\frac{\left(\frac{1}{2}\right)^0\left(1 - \left(\frac{1}{2}\right)^{\log_2 n}\right)}{1 - 1/2}\right]$ → will eventually be 0 because 0.5^n

$= 1 + n\left[\frac{1}{\frac{1}{2}}\right]$ $r < 1$ ∴ decreasing GP series

$= 1 + 2n$

Time complexity = $\Theta(n)$

(7) $T(n) = \begin{cases} 1 & n = 1 \\ 7 \cdot T\left(\frac{n}{2}\right) + n^2 & n > 1 \end{cases}$

$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + n^2$

$= 7 \cdot \left(7 \cdot T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)^2\right) + n^2$

$= 7 \cdot \left[7 \cdot \left(7 \cdot T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)^2\right) + \left(\frac{n}{2}\right)^2\right] + n^2$

$= 7^3 T\left(\frac{n}{2^3}\right) + 7^2\left(\frac{n}{2^2}\right)^2 + 7^1\left(\frac{n}{2^1}\right)^2 + 7^0\left(\frac{n}{2^0}\right)^2$

$= 7^k T\left(\frac{n}{2^k}\right) + 7^{k-1}\left(\frac{n}{2^{k-1}}\right)^2 + 7^{k-2}\left(\frac{n}{2^{k-2}}\right)^2 \cdots + 7^{k-k}\left(\frac{n}{2^{k-k}}\right)^2$

$= 7^k T\left(\frac{n}{2^k}\right) + n^2\left[\left(\frac{7}{2^2}\right)^{k-1} + \left(\frac{7}{2^2}\right)^{k-2} + \left(\frac{7}{2^2}\right)^{k-3} \cdots + \left(\frac{7}{2}\right)^0\right]$

$\frac{n}{2^k} = 1$

∴ $k = \log_2 n$

sustitute

$= 7^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + n^2\left[\left(\frac{7}{4}\right)^{\log_2 n - 1} + \left(\frac{7}{4}\right)^{\log_2 n - 2} \cdots + \left(\frac{7}{4}\right)^0\right]$

$= n^{\log_2 7} + n^2\left[\left(\frac{7}{4}\right)^{\log_2 n - 1} + \left(\frac{7}{4}\right)^{\log_2 n - 2} \cdots\right]$

$= n^{\log_2 7} + n^2\left[\left(\frac{7}{4}\right)^0 + \left(\frac{7}{4}\right)^1 + \left(\frac{7}{4}\right)^2 + \left(\frac{7}{4}\right)^3 \cdots + \left(\frac{7}{4}\right)^{\log_2 n}\right]$

$= n^{\log_2 7} + n^2\left[\frac{1\left(\left(\frac{7}{4}\right)^{\log n} - 1\right)}{\frac{7}{4} - 1}\right]$ (r) for $r > 1$ $\frac{a(r^n - 1)}{r - 1}$

$= n^{\cdots} + n^2 \left[\frac{1\left(\left(\frac{7}{4}\right)^{\log n} - 1\right)}{7/4 - 1} \right]$ $\frac{a(r^n-1)}{r-1}$

$= n^{\log_2 7} + n^2 \left[\frac{\left(\frac{7}{4}\right)^{\log n} - 1}{3/2} \right]$

$= n^{\log_2 7} + n^2 \left[\frac{7^{\log n}}{4^{\log n}} \right]$

$= n^{\log_2 7} + n^2 \frac{n^{\log 7}}{n^{\log_2 4}}$

$= n^{\log_2 7} + \frac{n^2 \, n^{\log 7}}{n^2}$

$= 2 n^{\log_2 7} \approx 2 n^{\log_2 2^3}$

$= 2n^3$

Time complexity = $O(n^3)$

(8) $\Rightarrow T(n) = \begin{cases} 2 & n = 2 \\ 2T(n/2) + n \log n & \text{if } n > 2 \end{cases}$

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

$= 2\left[2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2} \log \frac{n}{2}\right] + n \log n$

$= 2\left\{2 \cdot \left[2 \cdot T\left(\frac{n}{8}\right) + \frac{n}{4} \log \frac{n}{4}\right] + \frac{n}{2} \log \frac{n}{2}\right\} + n \log n$

$= 2^3 T\left(\frac{n}{8}\right) + 4 \cdot \frac{n}{4} \log \frac{n}{4} + 2 \cdot \frac{n}{2} \log \frac{n}{2} + n \log n$

$= 2^3 T\left(\frac{n}{8}\right) + n \log \frac{n}{4} + n \log \frac{n}{2} + n \log n$

$= 2^k T\left(\frac{n}{2^k}\right) + n\left[\log \frac{n}{2^0} + \log \frac{n}{2^1} + \log \frac{n}{2^2} \cdots + \log \frac{n}{2^{k-1}}\right]$

$\frac{n}{2^k} = 2$

$\log n = (k+1) \log 2$

$k = (\log_2 n) - 1$

substitute;

$= 2^{(\log_2 n)-1} \times 2 + n\left[\log \frac{n}{1} + \log \frac{n}{2} + \log \frac{n}{4} \cdots + \log \frac{n}{2^{\log n - 2}}\right]$

$= n^1 + n\left[\log n - \log 2^0 + \log n - \log 2^1 \cdots\right]$

$= n + n\left[(\log n - 1) \cdot \log n - (0 + 1 + 2 + 3 \cdots + (\log n - 2))\right]$

$= n + n\left[(\log n)^2 - \log n - \left[\frac{(\log n - 1)(\log n - 2)}{2}\right]\right.$

$$= n + n\left[(\log n)^2 - \log n - \frac{(\log n - 1)(\log n - 2)}{2}\right]$$

$$= n + n(\log n)^2 - \log n - \frac{(\log n)^2}{2} - \frac{3\log n}{2} + 1$$

Time complexity = $O(n(\log n)^2)$

(9) $\Rightarrow T(n) = \begin{cases} 2 & n = 2 \\ \sqrt{n} \cdot T(\sqrt{n}) + n & n > 2 \end{cases}$

$$T(n) = n^{1/2} T(n^{1/2}) + n$$

$$= n^{1/2}\left(n^{1/2^2} T(n^{1/4}) + n^{1/2}\right) + n$$

$$= n^{1/2}\left(n^{1/2^2}\left(n^{1/2^3} T(n^{1/8}) + n^{1/4}\right) + n^{1/2}\right) + n$$

$$= n^{1/2+1/4+1/8} T(n^{1/2^3}) + n^{1/2+1/4+1/4} + n^{1/2+1/2} + n$$

$$= n^{\frac{7}{8}} T(n^{\frac{1}{2^3}}) + n + n + n$$

$$= n^{\frac{2^K - 1}{2^K}} T(n^{\frac{1}{2^K}}) + Kn$$

$n^{\frac{1}{2^K}} = 2$

$\frac{1}{2^K}\log n = \log 2$

$\log_2 n = 2^K$

$\log_2 \log_2 n = K$

$$= n^{\frac{2^{\log_2 \log_2 n} - 1}{2^{\log_2 \log_2 n}}} \cdot 2 + n\log_2 \log_2 n$$

$$= n^{\frac{2^{\log_2 \log_2 n}}{2^{\log_2 \log_2 n}} - \frac{1}{2^{\log_2 \log_2 n}}} + n\log_2 \log_2 n$$

$$= \frac{n^1}{n^{1/2^{\log_2 \log_2 n}}} + n\log_2 \log_2 n$$

Assumption →

$$= \frac{n}{2} + n\log_2 \log_2 n$$

Time complexity = $O(n \log_2 \log_2 n)$

(10) $T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1) + n & n > 1 \end{cases}$

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

$$= 2\left(2 \cdot T(n-2) + (n-1)\right) + n$$

$$= 2\left(2 \cdot (2T(n-3) + (n-2)) + (n-1)\right) + n$$

$$= 2^3 T(n-3) + 2^2(n-2) + 2(n-1) + 2^0 n$$

$$= 2^K T(n-K) + \left[2^{K-1}(n-(K-1)) + 2^{K-2}(n-(K-2)) \cdots + 2^0 n\right]$$

$n-k = 1$

$k = n-1$

$= 2^{n-1} T(n-n+1) + [2^{n-2}(n-n+2) + 2^{n-3}(n-n+3) \ldots]$

$= 2^{n-1}.1 + [2^{n-2}.2 + 2^{n-3}.3 + 2^{n-4}.4 \ldots$

$= 2^0(n-0) + 2^1(n-1) + 2^2(n-2) \ldots + 2^{n-2}.2 + 2^{n-1}+1$

Arithmatico Geometric progression

We need to convert this in G.P.

Procedure.

① Find r from G.P. series

$r^0 + r^1 + r^2 \ldots$

$\therefore r = 2$

② $T(n) - r.(Tn)$

$T(n) = 2^0(n-0) + 2^1(n-1) + 2^2(n-2) + 2^3(n-3) \ldots$

$2.T(n) = \quad 0 + 2^1(n-0) + 2^2(n-1) + 2^3(n-2) \ldots$

$-T(n) = n - 2^1(1) + 2^2(1) + 2^3(1) \ldots$

$-T(n) = n - [2^1 + 2^2 + 2^3 \ldots 2^n]$ (a, r, n)

$-T(n) = n - \left[\frac{2(2^n - 1)}{2-1}\right]$

$-T(n) = n - \left[\frac{2^{n+1} - 2}{1}\right]$

$T(n) = -n + 2^{n+1} + 2$

Time complexity $= O(2^n)$

$T(n) = 2T(n/2 + C) + k.n$ ⌈⌉ ciel

$= O(n \log n)$ ⌊⌋ floor

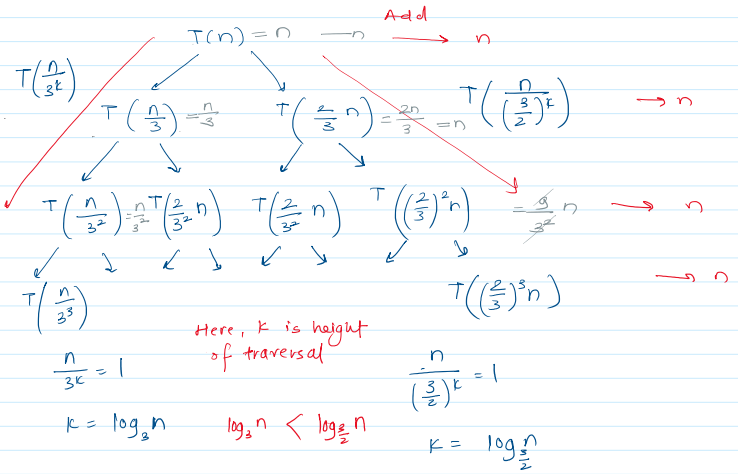
Recursive Tree Method

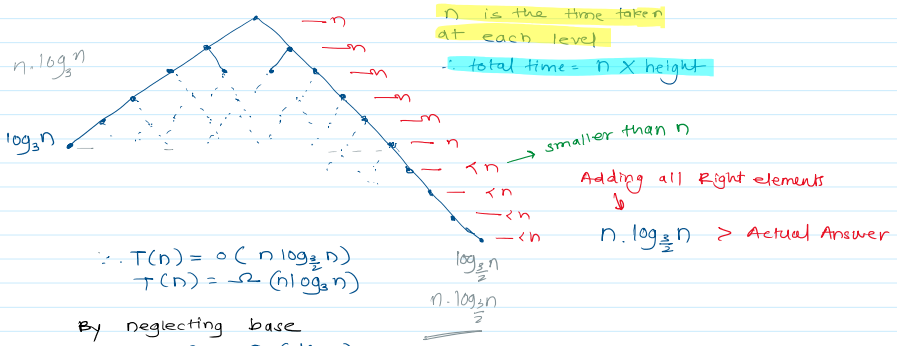
16 March 2021 11:36 PM

- Recursive Tree method

When recursion is more than once in a same function use recursive tree method.

① $T(n) = T(n/3) + T(\frac{2n}{3}) + n$



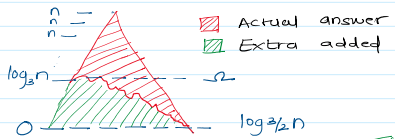


$\therefore T(n) = O(n \log_{3/2} n)$

$T(n) = \Omega(n \log_3 n)$

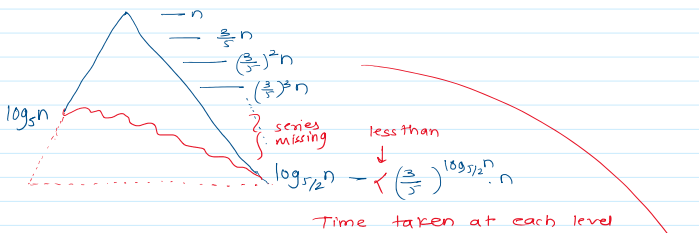
By neglecting base

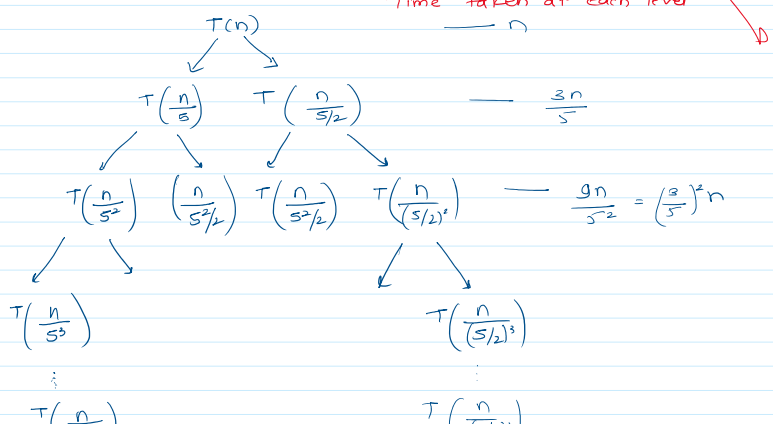
$T(n) = \Theta(n \log n)$



② $T(n) = T(\frac{n}{5}) + T(\frac{2n}{5}) + n$

doesn't add to 1
∴ will generate series





$$T(n) \le \left(\frac{3}{5}\right)^0 n + \left(\frac{3}{5}\right)^1 n + \left(\frac{3}{5}\right)^2 n + \left(\frac{3}{5}\right)^3 n \cdots + \left(\frac{3}{5}\right)^{\log_{5/2} n} \cdot n$$

$$\le \frac{n\left(1 - \left(\frac{3}{5}\right)^{\log_{5/2} n}\right)}{1 - \frac{3}{5}}$$

$$\le \frac{n \cdot \left(1 - \left(\frac{3}{5}\right)^{\log_{5/2} n}\right)}{\frac{2}{5}} \quad \text{constants}$$

$$\le O(n)$$

If we take series up-to $\left(\frac{3}{5}\right)^{\log_5 n} \cdot n$

Answer = $O(n)$

(5^5) $\quad\quad$ $((5/2)^5)$

$\vdots$ $\quad\quad$ $\vdots$

$T\left(\frac{n}{5^k}\right)$ $\quad\quad$ $T\left(\frac{n}{(5/2)^k}\right)$

$\frac{n}{5^k} = 1$ $\quad\quad$ $\frac{n}{(5/2)^k} = 1$

$\log n = k \log 5$ $\quad\quad$ $\log n = k \log 5/2$

$\therefore k = \log_5 n$ $\quad\quad$ $\therefore k = \log_{5/2} n$

If we take series upto $\left(\frac{3}{5}\right)^{\log_5 n} \cdot n$

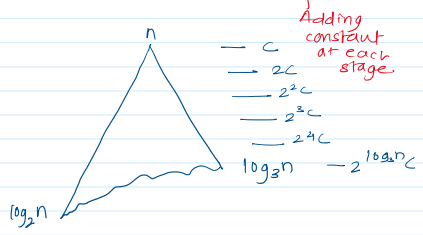
Answer = $O(n)$

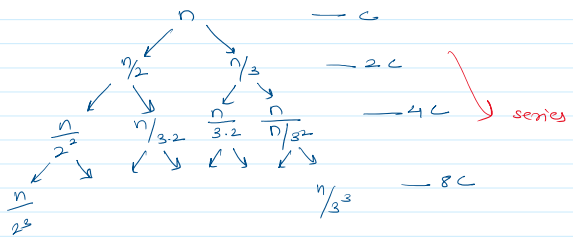
still asymptotically equal

Time complexity = $\Theta(n)$

③ $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{3}\right) + c$

Adding constant at each stage.





Time complexity = $O(n)$

$= \Omega(n^{0.6})$

$T(n) \leq 2^0c + 2^1c + 2^2c + 2^3c \cdots 2^{\log_3 n}c$

$\leq \left(2^0 + 2^1 + 2^2 \cdots 2^{\log_3 n}\right) c$

$\leq \left[2^0\left(\frac{2^{\log_3 n} - 1}{2 - 1}\right)\right] \cdot c = n^{\log_3 2} \cdot c$ (×1)

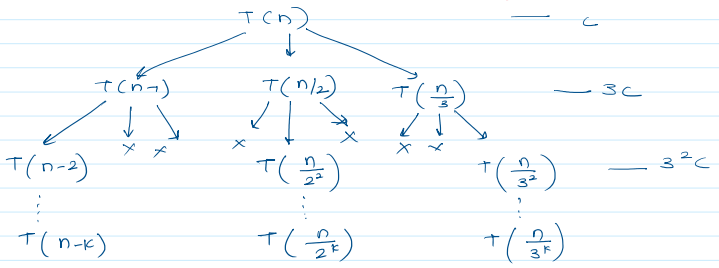
$= O(n^{0.6})$

If series upto $2^{\log_2 n} \cdot c$

$T(n) = O(n)$

④ $T(n) = T(n-1) + T\left(\frac{n}{2}\right) + T\left(\frac{n}{3}\right) + c$

Time taken at each level



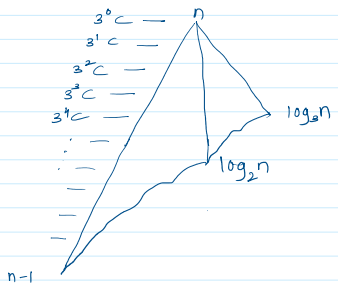
c

3c

3^2c

$n - k = 1$ $\quad\quad$ $k = \log_2 n$ $\quad\quad$ $k = \log_3 n$

$\therefore k = n - 1$



$T(n) \leq 3^0c + 3^1c + 3^2c \cdots 3^{n-1} \cdot c$

$\leq \left[\frac{(3)^{n-1} - 1}{2}\right] \cdot c$

$= O(3^n)$

upto $3^{\log_3 n} \cdot c$

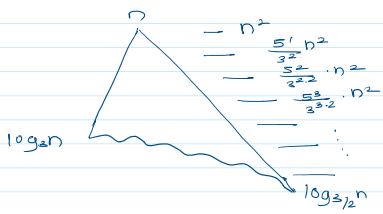
$T(n) = \left[\frac{3^{\log_3 n} - 1}{2}\right] \cdot c$

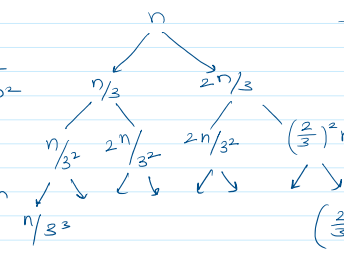
$= n \cdot c$

$\therefore$ Time complexity = $O(3^n)$

$= \Omega(n)$

⑤ $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n^2$





$T(n) = n^2\left[1 + \frac{5}{3^2} + \frac{5^2}{3^{2.2}} + \frac{5^3}{3^{3.2}} \cdots \frac{5^{\log_{3/2} n}}{3^{2.\log_{3/2} n}}\right]$

$\cdots \left[\cdots \left(\frac{5}{\cdots}\right)^2 \left(\frac{5}{\cdots}\right)^3 \cdots \left(\frac{\cdots}{\cdots}\right)^{\log_{3/2} n}\right]$

$$T(n) = n^2\left[1 + \frac{5}{3^2} + \frac{5^2}{3^{2\cdot 2}} + \frac{5^3}{3^{3\cdot 2}} \cdots \frac{5^{\log_{3/2} n}}{3^{2\cdot \log_{3/2} n}}\right]$$

$$= n^2\left[1 + \left(\frac{5}{3^2}\right)^1 + \left(\frac{5}{3^2}\right)^2 + \left(\frac{5}{3}\right)^3 \cdots + \left(\frac{5}{3^2}\right)^{\log_{3/2} n}\right]$$

$$= n^2\left[\frac{1.\left(1 - (5/9)^{\log_{3/2} n}\right)}{1 - 5/9}\right]$$

$$= O(n^2)$$

Time complexity = $O(n^2)$

Masters Theorem

16 March 2021 11:39 PM

Master theorem

Master theory only applies to this format

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

Where a and b are constants and $a \geq 1$, $b > 1$, $f(n)$ is ~~positive function~~

Case – 1

Bigger

If $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$

Where ε is constant & $\varepsilon > 0$

then,

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

case 2

smaller

If $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$

then,

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

case – 3

If $f(n) = \Theta\left(n^{\log_b a} (\log n)^k\right)$

where k is constant & $k \geq 0$

then,

$$T(n) = \Theta\left(n^{\log_b a} (\log n)^{k+1}\right)$$

① $T(n) = 8T(n/2) + n^2$ — If negative master theorem fails

$a = 8$, $b = 2$, $f(n) = n^2$

$$n^{\log_b a - \varepsilon} = n^{\log_2 8 - \varepsilon} = n^{3\log_2 2 - \varepsilon} = n^{3-\varepsilon}$$

comparing

$f(n) = n^{3-\varepsilon}$

$n^2 = n^{3-\varepsilon}$

$\therefore \varepsilon = 1$ .... case – 1

$\therefore f(n) < n^{\log_b a - \varepsilon}$

∴ Time complexity = $\Theta(n^3)$

② $T(n) = 2T(n/2) + n^2$

$a = b = 2$ $\quad f(n) = n^2$

$f(n) = n^{\log_b a - \varepsilon}$

$n^2 = n^{\log_2 2 - \varepsilon}$

$n^2 = n^{1-\varepsilon}$

$\varepsilon = -1$ ... case 2

$f(n) > n^{\log_b a - \varepsilon}$

∴ Time complexity = $\Theta(n^2)$

③ $T(n) = T(n/2) + c$

$a = 1$ $\quad f(n) = c$

$b = 2$

comparing

$f(n) = n^{\log_b a - \varepsilon}$

$c = n^{\log_2 1 - \varepsilon}$

$c = n^{0 - \varepsilon}$

$c = 1$

$\therefore \varepsilon = 0$

∴ Time complexity = $O(n^{\log_2 1} \cdot (\log n)^{0+1})$

$= O(\log n)$

④ $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$

$a = b = 2$ $\quad f(n) = n \log n$

$f(n) = n^{\log_b a - \varepsilon}$

$n \log n = n^{\log_2 2 - \varepsilon}$

$n \cdot \log n = n^{1-\varepsilon}$

Master theorem will not work as difference should be polynomial times not logarithmic

ε can't be find

But, by applying case 3

$n \log n = n^1 (\log n)^k$

satisfies case - III

∴ Time complexity = $O(n(\log n)^2)$

∴ Time complexity = $O(n(\log n)^2)$

⑤ $T(n) = 8T(n/2) + n^4 \log n$

$a = 8$ $\quad f(n) = n^4 \log n$

$b = 2$

$f(n) = n^{\log_b a - \varepsilon}$

$n^4 \log n = n^{\log_2 2^3 - \varepsilon}$

$n^4 \log n = n^{3-\varepsilon}$

$\varepsilon = -1$ .... case 2

polynomial ↓

Here case II is satisfied & LHS is greater by $n \log n$

∴ Time complexity = $\Theta(n^4 \log n)$

⑥ $T(n) = 2^n T\left(\frac{n}{2}\right) + n^2$

$a = 2^n$ $\quad f(n) = n^2$

$b = 2$

$n^2 = n^{\log_2 2^n - \varepsilon}$

$n^2 = n^{n-\varepsilon}$

$\varepsilon = (n-2)$ case-1

$n^2 < n^n$

Time complexity = $\Theta(n^n)$

But $a \neq$ constant

∴ Master theorem can't gaurantee the answer to be right

7 $T(n) = T(\sqrt{n}) + c$

Master theorem for examples with root

① Assume $n = 2^k$

$\therefore T(2^k) = T(2^{k/2}) + c$

② Assume $T(2^k) = S(k)$

$S(k) = S(k/2) + c$

Solve by master theorem

$a = 1$ $\quad f(k) = c$

$b = 2$

$c = k^{\log_2 1 - \epsilon}$

$c = 1$ .... case-III

$\therefore \quad S(k) = O(1 \cdot (\log k)^{p+1})$

Resubstitute

$T(2^k) = O(\log k)$

$T(n) = O(\log \log_2 n)$

8. $T(n) = T(\sqrt{n}) + \log_2 n$

① Assume

$n = 2^k$

$T(2^k) = T(2^{k/2}) + \log_2 2^k$

② Assume $T(2^k) = S(k)$

$S(k) = S(k/2) + \log_2 k$

$f(k) = \log_2 k$

$a = 1$

$b = 2$

$\log_2 k = k^{\log_2 1}$

$\log_2 k = \frac{1}{k^{\epsilon}}$

case I & II fails.

By case 3

$\log_2 k = k^{\log_2 1} (\log k)^p$

$\log_2 k = (\log k)^1$

case-III satisfied

Time complexity $= \Theta((\log k)^1)$

Resubstitution $T(2^k) = S(k)$

$= \Theta((\log 2^k))$

$T(2^k) = \Theta(\log 2^k)$

Resubstitute $2^k = n$

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

⑨ $T(n) = 2T(n/2) + \frac{n}{\log n}$

$a = 2 \qquad f(n) = \frac{n}{}$

$a = 2$
$b = 2$

$f(n) = \frac{n}{\log n}$

$$\frac{n}{\log n} = n^{\log_2 2 - \epsilon}$$

$$\frac{n}{\log n} = n^{1-\epsilon}$$

case - I & case - II not possible

∴ By case 3.

$$\frac{n}{\log n} = n \cdot (\log n)^k$$

$\therefore k = -1$

But $k \geq 1$ ∴ case - III fails.

∴ Master theorem not possible

8Time Complexity of Recursive Programs

16 March 2021 11:41 PM

- Time complexity of Recursive programs

①
```
fact(n) {
    if (n <= 1) return 1;
    return (n * fact(n-1));
}
```
} will take constant time

→

① Recurrence relation for time

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

solve by substitution.

$= T(n-2) + c + c$

$= T(n-3) + c + c + c$

$= T(n-k) + kc$

$n - k = 1$

$\therefore k = n-1$

Here k will be the space complexity. i.e. number of times function called

$= T(n-n+1) + (n-1) \cdot c$

$= T(1) + (n-1) \cdot c$

$= 1 + (n-1) \cdot c$

$\therefore$ Time complexity $= O(n)$

Recurrence relation for value

$= n \cdot T(n-1)$

② fibonacci series :

Addition of prev two terms.

i.e. $fib(n) = fib(n-1) + fib(n-2)$

```
fib(n) {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    return fib(n-1) + fib(n-2)
}
```

Recurrence relation for value

$T(n) = T(n-1) + T(n-2)$ ... $n > 1$

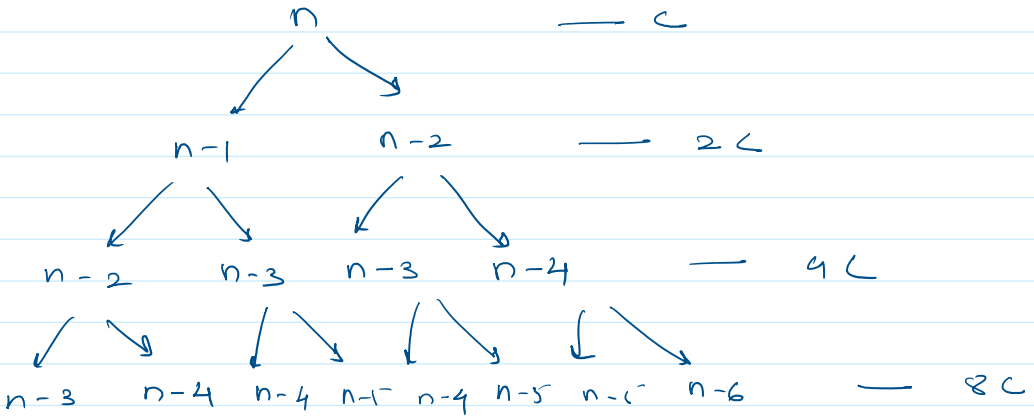
$= n$ ... $n = 1 \;||\; n = 0$

Recurrence relation for time

$T(n) = T(n-1) + T(n-2) + c$ ... $n > 1$

$= c$ ... $n = 1 \;||\; n = 0$

By tree method



$n-K$

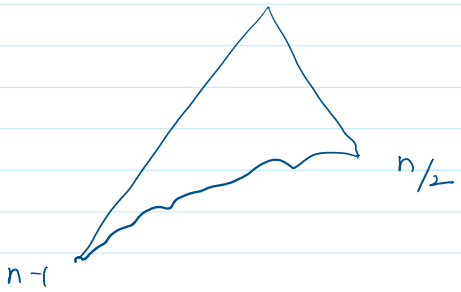
$n-2K$

$n-k=1$

$k=n-1$

$n-2k=0$

$k=n/2$



$$\text{Series} = c[2^0 + 2^1 + 2^2 \cdots 2^k]$$
$$= c[2^0 + 2^1 + 2^2 \cdots 2^k]$$

GP series.

$$= c\left[\frac{1(2^k - 1)}{2-1}\right]$$

$k = n-1$

$$= c[2^{n-1} - 1]$$

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

If $k = n/2$

$$= c[(2^{n/2}) - 1]$$
$$= c((\sqrt{2})^n$$

(3)

```
A(n) {
  if (n <= 1)
    return (n^2 + n + 10)        } c
  else
    for(i = 1; i <= n^2; i++)    } n^2
      x = y + z;
  return (A(n/2). A(n/2) .n^5)   } 2 . A(n/2)
}
```

Reccurrence relation for time.

$$T(n) = c + n^2 + 2A(n/2)$$

Reccurrence relation for time

$T(n) = c + n^2 + 2A(n/2)$

$= 2A(n/2) + n^2$

By master's theorem

$a = b = 2 \qquad f(n) = n^2$

$\therefore n^2 = n^{\log_2 2 - \varepsilon}$

$= n^{1-\varepsilon}$

$\varepsilon = -1$

$\therefore$ case 1

$n^2 > n^1$

$\therefore$ Time complexity $= \Theta(n^2)$

(4)

```
A(n) {
  if (n ≤ 1)
      return (n^10)          } c
  else
      for (i=1 ; i ≤ 4 ; i++)
          A (n/2);           } 4A(n/2)
      for (j=1 ; j ≤ n^2 ; j++)
          x = y + z;         } n^2
      return ( 5. A(n/2) . 3. A(n/2) + LS(n) );   } 2A(n/2) + n
}
```

recursion is only twice

Just constants, don't consider

```
LS(n) {
  for( i=0 ; i<n ; i++)      } n
       x = y + z;
}
```

$\therefore$ Reccurence relation for time

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

$= 6T(n/2) + n^2 + n$

By master's theorem

$a = 6 \qquad f(n) = n^2 + n$

$b = 2$

$n^2 + n = n^{\log_2 6 - \varepsilon}$

dominant $\rightarrow n^2 = n^{2.5 - \varepsilon}$

$\varepsilon = 0.5$

case 2.

$\therefore$ Time complexity $= O(n^3)$

(5) Program for finding GCD

GCD (0, n) || GCD (n, 0)

$= n$

```
GCD (n, m) {
    if (n == 0 || m == 0) && (n != m)
        return m+n;
    return GCD (m % n , n)
}
```

c

$\log_2 n$ if $n \geq m$

Recurrence relation $\approx T(n/2) + c$

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

Divide and conquer

02 March 2021 21:34

Min - 2 Marks
Max - 4 Marks
Mostly Time complexity.

Basics Needed:

1. Recursion
2. Recurrence relation
3. Recurrence relation solving.

- Divide and conquer
  1. Divide the problem into some sub problems
  2. Conquer the subproblem by calling recursively until we get subproblem solution
  3. Combine subproblems solutions to get final solution

- Algorithm

```
DAC ( n, i, j ) {
    if ( small ( n, i, j ) )
        return solution (n, i, j) ;
    else
        k = Divide ( n, i, j)
        S1 = DAC ( n, i, k)
        S2 = DAC ( n, k+1, j )
        S = Combine (S1, S2)
        return S;
}
```

Time complexity = $\begin{cases} c & \text{if } n \text{ is small} \\ f_1(n) + 2T(n/2) + f_1(n) & \text{if } n \text{ is big.} \end{cases}$

$$T(n) = 2T\left(\frac{n}{2}\right) + f(n)$$

∴ In general,

$$= aT\left(\frac{n}{b}\right) + f(n)$$

a = Number of subproblems
b = Partions of problem
f(n) = Time require for dividing and combining

For solving divide and conquer problems masters theorem is used

Applications:

1. Finding maximum and minimum
2. Binary search
3. Power of an element
4. Merge sort
5. Quick sort
6. Selection procedure
7. Finding inversions
8. Strassen's matrix multiplication
9. Continuous Maximum Sub-array sum

1. Finding Max and Min

19 March 2021 08:25

① Finding maximum and minimum

- Input : Array of 'n' elements.
- Output : Find maximum and minimum element

- Steps :
  ① Find lowest fragment of problem
  n = 1 — 0 comparisons
  n = 2 — 1 comparison

Solution by
- straight min and max

```
foo( a[], n ) {
    max = min = a[0]
    for ( i=0; i<n; i++){
        if ( a[i] > max)
            max = a[i] ;
        else if ( a[i] < min)
            min = a[i] ;
    }
    return min, max;
}
```

Best case = n-1 comparisons
Worst case = 2(n-1) comparisons
Average = $\frac{3}{2}(n-1)$

$T(n) = n$

- By divide and conquer.

Function Execution order : Post-order
Function calling order : pre-order (a[i], a[j])

```
foo( a[], i, j) {
    if ( i==j)
        temp.min = temp.max = a[i];
    else if ( j-i == 1) {
        if (a[i] > a[j]) {
            temp.max = a[i];
            temp.min = a[j]; }
        else {
            temp.min = a[i] ;
            temp.max = a[j] ;
        }
    }
    else {
        k = (i+j)/2
```

small solution conquer

```
else {
    k = (i+j)/2
    ans1 = foo(a[], i, k)
    ans2 = foo(a[], k+1, j)
    if (ans1.max > ans2.max)
        temp.max = ans1.max
    else
        temp.max = ans2.max
    if (ans1.min < ans2.min)
        temp.min = ans1.min
    else
        temp.min = ans2.min
}
return temp;
}
```

divide

combine

Height of tree will be the space complexity for given program

$$\text{Time complexity} = \begin{cases} c & \text{if } n \le 2 \\ c + T(n/2) + T(n/2) + c & \text{else} \end{cases}$$

$$T(n) = 2T(n/2) + c'$$

$$a = b = 2 \qquad f(n) = c'$$

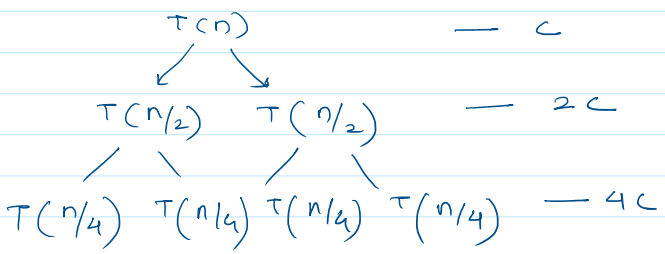
$$c' = n^{\log_2 2 - \varepsilon}$$

$$n^0 c' = n^{1-\varepsilon}$$

$$\varepsilon = 1 \quad \therefore \text{case-1}$$

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

By recursive tree method



$$\frac{n}{2^k} = 2$$

$$n = 2^{k+1}$$

$$k+1 = \log_2 n$$

$$\therefore k = (\log_2 n) - 1$$ ← Height of tree aka. space complexity

$$T(n) = c + 2c + 2^2c + 2^3c \cdots$$
$$= c\left[1 + 2 + 2^2 \cdots 2^{\log_2 n - 1}\right]$$

G.P. series

$$= c\left[1\left(\frac{2^{(\log_2 n)-1} - 1}{2-1}\right)\right]$$ (neglect the -1)

$$= c\left[n - 1\right]$$

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

$T(n)$ = number of comparisons betw elements of given array

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 2 \\ 2T(n/2) + 2 & \text{else} \end{cases}$$

$$T(n) = 2T(n/2) + 2$$
$$= 2.(2.T(n/2^2) + 2) + 2$$
$$= 2.(2.(2.T(n/2^3) + 2) + 2) + 2$$
$$= 2^3 T(n/2^3) + 2^3 + 2^2 + 2^1$$
$$= 2^k T(n/2^k) + 2^k + 2^{k-1} + 2^{k-2}$$

$\therefore \frac{n}{2^k} = 2$ mostly $k = \log_2 n - 1$

$\frac{n}{2^k} = 1$ This value applicable only if array is provided with single element $\therefore k = \log_2 n$

$$= 2^{\log_2 n - 1} T(2) + \left[2 + 2^2 + 2^3 \cdots + 2^{\log_2 n - 1}\right] \quad \cdots$$

G.P. series

$$= \frac{n}{2} \cdot 1 + \left[2\left(\frac{2^{\log_2 n - 1} - 1}{2-1}\right)\right]$$

$$= \frac{n}{2} + 2\left(\frac{2^{\log_2 n}}{2} - \frac{2}{2}\right)$$

$$= \frac{n}{2} + n - 2$$

$$= \frac{3n}{2} - 2 \Rightarrow \text{comparisons}$$

Conclusion:

No. of comparisons in straight max-min =
Best : n
Worst : $2n$
Avg : $\frac{3}{2}n$

DAC Algo =
Best = $3/2\,n - 2$
Worst = $3/2\,n - 2$

DAC Algo = Best = $3/2n - 2$
Worst = $3/2n - 2$
Avg = $3/2n - 2$

Q. To find max-min elements in 150 elements array best case how many comparisons needed

→ For best case straight max-min is better n comparisons.
= 150 comparisons.

Q How much time will it take to find neither first maximum nor first min in max-min algo in n distinct elements

→ n - distinct elements
We shouldn't return 1st max & min from array.

Take any three elements from array
sort them — 3 comparisons

x (y) z sorted

x < y z > y

y will be neither min nor max
In Best / Worst case will take 3 comparisons

∴ $T(n) = \Theta(1)$

If neither second min nor second max.
Just take 5 elements and sort them

$a < b < (c) < d < e$

∴ For neither k^{th} max nor k^{th} min
elements = $2k+1$
comparisons = $2k+1$

2. Binary Search

19 March 2021 15:15

- Linear search

Input: Array of n elements, and a element 'x'
Output: location of the element x

```
search ( a[] , x ) {
    for (i=0 ; i <n ; i++) {
        if a[i] ==x
            return i ;
    }
    return -1;
}
```

Best case = c = $\theta(1)$
Worst case = n = $\theta(n)$
Average case = $\frac{1+2+3 \ldots +n}{n}$ ... Average depends on all elements

$= \frac{n.(n+1)}{2n}$

$= \frac{n+1}{2} = \theta(n)$

$T(n) = \Omega(1) = O(n)$

Imp

- Binary search — (partial Divide nd conquer application as no combine)

Input = sorted array of n elements, element x
output = position of element x

```
binary_search ( a[] , x , i , j )
        mid = (i+j)/2
        if ( a[mid]== x )
            return mid
        else if ( i == j)
            return -1 ;
        elce if ( x < a[mid] )
            return ( a[] , x , i , mid-1)
        else if ( x > a[mid] )
            return ( a[] , x , mid+1 , j)
        else return -1 ;
}
```

c

only one recursion will be executed
T(n/2)

Time complexity

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

$= T\left(\frac{n}{2^k}\right) + kc$

$\therefore \quad k = \log_2 n$

$\therefore T(n) = O(\log_2 n)$

θ because for best case only 1 choice

Best case = $\theta(1)$

Worst case = $\theta(\log n)$

$\therefore T(n) = \Omega(1) = O(\log n)$

space complexity = $\Omega(1) = O(\log n)$

Recurrence relation

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + c & \text{else} \end{cases}$$

- Binary search without recursion

```
BS( a[], x) {
  while ( i ≤ j) {
     mid = (i+j)/2
     if( a[mid] == x)
          return mid;
     else if ( i == j)
          return -1;
     else if ( x < a[mid])
          j == mid-1;
     else
          i == mid +1;
  }
}
```

$T(n) = \Omega(1) = O(\log n)$

$S(n) = \theta(1)$

- Problems on binary search

(1) Input : A sorted array of n-distinct integers
output : Find any element a[i]
such that a[i] == i.

- sorted array
- no-repetation
- can be positive as well as negative

Algorithm: (Divide and conquer)

```
foo( a[] ) {
  while ( i <= j) {
```

```
foo ( a[] ) {
  while ( i <= j ) {
    mid = (i+j) /2
    if ( a[mid] == mid)
        return mid
    else if ( a [mid] > mid)
        j = mid -1
    else
        i = mid +1
  }
}
```

discard

-2	1	3	4	5	10	12	13
0	1	2	3	4	5	6	7

mid

a[mid] > mid

as array is sorted you'll not get the required element at right

if you arrange distinct elements in sorted order every element will be greater than mid at right

discard

-2	0	2	3
0	1	2	3

a[mid] < mid

that suggest there is no posibility of finding required element at left

2	3
2	3

a[mid] == mid

Return 2

Time complexity using

Linear search:

Best case = $\theta(1)$

Worst case = $\theta(n)$

Average case = $\frac{n.(n+1)}{2.n} = \theta(n)$

$T(n) = \Omega(1) = O(n)$

Binary search:

Best case = $\theta(1)$

Worst case = $\theta(\log n)$

Average case = $\frac{\log n (\log n + 1)}{2 \log n} = \theta(\log n)$

$T(n) = \Omega(1) = O(\log n)$

② Input: An array of n-elements in which

② Input: An array of n-elements in which until some position x elements increasing after elements in decreasing order
Output: Best algorithm worst case.

Algorithm: Divide and conquer

```
foo( a[] )
    while ( i≤j ) {
        mid = (i+j)/2
        if (a[mid] < a[mid+1])
            i = mid + 1
        if (a[mid-1] < a[mid] > a[mid+1])
            return mid
        else
            j = mid - 1
    }
}
```

$T(n) = \Omega(1) = O(\log n)$

- distinct not mentioned need to check again

If not distinct elements use recursion go both side and in combining return −1 or x

④ Input: Array of n-elements in which until some point all are infinite and afterwards all are integers

Algorithm (Divide and Conquer)

```
foo ( a[] ) {
    while ( i≤j ) {
        mid = (i+j)/2
        if ( a[mid] == INT_MAX)
            if ( a[mid+1] != INT_MAX)
                return mid
            else
                i = mid + 1
        else if ( a[mid - 1 ] = INT_MAX )
                return mid
        else    j = mid - 1
    }
}
```

$T(n) = \Omega(1) = O(\log n)$

⑤ Input: An array of n-elements in which until some positions all are integers after all infinites (Assume array size unknown and

are integers after all infinites
(Assume array size unknown and after array all are '$')

Output: Find position of 1st infinite

Algorithm: (Divide an conquer)

```
foo( a[] )
  while ( a[i] != $) {
      if ( a[i] == ∞)
          break
      else
          i = i^2
  }
   j = i ; i = i/2
  while ( i ≤ j) {
      mid = (i+j)/2
      if ( a[mid] == ∞ && a[mid-1] != ∞ ||
           a[mid] != ∞ && a[mid+1] == ∞)
              return mid
      else if ( a[mid] == ∞ || a[mid] == $)
              j = mid - 1
      else  i = mid + 1
  }
}
```

log n

log n

$T(n) = \Omega(1) = O(\log n)$

⑥ Input: Array of n-elements

Output: Find any 2-ele a & b such that $a + b > 1000$

① Try to find solution using linear search first.
If the answer is minimum in the option select that one.

② Else try applying Binary search and if T(n) is minimun select the optim

③ Else try minimizing further.

Algorithm:

```
If ( a[n] + a[n-1] > 1000)
        return 1
else    return -1.
```

$T(n) = \Theta(1)$

⑦ Input: n-elements array
Output: Find a & b such that a+b > 1000

Algorithm:

```
foo( a[] ) {
    max = a[0]
    for( i=1; i<n; i++ ) {
        if ( max + a[i] > 1000)
            return max, a[i]
        else if ( max < a[i])
            max = a[i]
    }
}
```

$T(n) = \theta(n)$

⑧ Input: sorted array of n-elements.
Output: a+b = 1000

Algorithm:

```
foo( a[] )
  for( k=0; k< n ; k++) {
     while ( i ≤ j) {
          mid = (i+j) /2
          if (a[k] + a[mid] == 1000)
               return a[k], a[mid]
          if ( a[k] + a[mid] > 1000)
               j = mid - 1
          else  i = mid + 1
     }
```

$T(n) = \Omega(1) = O(n \log\log n)$

But, By Greedy Algorithm

```
foo( a[] ) {
     i=0 ; j = n-1;
     while ( i < j){
          if ( a[i] + a[j] == 1000)
               return a[i] , a[j]
          else if ( a[i] + a[j] < 1000)
               i++;
          else  j--;
}
```

$T(n) = \Omega(1) = O(n)$

⑨ Input: unsorted n-element array
output: a+b = 1000

Algorithm:

① sort the array = $O(n \log n)$ ... DAC

② Apply greedy algorithm

```
if ( a[i] + a[j] == 1000)
    return
else if (a[i] + a[j] < 1000)
    i++
else j--;
```

$\therefore T(n) = n \log n + n$

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

(10) Input: sorted array of n-elements

output: $a + b + c = 1000$

Algorithm:

By linear search = $O(n^3)$ — check every combination

By Binary search = $O(n^2 \log n)$ — for every a,b find c by BS

By Greedy technique = $O(n^2)$ — $a\ (\underset{n}{b} \rightarrow \cdots \leftarrow \underset{n}{c})$

$\therefore T(n) = O(n^2)$

If array not sorted

$T(n) = n^2 + n \log$ ← for sorting

$\therefore T(n) = O(n^2)$

For k-elements whose sum is 1000

By linear search = $O(n^k)$

By Binary search = $O(n^{k-1} \log n)$

By Greedy technique = $O(n^{k-1})$

For unsorted array where $k > 2$

sorting doesn't effect asymptotically

$\therefore$ Answer will be same

(11) Input: array of n-elements

output: subset whose sum is 1000

Algorithm:

① sort

② For this we have to check every subset of given array

Tn for finding powerset $= O(2^n)$

$\therefore T(n) = n \log n + 2^n$

$= O(2^n)$

NP complete time ... as it takes exponential time.

3. Power of an Element

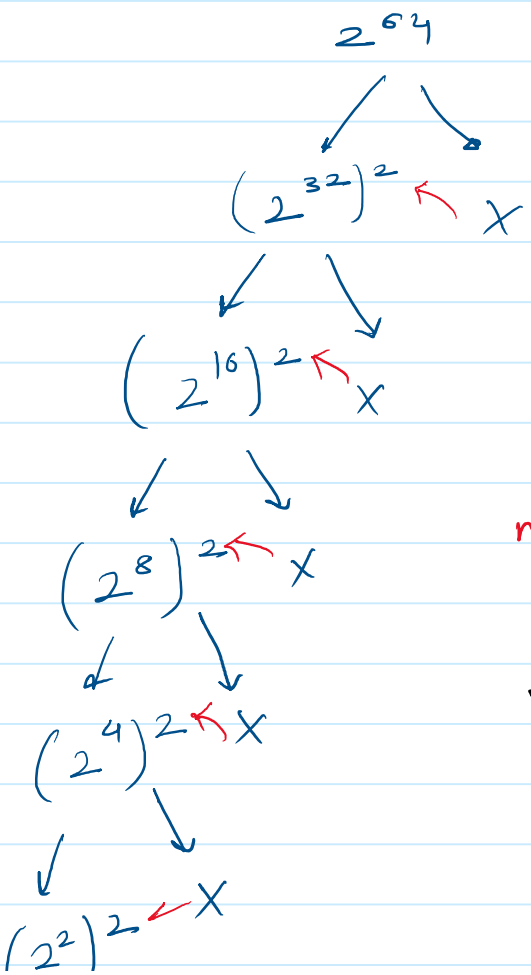
19 March 2021 08:27

Input: Two positive integers $a > 1, n \geq 1$

Output: Find a^n

```
pow ( a , n ) {
    if  a == 1  ||  n == 1
            return  a ;
    else
            result = pow(a, floor(n/2))
    if  a % 2 == 0
            return  result. result
    else
            return.  a . result. result
```

consider pow (2, 64)

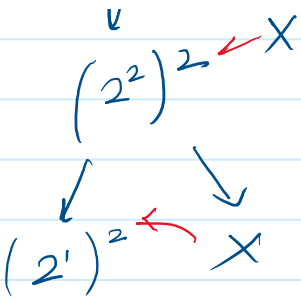


$k = \log_2 n$

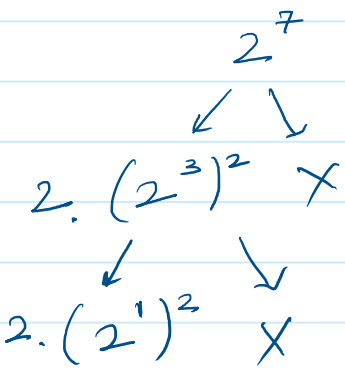
multiplications requried = $1 . \log_2 n$
i.e. 1 at each level.

$T(n) = \theta(\log_2 n)$

If n = odd



if n = odd



If n is even

Best case = $1.\log_2 n$ multiplications

If n is odd

Worst case = $2.\log_2 n$ multiplications

Recurrence relation.

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(n/2) + c & \text{else} \end{cases}$$

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

$$= T\left(\frac{n}{4}\right) + c + c$$

$$= T\left(\frac{n}{2^3}\right) + 3c$$

$$= T\left(\frac{n}{2^k}\right) + kc$$

$\therefore k = \log_2 n$

$$= T(1) + c.\log_2 n$$

$$= c.\log_2 n + 1$$

$\therefore T(n) = \Theta(\log_2 n)$

Number of multiplications

$$= \begin{cases} 0 & \text{if } n == 1 \\ T(n/2) + 1 & \text{if even} \\ T(n/2) + 2 & \text{if odd} \end{cases}$$

$$T(n) = T(n/2) + 1$$
$$= T\left(\frac{n}{2^k}\right) + k$$

$$k = \log_2 n$$

$$= T(1) + \log_2 n$$

Non-recursive with $T(n) = \Theta(\log n)$

```
pow(a) {
    result = a
    for(i=1; i<=n; i=2.i) {
        result = result.result
    return result
}
```

4. Merge Sort

21 March 2021 19:04

IMP

Merging two sorted sub-array is Merge - sort

Input : Unsorted array.
Output : sorted array.

- Merge Algorithm (outplace) — Using non-constant space other than given array.

outplace Algorithm i.e. taking extra memory other than program

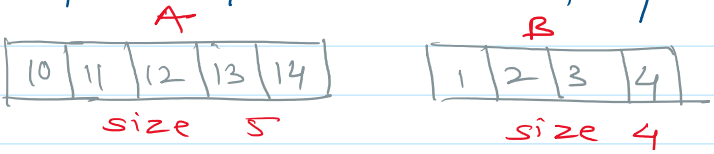
```
merge( a[], b[]) {
  while ( i < a.size() && j < b.size()) {
    if ( a[i] < b[j])
        result [k++] = a[i++]
    else
        result [k++] = b[j++]
  }
  while ( i = a.size() && j < b.size() ) { copy remaining from b
      result [k++] = b[j++]
  while( j = b.size() && i < a.size()) { from a
      result [k++] = a [i++]
  }

    i = k = 0
    while ( k < result.size())
        a[i++] = result [k++]   } To sort the array itself copy result back to a

}
```

Comparisons required

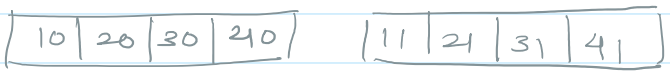
Comparisons only required until any array is traversed fully.



size 5 size 4

only 4 comparisons needed here because B array will be traversed and only A array remained just copy remaining element (no-comparison)

In worst case



m+n comparisons needed

For comparison time complexity

$\therefore T(n) = \Omega(\min(a, b))$

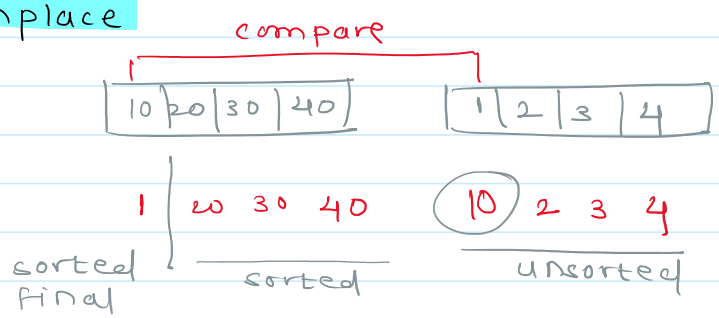
$= O(m+n)$

- Moves required

Here we need to move each element from both arrays to a new array.

$\therefore T(n) = \Theta(m+n)$

Inplace



We need to sort again by bubble sort but only one pass

$\therefore$ comparisons = n

In worst case,

$\therefore T(n) = O(m.n)$

Merge sort Algorithm

```
mergesort ( a[] , i , j ) {
    if (i==j)
        return a[i]
```

}

```
if (i==j)
    return a[i]                                  } c
mid = (i+j)/2
result1 = mergesort(a[], i, mid)      - T(n/2)
result2 = mergesort(a[], mid+1, j)    - T(n/2)
merge(result1, result2)
return                                - n
}
```

Recurrence relation for time

$$T(n) = c \quad \text{if } n = 1$$
$$= 2T(n/2) + n \quad n > 1$$

By master's theorem

$a = 2 \qquad f(n) = n$

$b = 2$

$$n = n^{\log_2 2 - \varepsilon}$$
$$n^1 = n^{1-\varepsilon}$$

$\therefore \varepsilon = 0$

$\therefore$ case - III

$$\therefore T(n) = n^{\log_2 2} (\log n)^{k+1} \quad \because k = 0$$
$$= \Theta(n \log n)$$

↑ because return is at last, so cases will not differ for best/worst

stack space = $\log(n)$

Merge space required = n

$\therefore$ space complexity = $n + \log n$
$= \theta(n)$

- Examples

① Input: $\frac{n}{\log n}$ sorted subarray with size $\log n$

output: Find single sorted array

with all elements.

Algorithm:

merge algorithm.

Time complexity of merge algorithm

$T(K) = K$

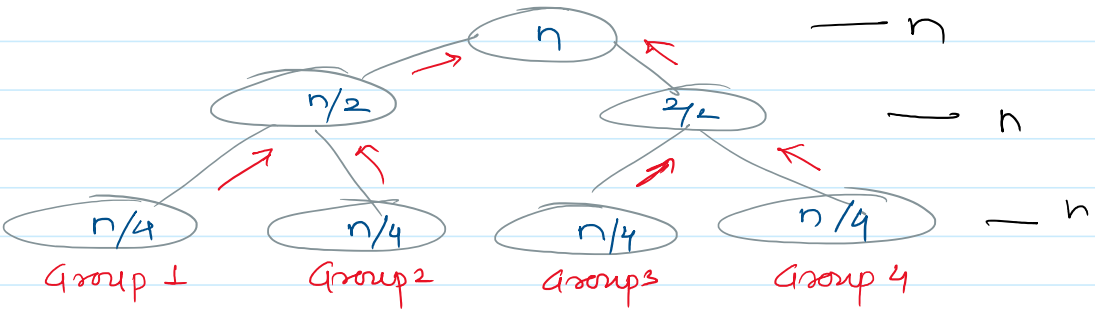
Here substitute $K = \frac{n}{\log n}$ ... size

$\therefore T\left(\frac{n}{\log n}\right) = \frac{n}{\log n}$

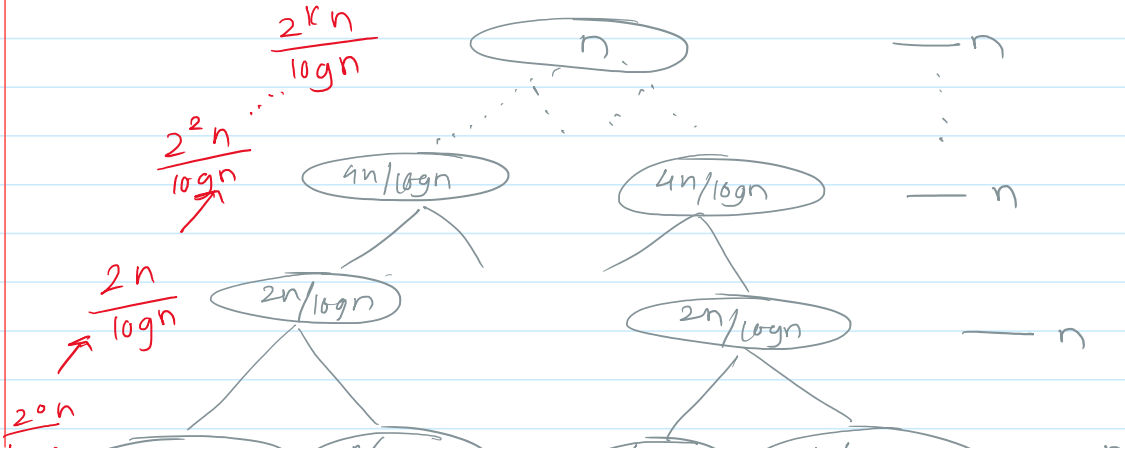
Number of elements

$\log n \cdot \frac{n}{\log n} = n$ elements

Number of times merge function called



If n-groups logn levels will be there



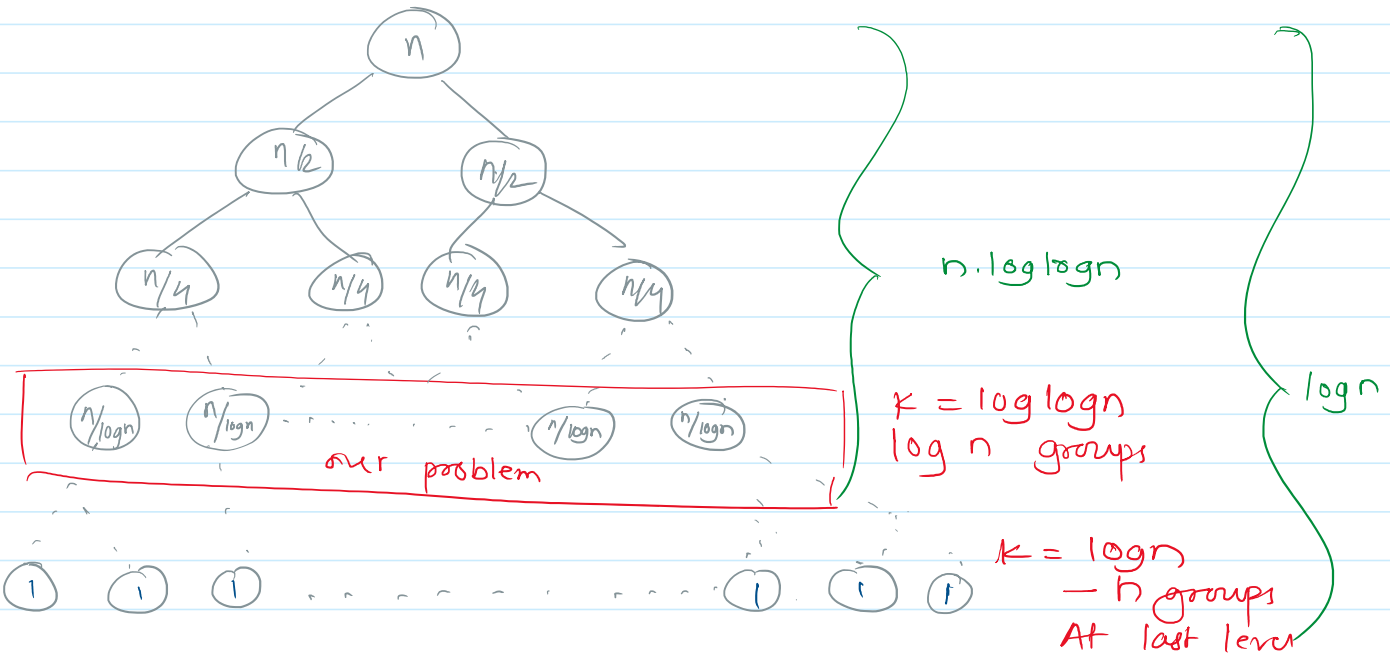
$$\therefore \frac{2^k n}{\log n} = n$$

$$\therefore 2^k = \log n$$

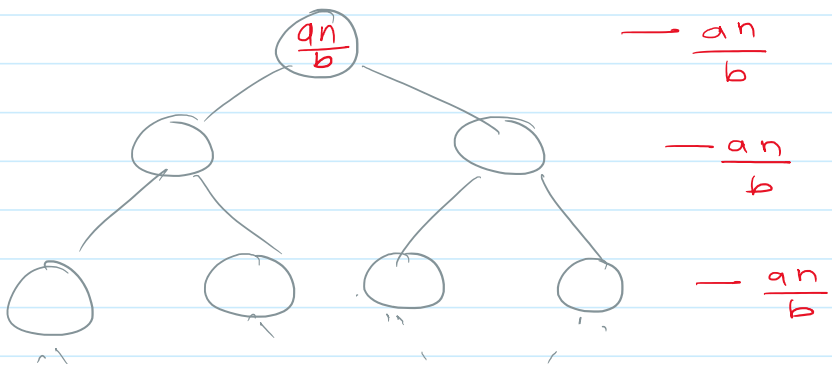
$$\therefore k = \log \log n \quad \text{... height}$$

$$\therefore T(n) = \text{height} \cdot \text{cost per height}$$
$$= (\log \log n) \cdot n$$

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



② Input: a-sorted arrays each of n/b
output: Find single sorted array with all elements.



$k = \log a$

Time complexity of merge algorithm

$T(p) = n$. — At each level of tree

Here $n = \frac{an}{b}$

$\therefore T(p) = \frac{an}{b}$

$\therefore$ Merging of $\frac{an}{b}$ elements at each level for $\log a$ levels

Time complexity = $\frac{an}{b} \log a$

③ Input : n/a subarrays each of size n/b
Output : Single sorted array with all elements.
sort the array using bubble-sort.

Time complexity to sort n/a sub-arrays each of size n/b

$$T(p) = \frac{n}{a} \cdot \left(\frac{n}{b}\right)^2$$

Time complexity to merge n/a arrays
height of tree will be = $\log n/a$

$$T(p) = \frac{n^2}{ab} \cdot \log(n/a)$$

Total

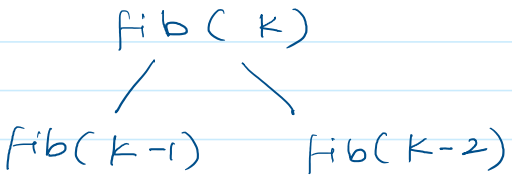
$$T(p) = \frac{n^2}{ab} \log\left(\frac{n}{a}\right) + \frac{n^3}{ab^2}$$

Here we don't know a & b

$$T(P) = O\left[\frac{n^2}{ab}\log\left(\frac{n}{a}\right) + \frac{n^2}{ab^2}\right]$$

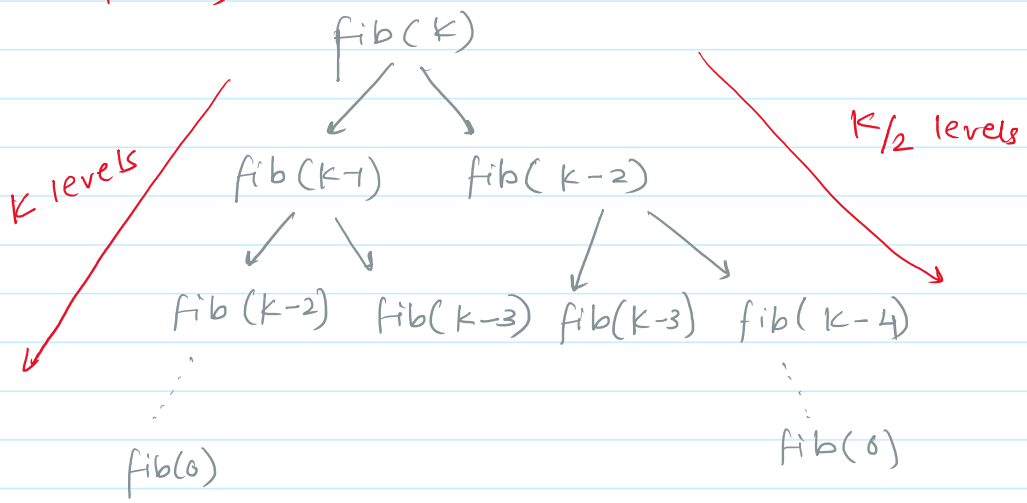
④ Fibonocci mergesort

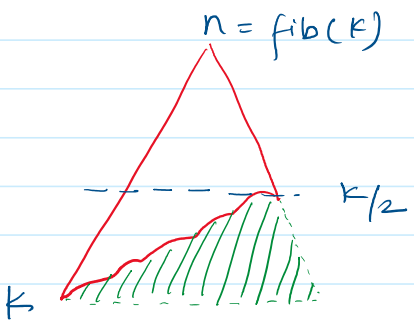
sorting is done in the following manner.



Input: fib(k) elements array
output: sorted array using fibbonocci mergesort.

n = fib(k)





For merge-sort algorithm

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

↑ elements ↑ height of tree

∴ T(P) for fibonocci mergesort

height = K

$T(P) = K \cdot fib(K)$

$= O(K \cdot fib(K))$

height = K/2

$T(P) = K/2\ fib(K)$

$= \Omega(K\ fib(K))$

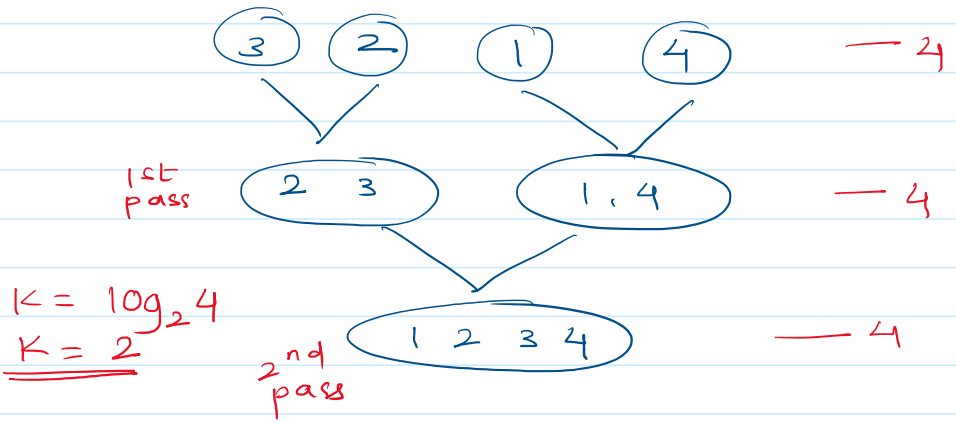
$\therefore\ T(P) = \Theta(K \cdot fib(K))$

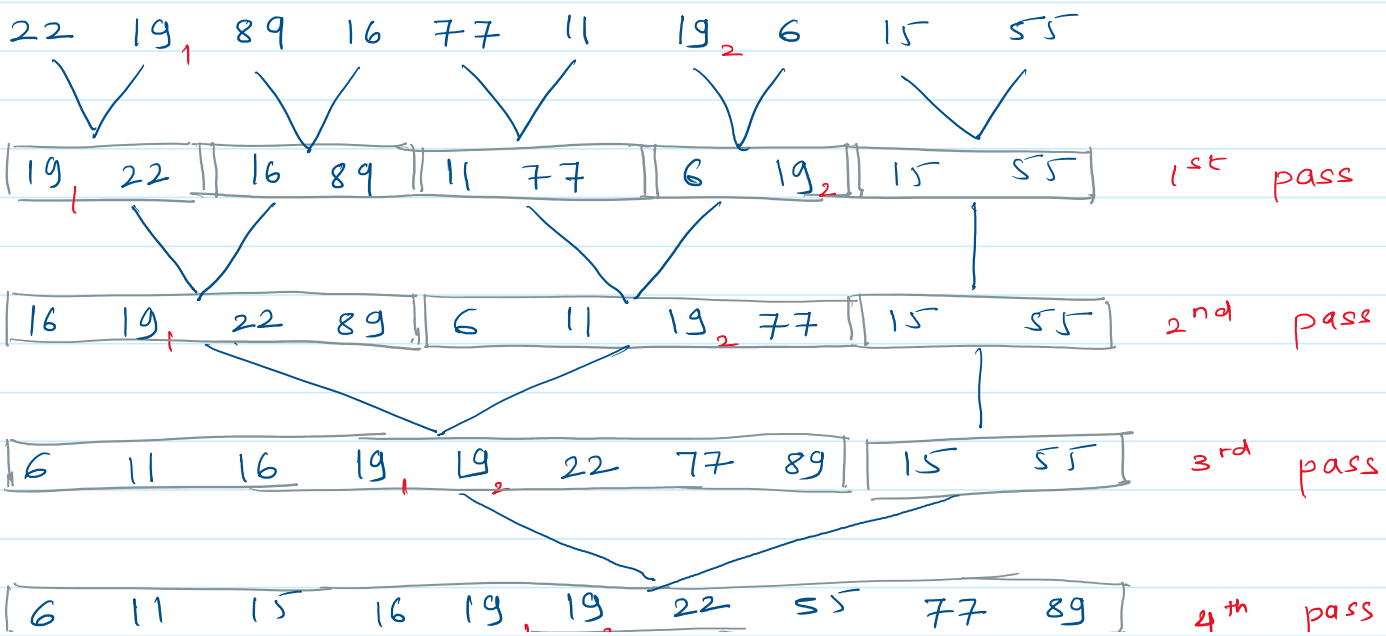
⑤ Consider the following array

22 19_1 89 16 77 11 19_2 6 15 55

What will be the output after second pass of straight 2-way mergesort algorithm.

straight 2-way mergesort will follow bottom up approach





order of elements
didn't change

∴ Merge sort is stable sorting technique

⑥ Input : Two sorted arrays distinct elements A.size = m B.size = n

output : $A \cap B$ = ? $A \cup B$ = ? for worst case Find T(n)

Using Linear search

for $A \cap B$ ... Brute force

$T(p) = \Theta(m.n) \approx \Theta(n^2)$

for $A \cup B$ ... Brute force

$T(p) = \Theta(m.n) \approx \Theta(n^2)$

Using Binary search.

for $A \cap B$

$T(p) = \Theta(m.\log n) \approx \Theta(n \log n)$

for $A \cup B$

$T(p) = \Theta(m.\log n) \approx \Theta(n \log n)$

Using merge algorithm

$A \cap B$ Algorithm

```
A∩B ( a[], b[] ) {
    while ( i < a.size && j < b.size) {
        if ( a[i] == b[j])
            print
        else if ( a[i] < b[j])
            i++ ;
        else  j++ ;
    }
}
```

$A \cup B$ Algorithm

```
A∩B ( a[], b[] ) {
    while ( i < a.size || j < b.size) {
        if ( a[i] < b[j])
            print , i++
        else if ( a[i] > b[j])
            print, j++
```

```
        else  i++ , j++
        while ( i == a.size && j < b.size)
            print b[j++]      $ break last
        while ( j == b.size && i < a.size)
            print a[i++]      $ break last
        }
    }
```

For both

$\therefore T(p) = m+n \approx 2n$

$= O(n)$

If un-sorted array

first sort the algorithm

$\therefore T(p) = 2n + 2\,n\log n$

$= \Theta(n\log n)$ — Better than Brute force

5. Quick Sort

10 April 2021 07:25 PM

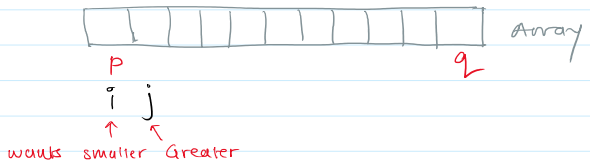
partial Divide and Conquer
∴ No combine

Here log n includes stack space for an Algo too covers recursive Algo

- In-place Algorithm (changes done directly on input array) — can take only constant space (max O(log n)) .... log n + C (stack space ↑, constant ↘)
- Not stable (Order of repeated elements may change)
- Mostly used sorting algorithm

Notations

p = 1st element index
q = last element index
i = p
j = p+1
} Initially.



- Algorithm

J moves to right,
If it gets greater, keeps it
smaller, gives to i
} Just to remember

① Select any element you want as a pivot.
lets take pivot = 1st element

② Compare j with pivot

j > pivot → j++

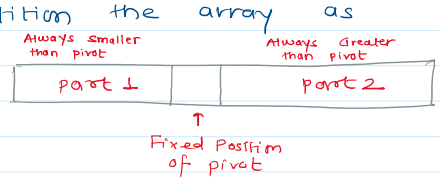
j <= pivot → { i++ → so i gives greater value
swap(a[i], a[j])
j++
} conquer

repeat until j <= q ← last element
At last swap(a[pivot], a[i])

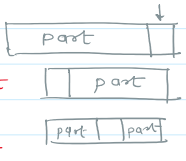
If i is incremented k times implies
there are k elements less than pivot
k = i+1

So we get fix position of pivot at every scan

③ Partition the array as ← partition is not blind.



possibilities of pivot
① Greatest element → only left part
② smallest element → only right part
③ not greatest not smallest → two parts left & right



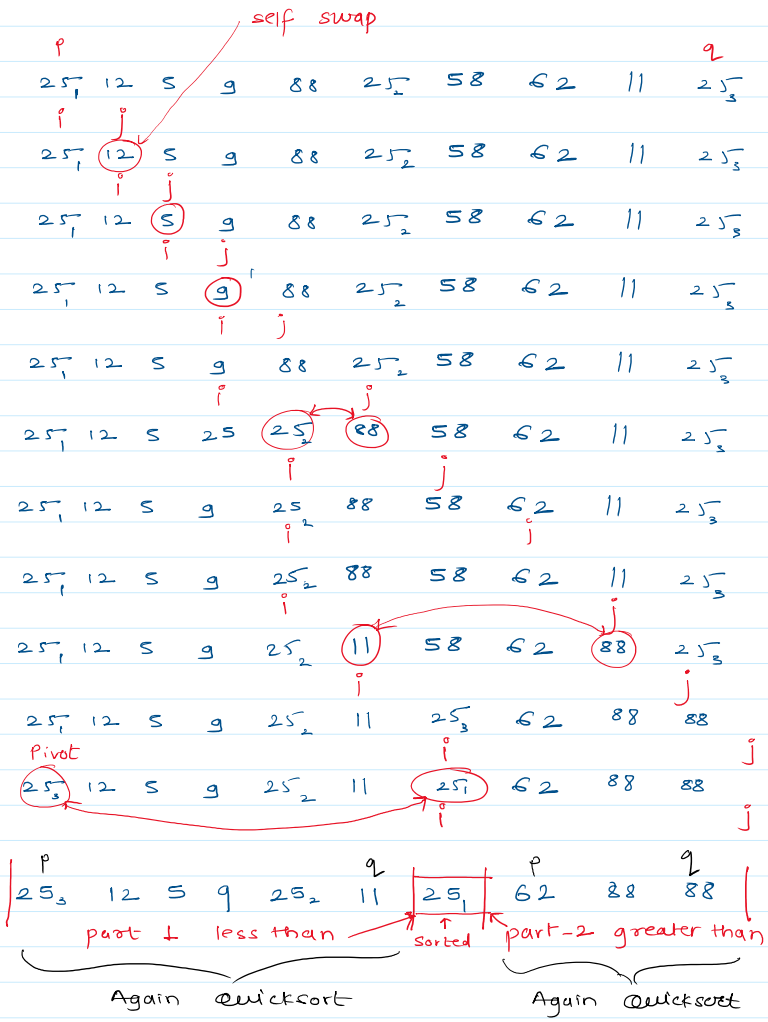
④ Again apply Quicksort to Part 1 & part 2 separately. → Divide
No combine required.

Fast than mergesort
① No combine required ← constant factor
② No swapping of arrays required ← constant factor

- Psuedocode:

```
Partition( a, p, q) {          ← helps in dividing the array into less than pivot and greater than pivot.
    i = p
    pivot = p
    for( j = p+1 ; j ≤ q ; j++)
        if (a[j] ≤ x) {
            i++                                   } n
            swap ( a[i], a[j] )
        }
    }
    swap ( a[i], a[pivot])                        } constant
    return i;
}
```

Example




```
Quicksort ( arr[], p, q ) {
    if  p == q  return ;                          } c
    else
        i = partition (arr[], p, q)               — n
        partition( arr[], p, i-1)                 — T( m-p)
        partition( arr[], i+1, q)                 — T( q-m)
}
```

— Time complexity

$$T(n) = \begin{cases} O(1) & \text{if } n == 1 \\ n + T(m-p) + T(q-m) & n > 1 \end{cases}$$

$m-p \Rightarrow 0, n/2, n-1$

$n + T(m-p) + T(q-m) \quad n \geq 1$

$m-p \Rightarrow 0, n/2, n-1$
$q-m \Rightarrow n-1, n/2, 0$

Best case :

$m-p = q-m = n/2$

Divide → $T(n) = 2T(n/2) + n$ ← partition Algo

By using recursive tree method

$T(n) = \Theta(n \log n)$ ← Best case ... stack space = $\log n$

because tree height $\log n$

Most of the time quick sort will give balanced partition ∴ Best case

Worst case

$m-p = 0$	$n-1$
$q-m = n-1$	0

Divide → $T(n) = T(n-1) + n$ ← partition Algo

By using substitution method

$T(n) = \Theta(n^2)$ ← worst case

Stack space = n

∵ substitution n times

Generalized Time complexity

$T(n) = n + T(m-p) + T(q-m)$

$T(n) = \Omega(n \log n)$
$= O(n^2)$

Stack space

$S(n) = \Omega(\log n)$
$= O(n)$

Q. Consider following inputs

① 1, 2, 3, 4 . . . n-, n ← Ascending
② n, n- n 2, n-3, . . 2, 1 ← descending
③ n, n, n n . n ← All equal

Let c_1 c_2, c_3 be the comparisons made to keep them in ascending order using quicksort relation between c c_2 c_3 ?

→ All inputs have n elements

Quicksort always keeps pivot element at its right position.
It always considers the remaining parts of array to be unsorted.

∴ providing number of elements equal in a inputs number of comparisons will be same.

considering total swaps.

① As input 1 is already sorted in ascending

considering total swaps.

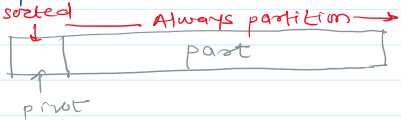
① As input I is already sorted in ascending order,

i.e. if we consider any pivot element it will be always at right position

so only one swap needed

swap(a[i], a[pivot]) ← compulsary swap At end of each pass
with itself

But for partition it will be worst case



∴ For n elements

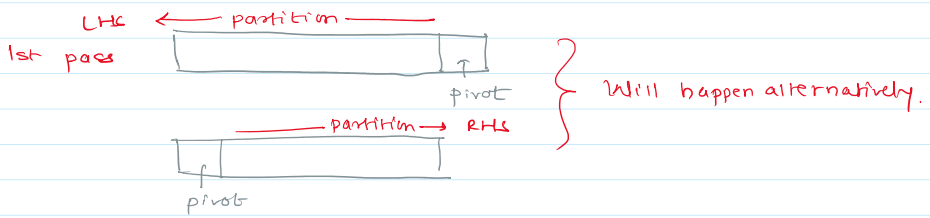
	comparisons	swaps
T(n-)	n-1	1
T n-2	n 2	1
T(n 3)	n-3	1
T(1)	1	1

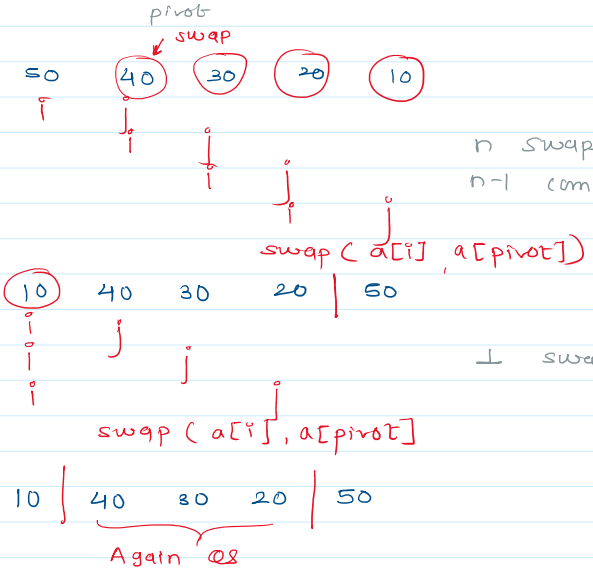
∴ No. of comparisons = $\frac{n.(n-1)}{2} = \frac{n^2+n}{2}$

number of swaps = n = O(n)

② Input array in descending order
consider pivot = p

Everytime 1st element will be kept at last position





Above condition will happen alternately.

	comparisons	swaps
T(n)	n-1	n
T n-	n-2	1
T n-2	n 3	n 2
T)	1	2

Total comparisons = $(n-1)+(n-2)+(n-3)\cdots 2+1$

$C_2 = \frac{n^2+n}{2}$

Total swaps $= n + 1 + (n-2) + 1 + (n-4) + \cdots + 4 + 1 + 2$

$= \frac{n}{2}(1) + [n + (n-2) + (n-4) \cdots + 4 + 2]$

$= \frac{n}{2} + [2 + 4 + 6 \cdots (n-2)]$

$= \frac{n}{2} +$ [sum of $\frac{n}{2}$ even numbers, 1 — n, total n/2 even numbers]

$= \frac{n}{2} + \left(\frac{n}{2}\right)^2 + \frac{n}{2}$

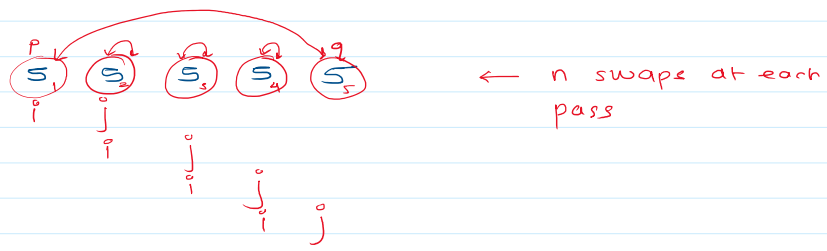
$= n + \frac{n^2}{4}$

swaps $= O(n^2)$

③ For array will all same elements

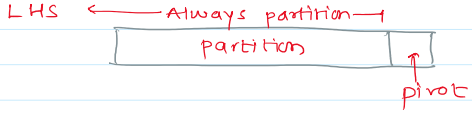
consider pivot = p

ex.



← n swaps at each pass

s_5 s_2 s_3 s_4 s_1 ← Non-stable



Always the partition will divide the array into left hand side, giving us worst case

	comparison	swaps
T(n)	n-1	n
T(n-1)	n-2	n-

$T(n-2)$	$n-3$	$n-2$
$T(1)$	1	2

Time complexity = $C_3 = \frac{n^2+n}{2}$

Number of swaps = $\frac{n^2-n}{2}$

So Relation between these

comparisons

$C = C_2 = C_3$

swaps

$C_1 < C_2 < C_3$

- For quick sort

  Best case = $\theta(n \log n)$

  Worst case - $\theta(n^2)$

Average case

Method 1

Best case → $T(n) = 2T(n/2) + n$

Worst case → $T(n) = T(n-1) + n$

$T(n) = T(n-1) + n$ ← Worst case

substitute for best case

$T\,n = 2T\left(\frac{n-1}{2}\right) + n - 1 + n$

$\approx 2T\left(\frac{n}{2}\right) + 2n$

which is nearly equal to best case differs by only constant.

∴ Average case - $O(n \log n)$

Method 2.

Consider array was divided as

① $\frac{n}{2} \mid \frac{n}{2}$

$T(n) = 2T\left(\frac{n}{2}\right) + n$

$= n \log_2 n$ — $\theta(n \log n)$

② $\frac{n}{3} \mid \frac{2n}{3}$

$T(n = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$

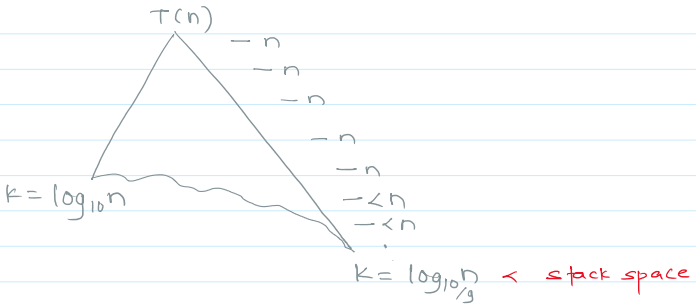
By recursive tree method

$T(n) = n \log_{3/2} n = \Theta(n \log n)$

⋮

③ $\frac{n}{10}$ | $\frac{9n}{10}$

$T(n) \quad T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$



$T(n) = n \cdot \log_{10/9} n = \Theta(n \log n)$

By evaluating all these conditions.
We can say that,

most frequent time complexity is nlogn.

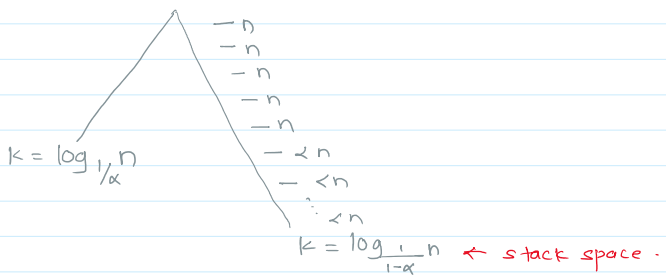
∴ Average = $\Theta(n \log n)$

This can be stated as

$T(n) = T(\alpha n) + T((1-\alpha) n) + n$

where $0 < \alpha < 1$

α is partition



$T(n) = n \log_{\frac{1}{1-\alpha}} n = O(n \log n)$

stack space = $\Theta(\log n)$

- Conclusion:
  - If array is sorted
    We get worst case
    $T(n) = T(n-1) + n$
    $T(n) = \Theta(n^2)$

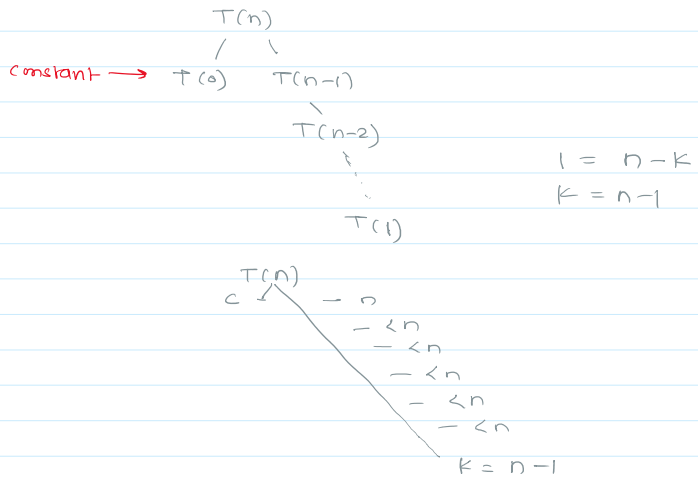
- If array is unsorted
  we get best case
  $T(n) = 2T(\frac{n}{2}) + n$

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

For worst case
If Array is divided as

① 0 | n-1

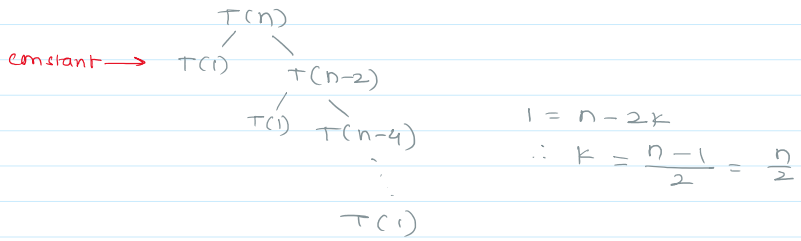
$T(n) = T(0) + T(n-1) + n$



$\therefore T(n) = n \cdot (n-1) = \Theta(n^2)$

② 1 | n-2

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



$T(n) = \frac{n}{2} \cdot n = \Theta(n^2)$

⋮

③ 9 | n-9

$T(n) = T(9) + T(n-9) + n$

Here left height of tree will be constant for every value of n

$\therefore T(n) = \frac{n}{9} \cdot n = \Theta(n^2)$

By evaluating above conditions
We can conclude that
if array is divided as constant elements

By evaluating above conditions
We can conclude that
if array is divided as constant elements
in one part then we get worst case

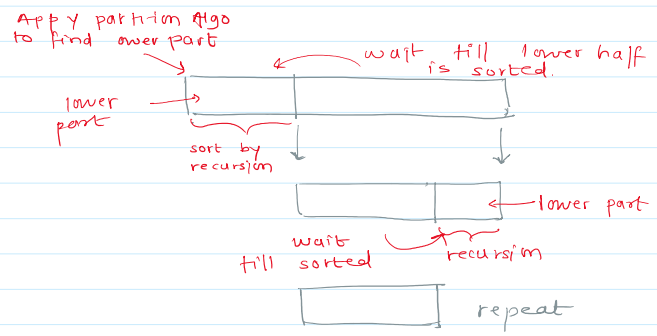
$\therefore T(n) = T(c) + T(n-c) + n$
$= \Theta\left(\frac{n}{c} \cdot n\right) = \Theta(n^2)$

- For inplace - Algorithms we need to have maximum space complexity to be $O(\log n)$
But above algorithm takes n^2 stack-space in worst case.

∴ To Fix this
① Apply recursion only on one side at a time
we will apply on the partition with less elements.
② Two posibilites.
If minimum partition is always $1 \mid n-1$ ← worst case for Time complexity
stack space = $O(1)$
If partition is always $n/2 \mid n/2$ ← Best case for Time complexity
$S(n) = S\left(\frac{n}{2}\right) + n$ ← only one recursion at a time.
$= O(\log n)$

- Psuedocode (Best QuickSort Algorithm)

```
Quicksort ( arr[], p, q ) {
    If  p == q
        return
    while ( p ≤ q ) {
        i = partition( arr[], p, q)
        if ( i - p < q - i )        ← left part small?
            Quicksort ( arr , p , i-1)   ← if recursim
            p = i+1                 ← Big part will be taken care by while oop.
    else
            quicksort ( arr , i+1 , q)   ← f not other half recursim
            q = i - 1
    }
}
```



This will be repeated until $p = q$

Every greater part will wait till smaller are sorted
So at every instance only one part will

Every greater part will wait till similar are sorted

So at every instance only one part will go under recursion

So the space complexity will never exceed than logn in worst case also

$S(n) = S(1) + c$ — sorted array

$= S(n/2) + c$ — unsorted array

$\therefore S_n = \Omega(1)$

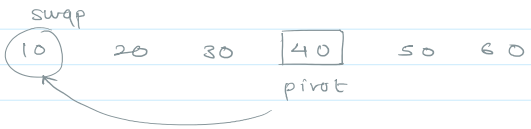
$= O(\log n)$

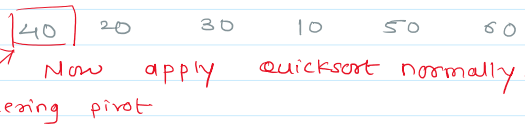
- Randomized Quicksort

Quicksort time complexity for sorted arrays = $O(n^2)$

We can improve the worst case time complexity such that the monopolic (only left/right) partition is reduced significantly.

We can take pivot element randomly instead of choosing first element always





Now apply quicksort normally.

considering pivot

Next pass

10 20 30 | 40 | 50 60

Again Apply RQS

Again Apply RQS

Doing this will significantly reduce the monopolic partition.

$T(n) \approx \Omega(n \log n)$

Most of the times for sorted array.

Time complexity

	Best case	Worst case	Average Case
Normal QS	$\Omega(n \log n)$	$O(n^2)$	$\Theta(n \log n)$
Improved QS	$\Omega(n \log n)$	$O(n^2)$	$\Theta(n \log n)$
Randomized QS	$\Omega(n \log n)$	$O(n^2)$	$\Theta(n \log n)$

↖ very less probability.

Space complexity

	Best case	Worst case	Average Case
Normal QS	$\Omega(\log n)$	$O(n)$	$\Theta(\log n)$
Improved QS	$\Omega(1)$	$O(\log n)$	$\Theta(1)$
Randomized QS	$\Omega(1)$	$O(\log n)$	$\Theta(1)$

Quick Sort Problems

13 April 2021 15:07

★ Q In Quicksort the sorting of n-elements the $n/5^{th}$ smallest element is selected as pivot in $O(n)$ time complexity. Then what will be worst case time complexity.

A. $O(n^2)$ ✓ C. $O(n^3)$ ✓

B. $O(n)$ D. $O(n \log n)$ ✓ ← Big O means answer $\geq n \log n$

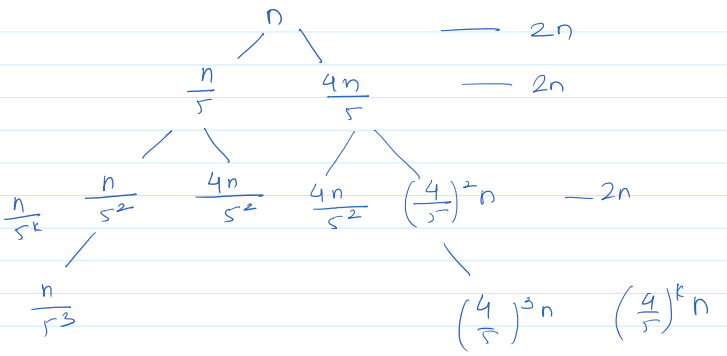
If we select $n/5^{th}$ element no gaurantee which position it will go.

But selecting $n/5^{th}$ smallest element will gaurantee always 5th position.

→ Array will alway be divided as $n/5$ — $4n/5$

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right) + 2n$$

→ Pivot selection + partition Algo



$\frac{n}{5^k} = 1$

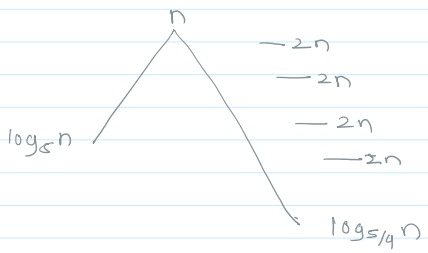
$n = 5^k$

$\therefore k = \log_5 n$

$\left(\frac{4}{5}\right)^k n = 1$

$n = \left(\frac{5}{4}\right)^k$

$\therefore \log_{5/4} n = k$



$\therefore T(n) = O(2n \cdot \log_{5/4} n) = O(n \log n)$

$T(n) = \Omega(2n \log_5 n) = \Omega(n \log n)$

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

If we take only $n/5^{th}$ element in above question will not gaurantee what will be partition.

∴ In Best case.

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$= \Theta(n \log n)$$

In worst case.

$$T(n) = T(n-1) + n$$

$$= \Theta(n^2)$$

$\therefore T(n) = \Omega(n \log n)$

$= O(n^2)$ ← satisfies Big O

~~A. $O(n^2)$~~ ~~C. $O(n^3)$~~

B. $O(n)$ ✗ D. $O(n \log n)$ ✗

Q.2. In QS the sorting of n-elements, The $n/10^{th}$ largest element is selected as pivot using $O(n^2)$ Time complexity. Find worst case time complexity.

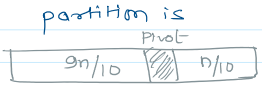
A. $O(n \log n)$ ~~C. $O(n^3)$~~

~~B. $O(n^2)$~~ D. $O(n)$

$n/10^{th}$ largest element selected as pivot

Hence $9n/10 - n/10$ partition is guaranteed

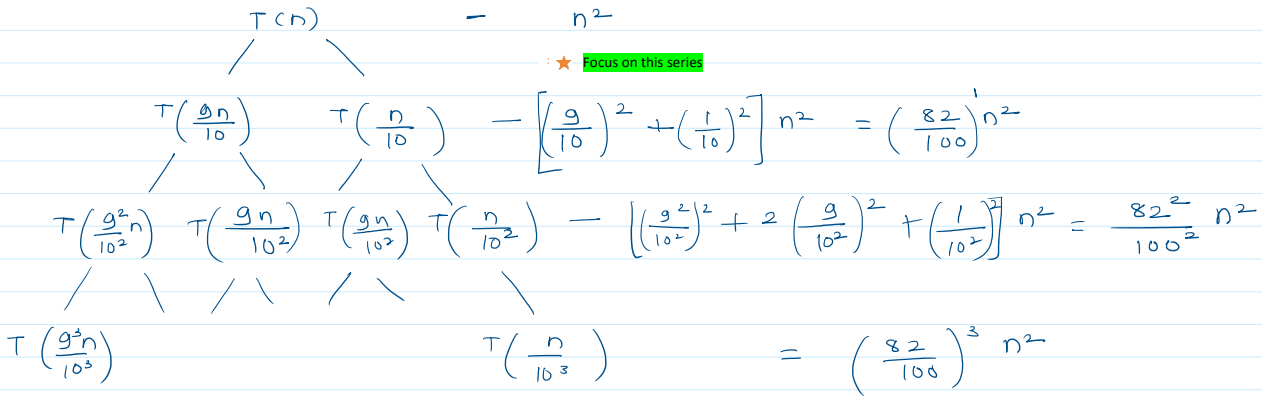
In worst case



$\therefore T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + n + n^2$ → n^2 dominant

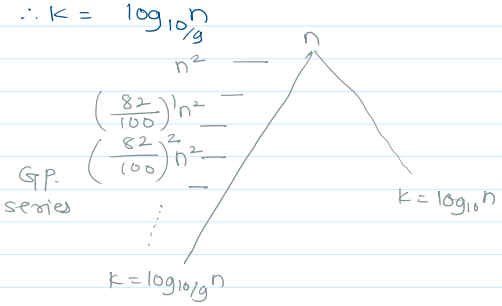
↑ Partition Algo ↑ For selecting pivot

By recursive tree method



★ Focus on this series

$\therefore k = \log_{10/9} n$ $k = \log_{10} n$



Solving G.P. series

$$= \left[n^2 + \left(\frac{82}{100}\right)^2 n^2 + \left(\frac{82}{100}\right)^3 n^2 \cdots + \left(\frac{82}{100}\right)^k n^2\right]$$

$$= n^2\left[1 + \left(\frac{82}{100}\right)^2 + \left(\frac{82}{100}\right)^3 \cdots + \left(\frac{82}{100}\right)^k\right]$$

G.P series

$$= n^2\left[\downarrow\left(\frac{1 - \left(\frac{82}{100}\right)^k}{1 - \frac{82}{100}}\right)\right]$$

Here for best case $k = \log_{10} n$

for worst case $= \log_{10/9} n$

But by further solving G.P. series becomes constant

for worst case $= \log_{10/9} n$

But by further solving

G.P. series becomes constant

$\therefore \approx c_1 n^2$

$\therefore$ By above

Best case: (G.P. with $k = \log_{10} n$)

$T(n) = c_1 n^2$

$= \Theta(n^2)$

Worst case (G.P. with $k = \log_{10/9} n$)

$T(n) = c_2 n^2$

$= \Theta(n^2)$

$\therefore T(n) = \Theta(n^2)$

Q3 What is worst case time complexity if we select middle element as pivot?

There is no guaranteed partition if we select middle element as pivot.

$\therefore$ In worst case

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

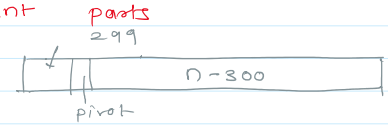
* If Median is mentioned as a pivot element it is $\left(\frac{n}{2}\right)^{th}$ smallest element

Middle $= \left(\frac{n}{2}\right)^{th}$ element

Average $\neq$ Median

Q What will be Best case time complexity if we select 300th largest element as pivot using $O(n^2)$ Time complexity

Here array will always be divided into 2 constant parts



$\therefore T(n) = T(299) + T(n-300) + n^2 + n$

(constant neglect)

$= T(n-300) + n^2$

Solving by substitution

$= T(n-600) + (n-300)^2 + n^2$

$= T(n-900) + (n-600)^2 + (n-300)^2 + n^2$

$= T(n-300k) + (n-300k)^2 + (n-300(k-1))^2 + \dots + n^2$

considering

$T(1) = T(n-300k)$

$\therefore n - 300k = 1$

$n - 1 = 300k$

$\frac{n-1}{300} = k$

$= T(1) + (n-300(n-1))^2 + (n-300(n-1-300))^2 + (n-300(n-1-600))^2 + \dots n^2$

$\frac{}{300}$

$$= T(1) + \left(n - 300\left(\frac{n-1}{300}\right)\right)^2 + \left(n - 300\left(\frac{n-1-300}{300}\right)\right)^2 + \left(n - 300\left(\frac{n-1-600}{300}\right)\right)^2 + \cdots n^2$$

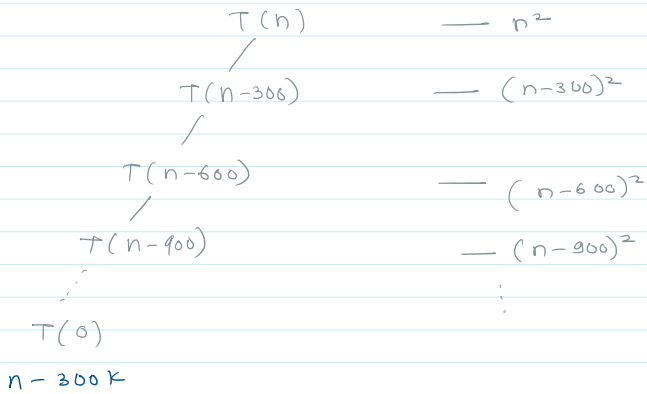
$$= 1 + \left(n - n + 1\right)^2 + (301)^2 + (601)^2 + \cdots n^2$$

$$= 1 + 1^2 + (301)^2 + (601)^2 + \cdots n^2$$

$$\approx \frac{n^2(n^2+1)}{2}$$

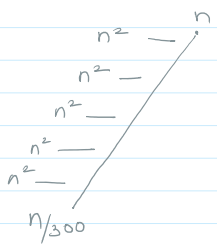
$$= \Theta(n^2)$$ ← Best & worst case same

Easy Method = $T(n-300) + n^2$



$\therefore n - 300k = 0$

$k = \frac{n}{300}$



only unary tree

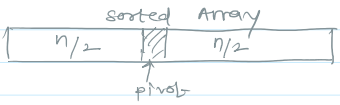
$\therefore$ Best case = Worst case

$$\therefore T(n) = \frac{n}{300} \cdot n^2$$

$$= \Theta(n^3)$$

Q In sorted array of n-elements if we take middle element as pivot. What is worst case time complexity

If array is sorted, and we select middle element as pivot.
It will always divide the array into two halves.



$\therefore$ Always Best case gauranteed.

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

$$= \Theta(n \log n)$$

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

$= \Theta(n \log n)$

Note : In the given array of N elements to find median
T(n) = O(n) using 'Median of Medians ' algorithm.

Time complexity to find $\left(\frac{n}{10}\right)^{th}$ smallest element $= \Theta(n)$

6. Selection Procedure

13 April 2021 07:50 AM

Input: Array of n elements, and an element k.
Output: Kth smallest element.

- Algorithm

1. Method 1

No mentioning of array to be already sorted/unsorted so best case by mergesort

① Sort elements ← nlogn - mergesort
② Return a[k]

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

2. Method 2

places smallest element at its right position.

① Run selection sort for k passes
② Return a[k]

Worst case = $O(n^2)$

3. Method 3.

Refer Quicksort

① Apply partition Algo
② check the index of pivot.
③ if k = idex of pivot
return a[k]
else if index is < k
find the element in right part.
if index > k
find the element in left part.
④ Repeat

psuedocode:

```
selectionproc ( arr[], p, q, k) {
    if p == q
        return arr[p]
    i = partition( arr[], p, q)
    if ( i == k)
        return arr[i]
    else if ( k<i )
        return selectionproc( arr[], p, i-1, k)
    else
        return selectionproc(arr[], i+1, q, k)
}
```

Best case:

If $i = k$ in first pass, or partition always happens best i.e. $n/2 - n/2$

$$\therefore T(n) = T\left(\frac{n}{2}\right) + n$$

$$T(n) = \Theta(n)$$ ← Most of the time we will get n so its Average too

Worst case

If partition happens only one side

$$\therefore T(n) = T(n-1) + n$$

$$T(n) = \Theta(n^2)$$

Generalized:

$$T(n) = T(m-p) + n$$

or

$$T(q-m) + n$$

Best case = $\Theta(n)$
Average case = $\Theta(n)$
Worst case = $\Theta(n^2)$

- We can further improve algorithm by selecting pivot by median of medians Algorithm

median of medians = $O(n)$

Always best partition if pivot = median

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

$$= \Theta(n)$$

For every case

7. Counting number of Inversions

22 March 2021 21:34

Input: An array of elements
Output: Count number of inversions

Inversion

In an array $i < j$ but $a[i] > a[j]$
i.e. count every smaller element to right for every element

30	10	35	25	45	1	2
↓	↓	↓	↓	↓	↓	↓
10	1	25	1	1		
25	2	1	2	2		
1		2				
2						

By Linear search / Brute force

$T(n) = \theta(n^2)$

Algorithm : (By Merge sort)

same algorithm as of merge sort. Just modify merge code.

① count remaining elements in the left array every time an element from right is moved to new array

② If any group is traced no further inversions

```
merge ( a[], b[] ) {
    while( i < a.size && j < b.size) {
```

```
        If ( a[i] > b[j] ) {
            count ++
            result[k++] = b[j++]
        else
            result[k++] = a[i++]
    }
    copy remaining to result.
    return result, count
}
```

$\therefore \quad T(P) = 2T(n/2) + n$

$= O(n \log n)$

8. Strassen's Matrix Multiplication

12 April 2021 06:34

- Matrix addition

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}_{2\times 3}$$

$$B = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \end{bmatrix}_{2\times 3}$$

$$C = A+B = \begin{bmatrix} A_{11}+B_{11} & A_{12}+B_{12} & A_{13}+B_{13} \\ A_{21}+B_{21} & A_{22}+B_{22} & A_{23}+B_{23} \end{bmatrix}$$

```
matrix_Addition ( A[][], B[][]) {

    for (i = 0 ; i < r ; i++)  {
        for ( j = 0 ; j < c ; j++)  {
            C[i][j] = A[i][j] + B[i][j]
        }
    }
}
```

Time complexity = $\theta(n^2)$

- Matrix Multiplication

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}_{2\times 3}$$

$$B = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix}_{3\times 3}$$

Multiplication possible only if $A_{ij} = B_{jj}$

A row × B column

$$C = \begin{bmatrix} A_{11}B_{11} + A_{12}\cdot B_{21} + A_{13}\cdot B_{31} & \cdot & \cdot \\ A_{21}\cdot B_{12} + A_{22}B_{22} + A_{23}B_{32} & \cdot & \cdot \end{bmatrix}_{2\times 3}$$

In Result Matrix

For each element n multiplications needed

Total elements = $i \times j = 2 \times 3 = 6$

∴ Total multiplications needed $= i \cdot j \cdot n$

∴ Time complexity = $\Theta(n^3)$

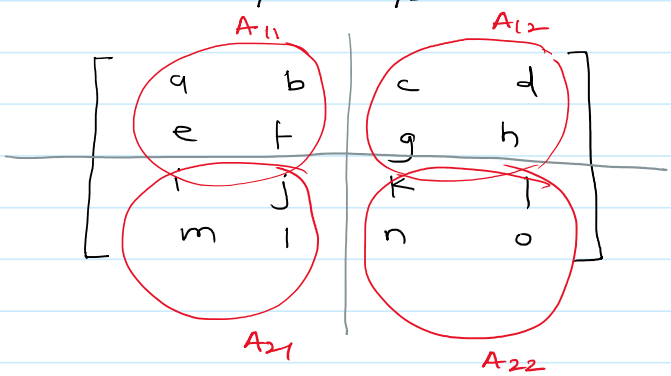
```
matrix_multiplication ( A[][], B[][] ) {

    for( i = 0 ; i < r ; i++ )  {
        for( j = 0 ; j < c ; j++ )      {
            for( k = 0 ; k < c ; k++ )      {
                c[i][j] += A[i][k] * B[k][j]
            }
        }
    }
```

- Matrix multiplication by DAC

Algorithm:

① Divide the given array $n \times n$ as $n/2 \times n/2$



② $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$

$C_{12} = A_{11} \cdot B_{12} + A_{12} B_{22}$

$C_{21} = A_{21} B_{11} + A_{22} B_{21}$

$C_{22} = A_{21} B_{12} + A_{22} B_{22}$

8 Multiplications of $\frac{n}{2}$ matrices

and 4 Additions of $\frac{n}{2}$ matrices

T(n) for addition

$$T(n) = 8T\left(\frac{n}{2}\right) + 4 \cdot \frac{n}{2} \cdot \frac{n}{2}$$

$$T(n) = 8T\left(\frac{n}{2}\right) + n^2$$

Recurrence relation

$$T(n) = 1 \qquad n \leq 2$$

$$= 8T\left(\frac{n}{2}\right) + n^2 \qquad n > 2$$

But, by strassen's findings
it can be improved further

$$T(n) = 1 \qquad n \leq 2$$

$$= 7\left(\frac{n}{2}\right) + 18\frac{n^2}{4} \qquad n > 2$$

$$\therefore T(n) = O\left(n^{2.81}\right)$$

To make the above algorithm to work for any matrices like 2×3 3×4
take the biggest number and convert both matrices in 4×4, by appending 0's if its not in powers of two convert to closest bigger power of 2.

Above Algorithm is more efficient for sizes of $2^k \times 2^k$ matrices

9. Continuous Maximum Sub-array Sum

12 April 2021 06:09 PM

Input: Array of n- integers (+ve / -ve)
Output: Find maximum sub-array sum

Array: 20 -25 130 -80 10 90 -43 12 -48

Answer

- Algorithm:

```
foo( arr[], p, q ) {
    if ( p == q )
        return p
    mid = ⌊(p+q)/2⌋
    left_sum  = foo( arr, p, mid )
    right_sum = foo( arr, mid+1, q)
    cross_sum = crosssum ( arr, p, mid, mid+1, q)

    return max( left_sum, cross_sum, right_sum)
}

crosssum ( arr[], p, m, n, q)
    for( i = m ; i ≥ p ; i--) {
        if ( LTS < LTS + arr[i] ) {
            LTS += arr[i]
            lpos = i
        }
    }
    for ( j = q ; j ≤ q ; j++)
        if ( RTS < RTS + arr[j]) {
            RTS += arr[i]
            rpos = j
        }
    }
}
```

conquer → c

divide → 2T(n/2)

conquer → n/2 + n/2 = n

combine → c

p + m + n + q = All elements

All elements → n/2

→ n/2

∴ Recurrence Relation for time

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

$$\therefore T(n) = 2T\left(\frac{n}{2}\right) + n$$

By master's theorem

$a = 2$ $\quad f(n) = n$

$b = 2$

$$f(n) = n^{\log_b a - \varepsilon}$$

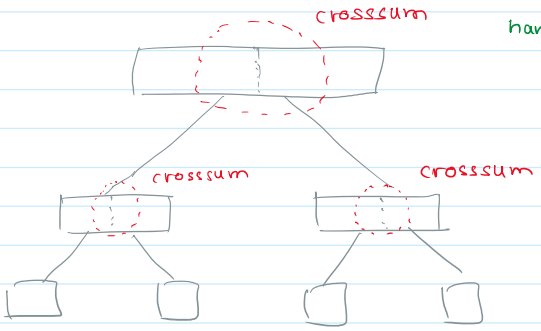
$$n = n^{\log_2 2 - \varepsilon}$$

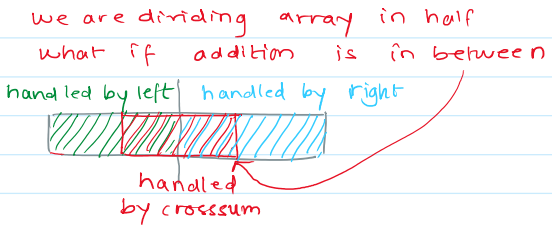
$$n = n^{1-\varepsilon}$$

$$n = n^{\log_2 2 - \varepsilon}$$
$$n = n^{1-\varepsilon}$$
$$\varepsilon = 0$$

case 3-

$$\therefore T(n) = \Theta\left(n^{\log_b a}(\log n)^{\varepsilon+1}\right)$$
$$= \Theta(n \log n)$$





Sorting Techniques

13 April 2021 10:03 AM

1. Heap Sort
2. Insertion Sort

Heap Sort

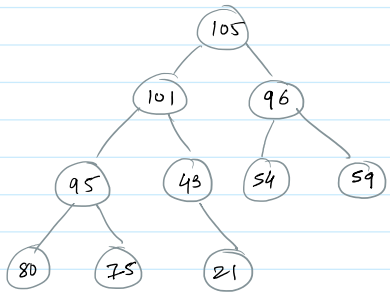
13 April 2021 11:57 AM

Refer Binary tree first

Heap Trees (Always ACBT / CBT)

- Max Heap Tree — parent ≥ children
- Min Heap Tree — parent ≤ children

- Max-heap Tree (Binary search tree & max heap are different)
  - Should be ACBT / CBT — BST can't guarantee ACBT
  - At every node parent ≥ each child



- ACBT
- Every parent ≥ each child

Array representation

105 . 101 . 96 . 95 . 43 . 54 . 59 . 80 . 75 . 21

$n = 10$

levels = $K = \log_2(10+1)$

$K = 4$

$h = K - 1 = 3$

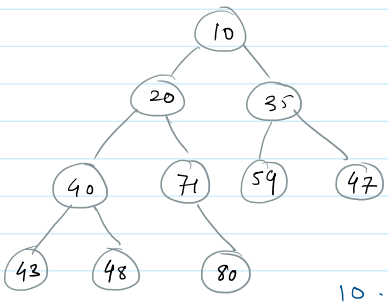
leaf nodes = $\left\lceil \frac{n}{2} \right\rceil = 5$ → leaf nodes always ≥ Nm-leaf

leaf + 1 = Nm-leaf

or leaf = Nm-leaf In ACBT

Nm-leaf = $\left\lfloor \frac{n}{2} \right\rfloor = 5$

- Min-heap tree
  - ACBT / CBT tree
  - At every node parent ≤ children



- ACBT
- Every parent ≥ each child

10 . 20 . 35 . 40 . 71 . 59 . 47 . 43 . 48 . 80

Q. n-elements max-heap tree. Find time complexity to find (data structure : Array)

1. 1st Maximum
2. 2nd Maximum

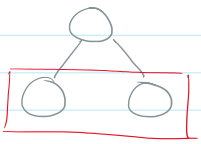
Complexity to find (data structure : Array)

1. 1st Maximum
2. 2nd Maximum
3. nth Maximum

(1) The root of the max-heap Tree is always the maximum

$\therefore T(n) = \Theta(1)$

(2) For 2nd maximum

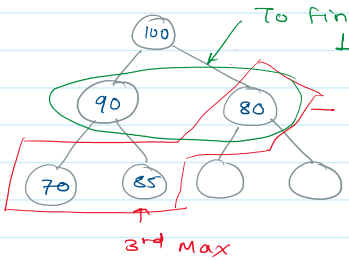


comparison between these two.

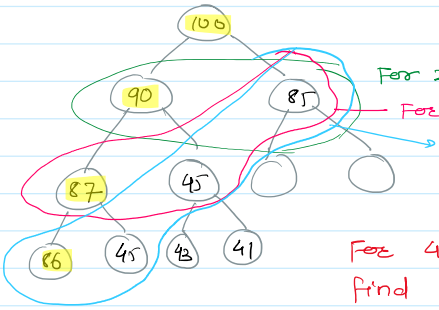
2 comparisons.

$T(n) = 2 = \Theta(1)$

(3) For 3rd Maximum



(4) For 4th Maximum



For 4th maximum we have to find every previous maximum.

$T(n) = 1+2+3 = 7 = \Theta(1)$

(5) For nth Maximum

$T(n) = \frac{(n-1).n}{2} \approx \Theta(n^2)$

For 1st minimum, max-heap tree doesn't help. it takes $\Theta(n^2)$ time complexity.

But, given tree is stored as array. So we can find minimum element by applying one pass of selection sort

$T(n) = \Theta(1)$

	Max-heap	Min-heap
1st Max	$\Theta(1)$	$\Theta(n)$

	Max-heap	Min-heap
1st Max	$\Theta(1)$	$\Theta(n)$
nth Max	$\Theta(n^2)$	$\Theta(n^2)$
1st Min	$\Theta(n)$	$\Theta(1)$
nth Min	$\Theta(n^2)$	$\Theta(n^2)$

selection / insertion sort n-times

can be found in leaf nodes $\Theta(n/2) = \Theta(n)$ (selection / insertion sort)

★ Q Maxheap with 1023 nodes
Comparisons needed to find minimum element.

$k = \log_2(1023+1)$
$k = 10$
$h = 9$

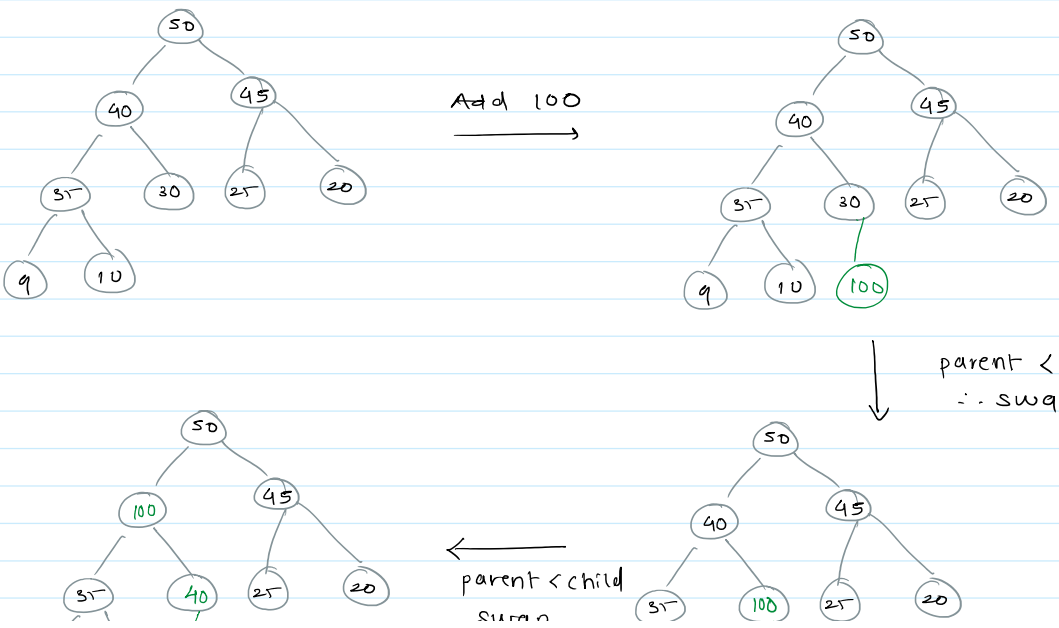
In Max-heap Minimum element can be found in leaf nodes

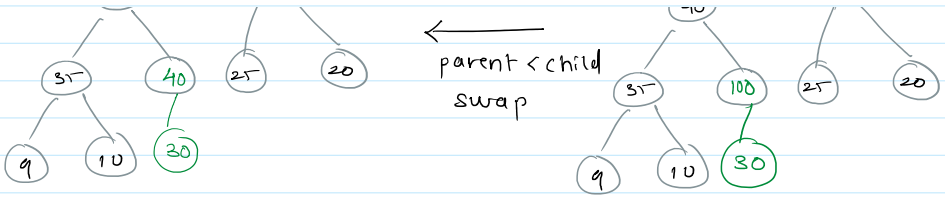
$\therefore$ leaf nodes $= \left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{1023}{2} \right\rceil = 512$

By applying Bubble sort for 512 elements
comparisons $= 512 - 1 = 511$

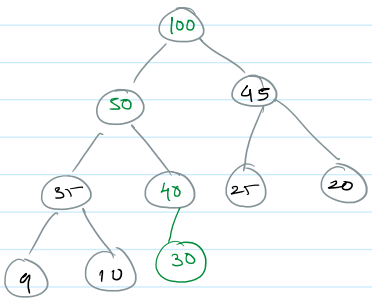
- Insertion:

1. Max heapify
   - Add element to the end of array of max heap.
   - check, if parent < child
     swap (parent, child)
     repeat until parent > child or parent == root.





parent < child
swap



∴ In worst case

$T(n) = \Theta(\log n)$ ← Added element reaches to root

In Best case

$T(n) = \Theta(1)$ ← child < parent, At time of adding.

— Psuedocode

```
max_heapify ( arr[] , element ) {
        arr[n] = element
        parent = find_parent(n)
        child = n
        while( arr[child] > arr[parent]
                    && child != 1)        ← root
            swap ( parent, child)
            child = parent
            parent = find_parent( parent)
        }
}
```

Deletion from a max heap.

- Replace root by last node
- check,

```
if parent < rightchild > leftchild
        swap ( parent, righchild )
        parent = rightchild.
else if parent < leftchild > rightchild
        swap ( parent, leftchild)
        parent = left child
```

- repeat until no further child

For max_heapify

$T(n) = 1$

$= 2 \log n$ ← 1 swap + 1 comparison

$T(n) = \Omega(1)$

$= O(\log n)$

For deletion

$T(n) = \Omega(1)$ ← 1 swap + 2 comparisons

$= O(3 \log n) = O(\log n)$

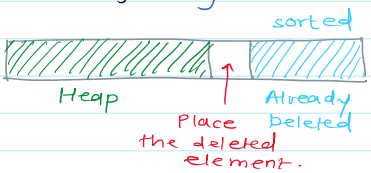
In Max heap.

In Max heap.

Finding max element = $O(1)$

Deleting max element = $O(\log n)$, $\Omega(1)$]

- Heap sort.
  - Delete max from max heap or in general delete root and append in the blank space remained after adjusting.



  - After deleting all we will get sorted array.

If Max heap
Every time max is deleted and appended Ascending order

If min heap
min is deleted and appended Descending order.

- Heap sort is not stable

- Build heap

Creating max / min heap using $O(n)$ time complexity for every case

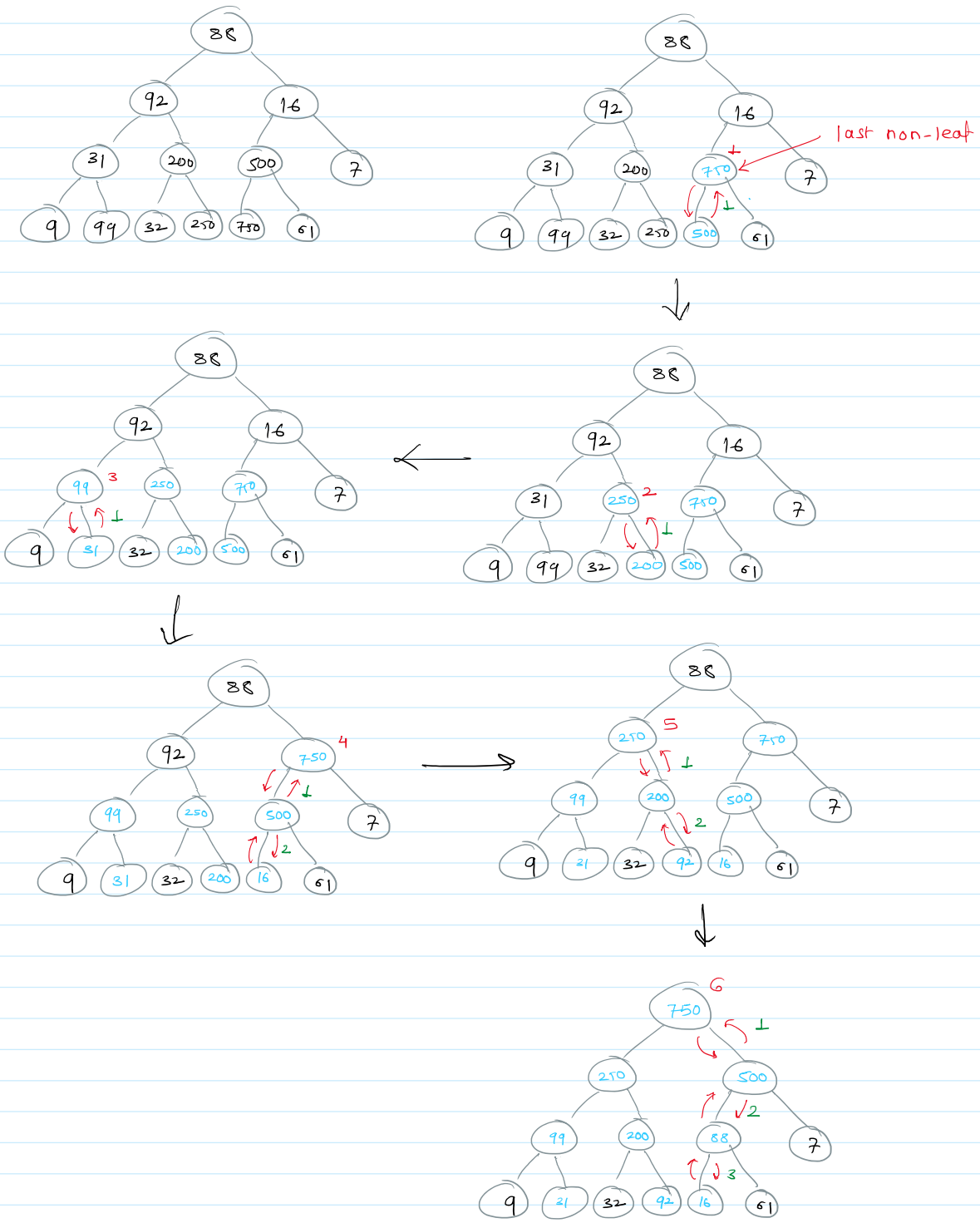
Input: Array of n-elements
Output: Max / min heap.

Algorithm:

```
build_heap ( arr[], n) {
    for( i = ⌊n/2⌋ , i ≥ 1 ; i--) {
        max_heapify_parent ( arr , i)
    }
}

max_heapify_parent ( arr[], i) {
    while ( parent ≥ null ) {
        if parent < lchild > rchild
            swap ( parent , lchild)
            parent = lchild
        if parent < rchild > lchild
            swap (parent , rchild)
            parent = rchild.
    }
}
```

Ex. 88 92 16 31 200 500 7 9 99 32 250 750 61



$T(n) = O(n)$ ← for proof refer cormen

- Heapsort Algorithm:
  1. Create Max heap using buildheap → $O(n)$
  2. Delete one by one and store to end of last node → $O(n \log n)$

$\therefore$ $T(n) = \Theta(n \log n)$

For deleting one element

Best case = $\Theta(1)$ ← very rare

Worst case = $\Theta(\log n)$

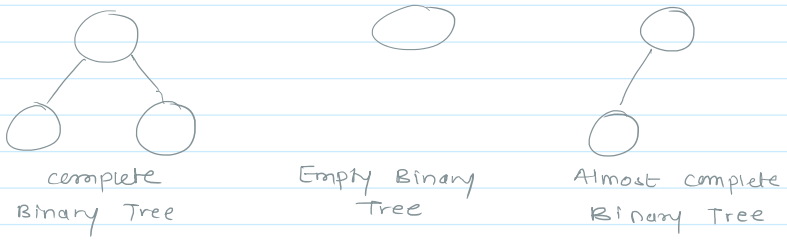
$\therefore$ Overall deletions = $\Theta(n \log n)$

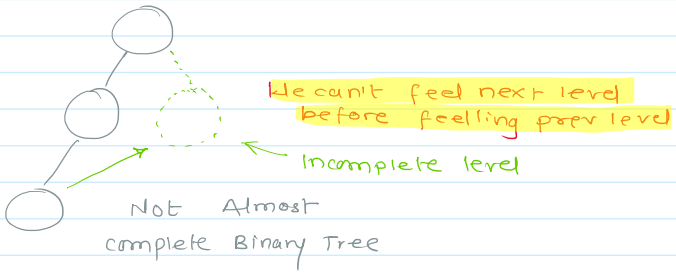
For same elements array = $\Theta(n)$

Binary Tree

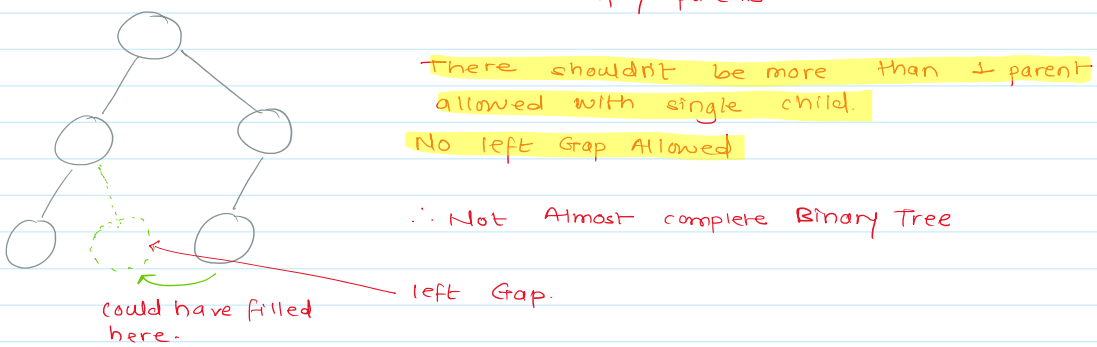
13 April 2021 10:03 AM

- Almost complete binary tree needed.

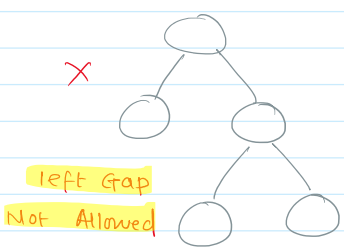


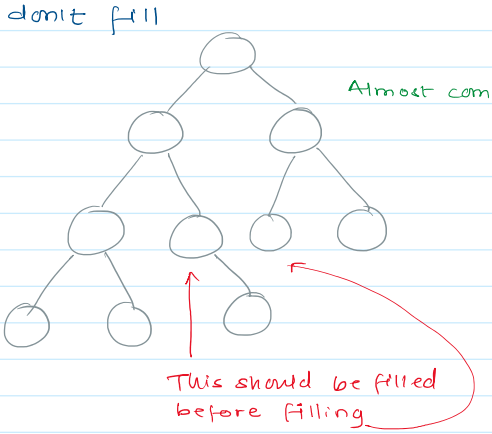


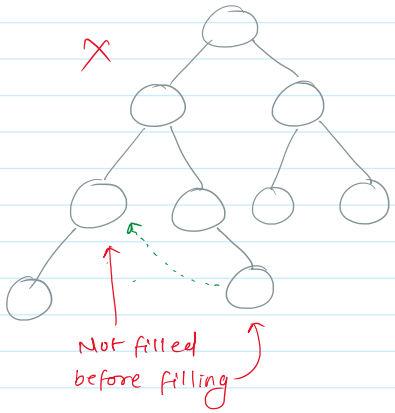
∴ Almost Complete Binary Tree
∵ only one right binary node
& others are empty parents



- Almost comple binary tree
  1. No prev level should be unfilled
  2. Before completing left, don't fill right.

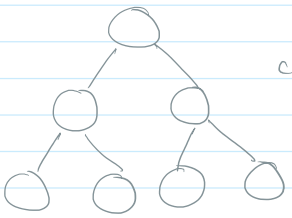




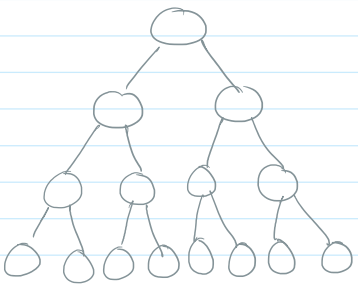


Not Almost complete Binary Tree.

- Complete Binary Tree :
  Completely filled binary tree without any gaps



complete Binary Tree



	level	height
$1 = 2^0$ ←	1	0
$2 = 2^1$ ←	2	1
$4 = 2^2$ ←	3	2
$8 = 2^3$ ←	4	3

① ∴ Nodes at each level = 2^h
where h is height of tree considering root at height 0

Imp ⇒ Total levels = h+1
$k = h+1$

② Total nodes = $2^{h+1} - 1$ = n

③ Non-leaf Nodes = $2^{h-1} - 1$ ← CBT
= $\lfloor \frac{n}{2} \rfloor$ ← floor
} CBT/ACBT

④ Leaf nodes = 2^h ← CBT
= $\lceil \frac{n}{2} \rceil$ ← ceil
} CBT/ACBT

⑤ Total levels = $\log_2(n+1)$ ← ACBT & CBT

$k = \log_2(n+1)$

$k = h+1$

Q. $n = 1023$
levels = $k = \log_2(1023+1)$
$k = 10$

Q. $n = 1023$

levels $= K = \log_2(1023+1)$

$K = 10$

$\therefore h = 9$

Nm - leaf nodes $= \left\lfloor \frac{1023}{2} \right\rfloor$

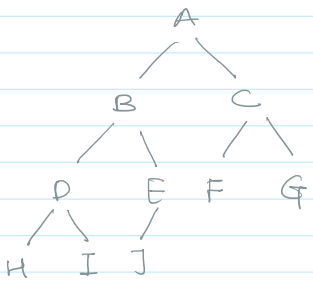
$= 511$

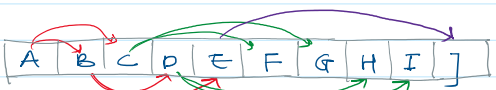
leaf nodes $= \left\lceil \frac{1023}{2} \right\rceil$

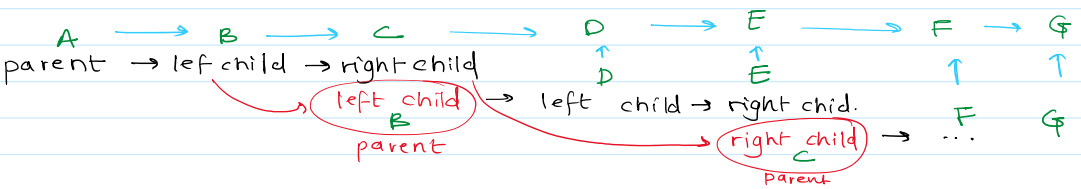
$= 512$

- Binary Tree Representation
  1. Array Format
  2. Linked List format

1. Array Format







parent Index $= \left\lfloor \frac{\text{child Index}}{2} \right\rfloor$

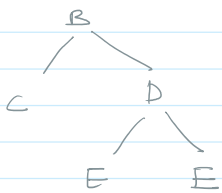
child Index $= 2 \cdot \text{parent Index}$ ← left child

$(2 \cdot \text{parent Index}) + 1$ ← right child

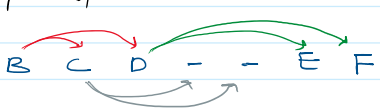
For Array index starting from 0.

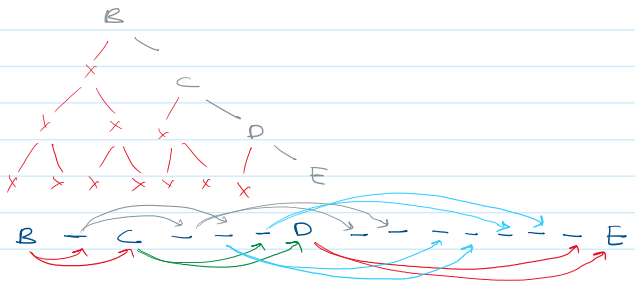
If child index crosses the array length
or child index has value Null
Represents parent is leaf Node

Q

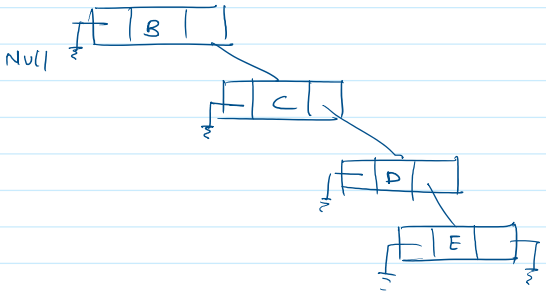


Array Representation





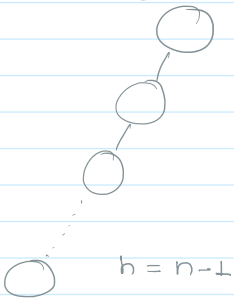
Linked list



Array representation: For ACBT & CBT Tree
Linked List : Other than ACBT / CBT

Q If n-nodes Binary tree represented by array state max and min size of array to store

→ Max size when



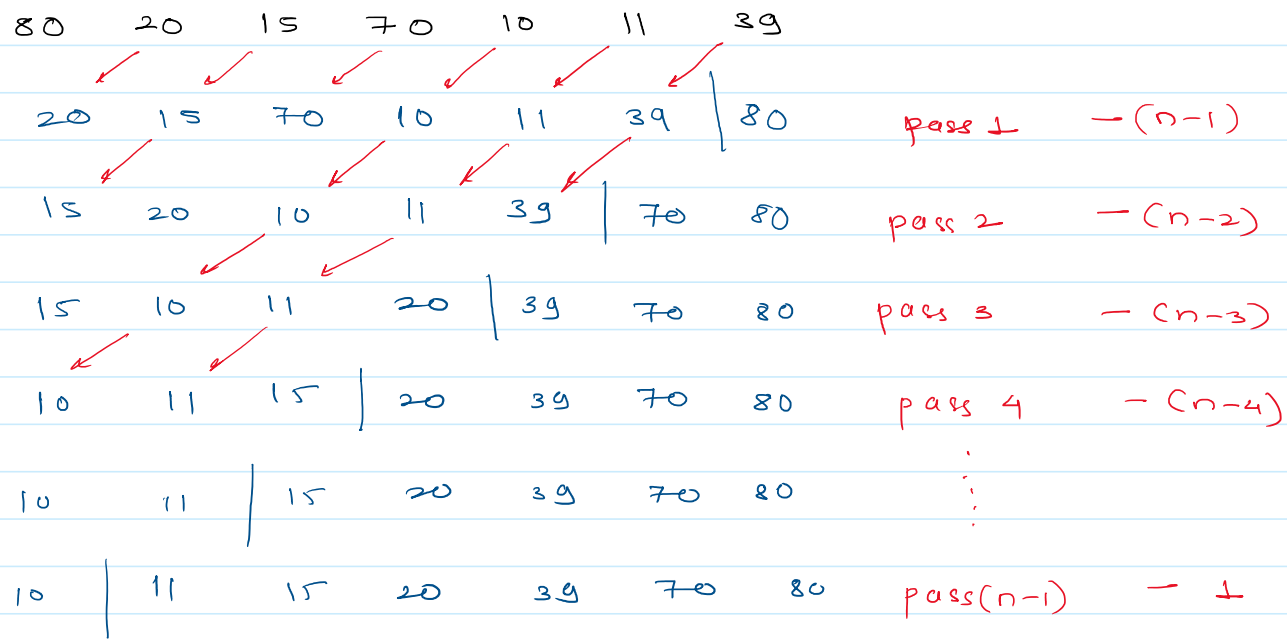
Total elements in CBT = $2^{h+1}-1$
2^{h+1} This much size should be reserved
∴ Size required = $2^{h+1}-1$
$= 2^{n-1+1}-1$
$= 2^n - 1$ ← Max

Min size of array required if complete binary Tree
∴ n-nodes stored in n-size array ← Min

Bubble Sort

24 April 2021 17:53

- swap if (a[i] < a[i+1])
- At each pass max element is placed at the right position.



① n-1 passes required to sort

② Total comparisons = $(n-1)+(n-2)+\cdots 1$

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

③ For swaps = min 0 $= \Omega(1)$ ← sorted Array

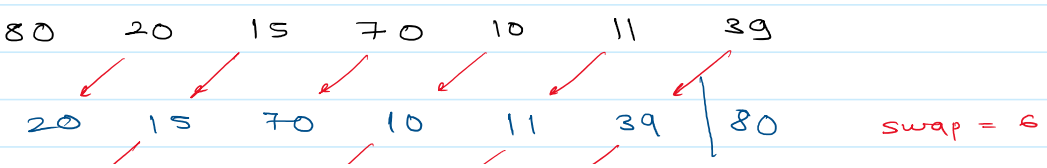
= max $\frac{(n-1)(n)}{2} = O(n^2)$

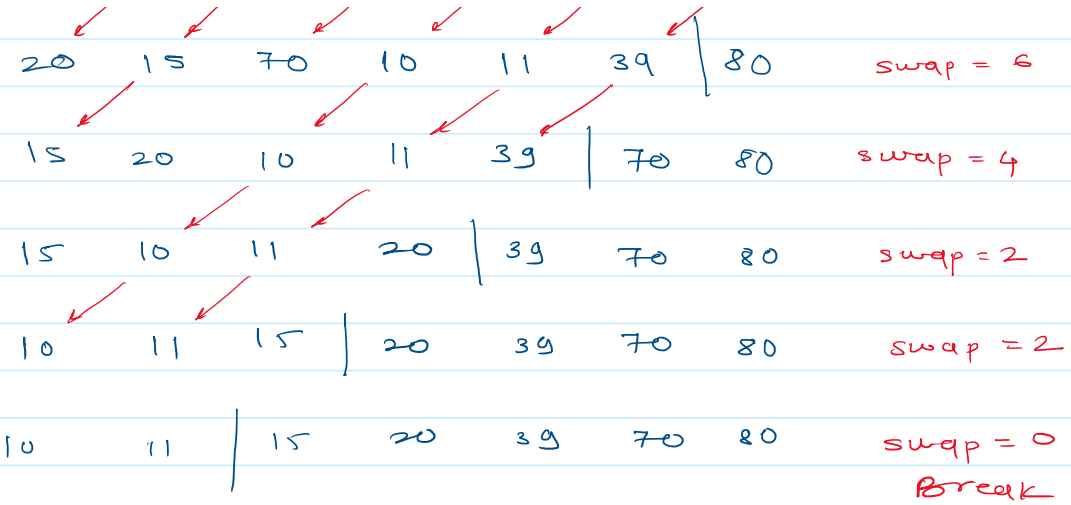
$\therefore T_c = \underset{\text{comparisons}}{O(n^2)} + \underset{\text{swaps}}{\begin{matrix}\Omega(1)\\ O(n^2)\end{matrix}}$

$T_c = \begin{matrix}\Omega(n^2)\\ O(n^2)\end{matrix} = \Theta(n^2)$

④ In-place

⑤ Stable





Swap = 0 Array Sorted

With swap variable

Best case : $\Theta(n)$

Worst case : $\Theta(n^2)$

$\therefore T_C = \Omega(n)$

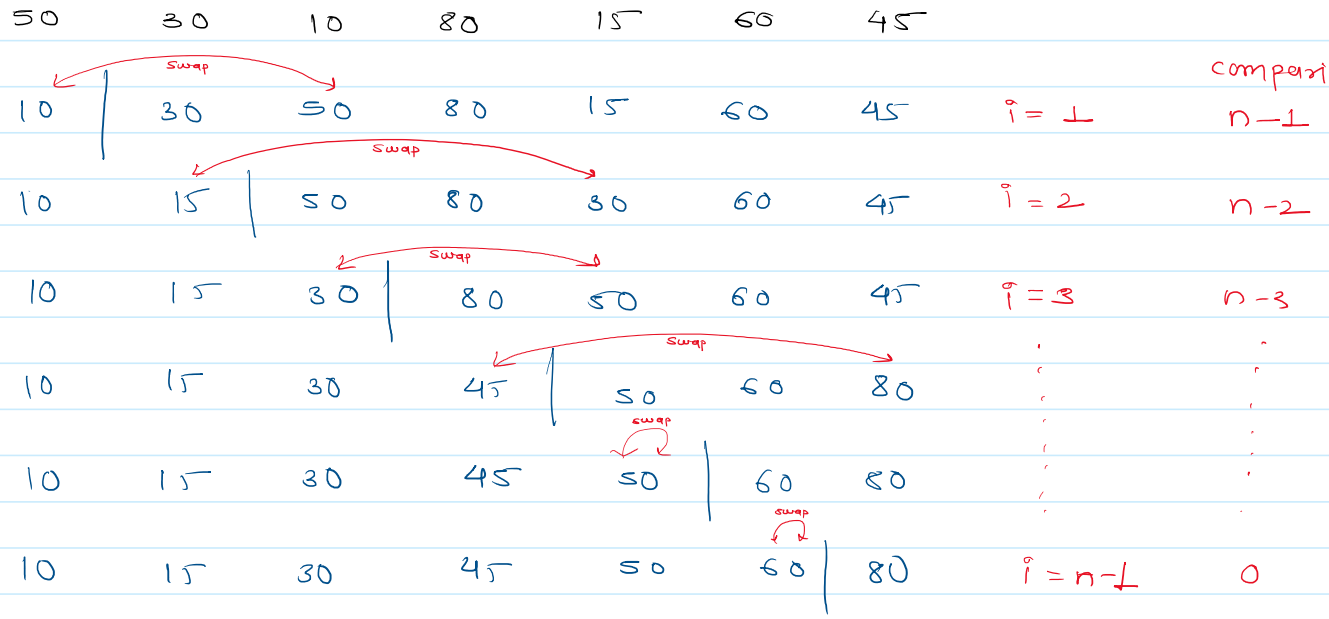
$O(n^2)$

Selection Sort

24 April 2021 18:49

Find min and swap a[i] and a[min]

Initially i = 1 , i++ after every pass



$$\text{comparisons} = (n-1) + (n-2) + (n-3) + \cdots$$
$$= \frac{(n-1)(n)}{2}$$
$$= \Theta(n^2)$$

$$\text{swaps} = n-1 = \Theta(n)$$

① $\therefore T_C = \Theta(n) + \Theta(n^2)$
$= \Theta(n^2)$

② Unstable

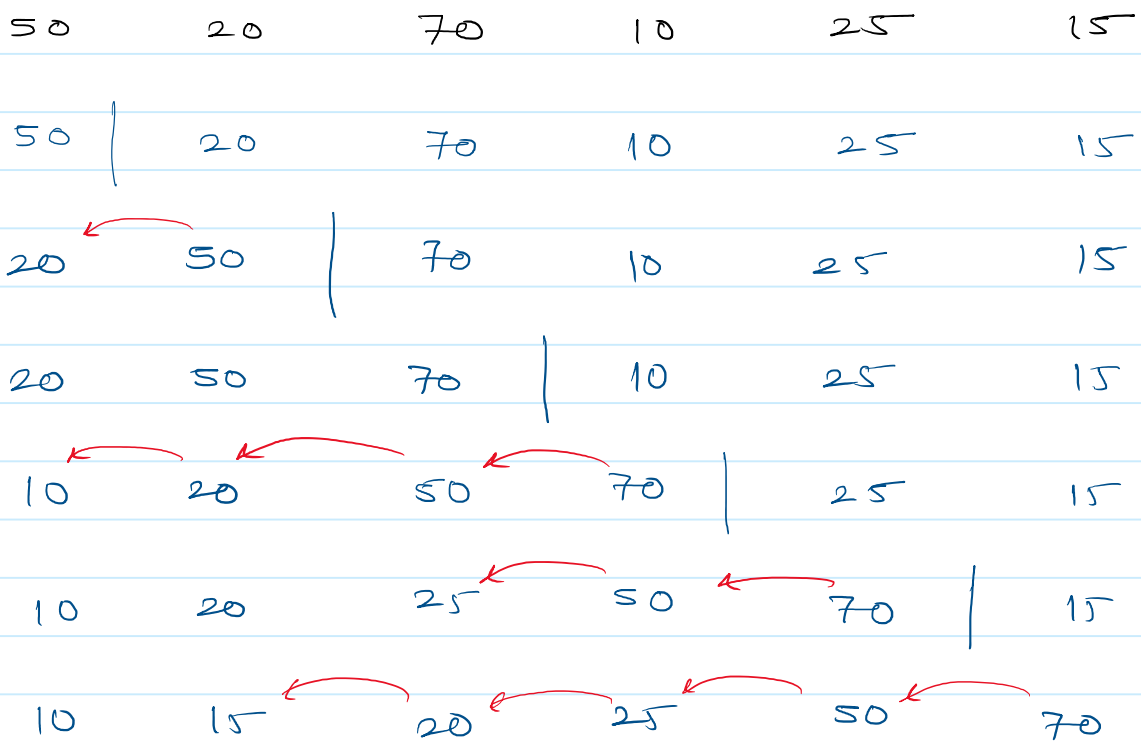
③ Inplace

Insertion Sort

24 April 2021 19:24

Number of swaps = Number of inversion

slowing increase the size of array and place the new inserted element to right place by swaping.



passes $n-1$

Worst case swaps = $1+2+3+4+\cdots(n-1)$

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

Best case swaps = $0 = \Theta(1)$

Comparisions

Best case = $n-1$

worst case = $1+2+3\cdots(n-1)$

$= \Theta(n^2)$

$\therefore T_C = \underset{\Theta(n^2)}{\underset{\text{comparisions}}{\Omega(n)}} + \underset{\Theta(n^2)}{\underset{\text{swaps}}{\Omega(1)}}$

$T_c = \Omega(n) = O(n^2)$

1. Inplace
2. Stable
3. Best for Almost sorted array $O(n)$
   $n-1$ comparisions
   0 swaps

If array contains atmost n inversions
Then array is almost sorted
$\therefore$ T_c Insertion sort $= O(n)$

Greedy Techniques

14 April 2021 18:54

Min - 2 Marks
Max - 4 Marks
Mostly Numerical

Q. Find top 10 students out of 300 peole whose average is more than others
Basic criteria they should have more than 80 %.

- Solution Space
  Set of all posible solution for given problem is known as solution space

Here solution space is every combination of 10 students out of 300.
Total = $^{300}C_{10}$

- Feasible solution
  Solutions from solution space which satisfies the basic criteria.
  ∴ Selecting students withe > 80%

- Optimal solution
  - Optimal solution is one of the feasible solution which maximizes our goal / best solution.
  - Optimal solution need not be unique.

Note : Most of the problems in greedy technique contain n - number of inputs and our objective is finding solution space

Greedy Technique is shortcut to find optimal solution.

Applications:

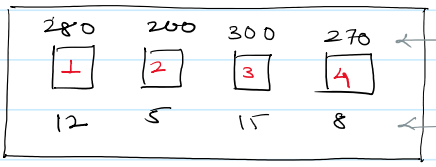
1. Knapsack Problem
2. Job Sequencing with deadlines
3. Huffman Coding
4. Optimal Merge Pattern
5. Minimum cost spanning tree
    a. Prims
    b. Kruskal
6. Single Source Shortest Path
    a. Dijkstra's Algorithm
    b. Bellman-form Algorithm
    c. Breadth First Traversal

1. Knapsack Problem

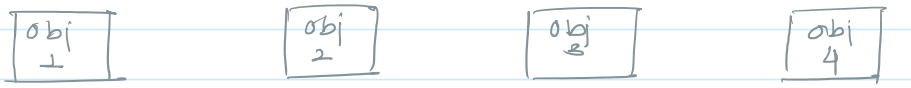
- Fractional knapsack
- Greedy knapsack
- Real knapsack

14 April 2021 09:23 PM

Q



A thief want to steal from a house which has 4 objects as shown in figure. But he has a knapsack which can hold weight only 30. Help him select the objects for maximum profit. Fraction of object is allowed.



$\frac{280}{12} = 23.3$ $\frac{200}{5} = 40$ $\frac{300}{15} = 20$ $\frac{270}{8} = 33.33$ ← profit to weight ratio

Space remained	Profit	
30	0	
$30 - 5 = 25$	200	← 1 unit of obj. 2
$25 - 12 = 13$	480	← 1 unit of obj. 4
$13 - 8 = 5$	750	← 1 unit of obj. 1
$5 - \frac{5}{15} 15 = 0$	850	← $\frac{1}{3}$ unit of obj. 3

Algorithm

1. Find profit : weight ratio
2. Arrange in Descending order
3. Select one-by-one from left

Q $n = 7$ capacity of knapsack = 17

objects	1	2	3	4	5	6	7
profit	25	55	100	15	95	65	35
weights	2	4	3	2	5	4	3
p/w	12.5	13.75	33.3	7.5	19	16.25	11.75

objects

sort$_{desc}$:	33.3	19	16.25	13.75	12.5	11.75	7.5
objects :	3	5	6	2	1	7	4
	1	1	1	1	1/2	0	0

← O(n log n)

knapsack | profit

1 1 1 1 1/2 0 0

knapsack	profit			
17	0			Obj
14	100	←	1 →	3
9	195	←	1 →	5
5	260	←	1 →	6
1	315	←	1 →	2
0	327.5	←	1/2 →	1

One loop required
$= O(n)$

$\therefore T(n) = n\log n + n$
$= \Theta(n\log n)$

Feasible solution infinite for fractional Greedy

Q Worst case time complexity of knapsack if you use Quicksort, mergesort.

① Quicksort:

Worst case $= \Theta(n^2)$

$\therefore T(n) = n^2 + n + n$ ← Finding soln.
↑ sorting ↑ PI ratio calculation

$= \Theta(n^2)$

② Mergesort:

Worst case: $\Theta(n\log n)$

$\therefore T(n) = \Theta(n\log n)$

Q. Time complexity of knapsack if elements already sorted

Elements already sorted.

No need of sorting.

$\therefore T(n) = n + n$
$= \Theta(n)$

2. Job Sequencing with Deadlines

15 April 2021 09:00 AM

Conditions:

1. Single CPU (No parallel processing)
2. No preemptions (you can't keep process unfinished)
3. All jobs running time 1 unit each (No Shortest Job first)
4. All jobs arrival time same (No First come First serve)

Q n = 4

Jobs	J_1	J_2	J_3	J_4
profit	80	200	150	70
Deadline	2	1	2	1

← Every job requires 1 unit time. Deadline represents when you can finish the job before expiring.

Here only two deadlines possible. Separate the jobs according to deadline.

To be done in 1st unit: consider Here | Not selected | To be done in 2nd unit

1			2
J_2	J_4	J_1	J_3
200	70	80	150

Choose the one with max profit. ← from more deadlines to less.

∴ Max profit = $J_2(200) + J_3(150)$
= 350

left out Jobs = $J_1 + J_4$
lost profit = 80 + 70
= 150 ← penalty.

Q

Jobs	J_1	J_2	J_3	J_4	J_5	J_6	J_7
profit	80	20	150	25	95	15	70
Deadline	5	3	1	4	3	2	4

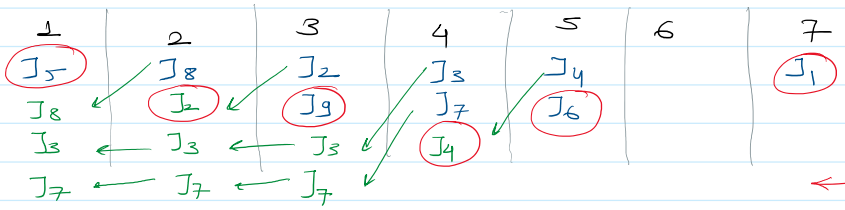
1	2	3	4	5	← deadlines
J_6 ← J_3 ← J_2	J_4	J_4 ← J_5 ← J_2	J_4 ← J_7	J_1	← selected as per max profit.

= 150 + 25 + 95 + 70 + 80
= 420 profit

penalty = $J_6 + J_2$ ← Jobs lost
= 15 + 20
= 35

Q.

Jobs	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9
profit	55	77	11	39	65	95	25	10	88
Deadline	7	3	4	5	1	5	4	2	3



← Sequence of doing jobs can be different if more jobs have same deadlines.

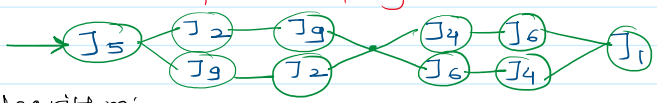
$= 65 + 77 + 88 + 39 + 95 + 0 + 55$
$= 419$

Penalty = $J_8 + J_3 + J_7$
= 46

Order:

J_5 J_2 J_9 J_4 J_6 J_1

Can be swapped because they have same deadlines and can be fulfilled after swapping too



Follow Any path

Algorithm:

① Sort Jobs on the basis of profits.

J_6	J_9	J_2	J_5	J_1	J_4	J_7	J_3	J_8
95	88	77	65	55	39	25	11	10
5	3	3	1	7	5	4	4	2

② Find max deadline = 7
③ Find first occurance of 7
Add to profit and mark as finished
④ n--; repeat above step again untill you get n=1

$\therefore T(n) = \Omega(n \log n)$
$= O(n^2)$

3. Optimal Merge Pattern

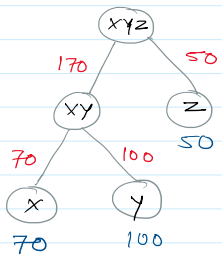
15 April 2021 17:36

No condition that it should form BST as in Huffman coding.

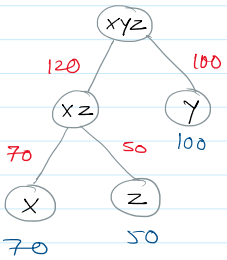
Merging n files with sorted records to form a single file.

Q. n = 3 files.
X = 70
Y = 100
Z = 50
Find optimal merge pattern.

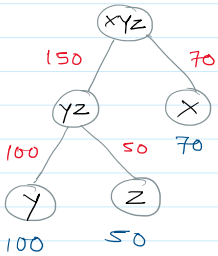
→ Solution Space Two Way Optimal Merge → At a time two files only.



= 170 + 50 + 70 + 100
= 390



= 70 + 50 + 120 + 100
= 340
Optimal solution



= 100 + 50 + 70 + 150
= 370

← Moves required to merge files.

← Moves at lowest level have to be moved again at upper levels.
∴ Try merging low records at lower levels.

← Total moves required to merge.

Q n = 7

A	B	C	D	E	F	G
5	6	50	90	20	3	7

① Sort the files according to number of records

F	A	B	G	E	C	D
3	5	6	7	20	50	90

② Merge two best
F & A.

AF	B	G	E	C	D
8	6	7	20	50	90

From above B & G Best

AF	BG	E	C	D
8	13	20	50	90

From above FA & BG Best

ABFG	E	C	D
21	20	50	90

From above ABFG & E Best.

ABEFG	C	D
41	50	90

From above ABEFG & C best.

ABCEFG	D
91	90

Merge remaining.

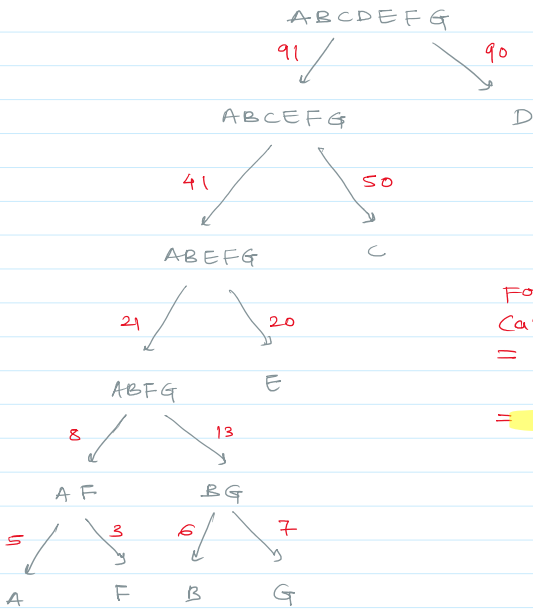
Sorting done using selection sort

91 90

Merge remaining.

ABCDEFG
181

Sorting done using selection sort
For every pass 2n
$T(n) = 2n.n = \theta(n^2)$
By selection sort



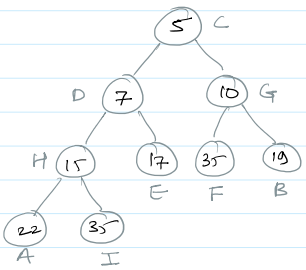
For optimal merge pattern
Calculate the weight of middle nodes.
= 91 + 90 + 41 + 50 + 21 + 20 + 8 + 13
+ 5 + 3 + 6 + 7
= 355

Q. n = 9

A	B	C	D	E	F	G	H	I
22	19	5	7	17	35	10	15	35

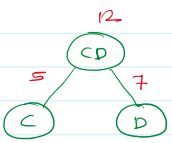
→

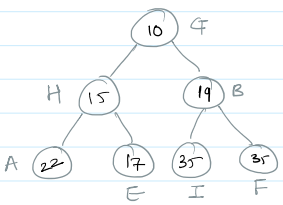
① Create Min heap ($T(n) = \theta(n)$)



② Take out two elements from min-heap

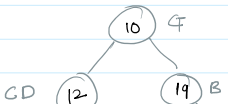
min_1 = 5, C
min_2 = 7, D



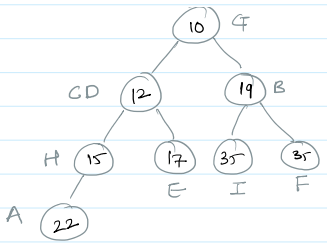


For two deletions
= logn + logn
= O(logn)

Add CD = 12 to min heap again.



For insertion of 12 (CD)
= O(logn)

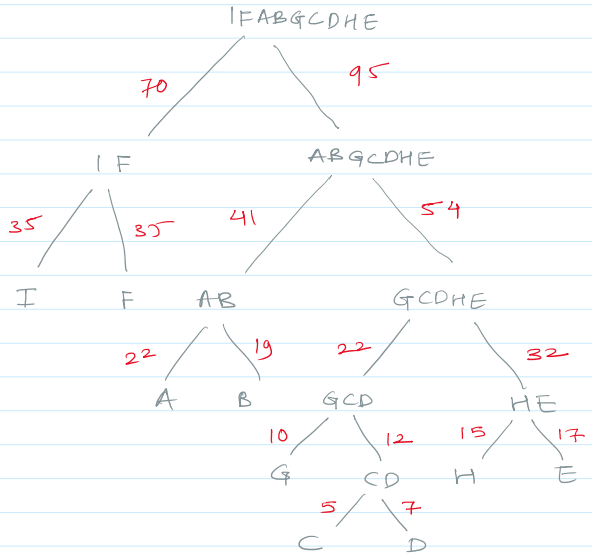


For insertion of 12 (CD)
= $O(\log n)$

Total $3 \log n$ for this step.
$T(n) = \Theta(n)$

③ Repeat above procedure until one element is left in heap.

$\therefore T(n) = n + (n-1) \cdot 3 \cdot \log n$
$= n + 3n \log n - 3 \log n$
$T(n) = \Theta(n \log n)$ ← using min heap.



$\therefore$ By calculating intermediate nodes
Total optimal moves = 491

Note:
If data structure is
Array : $T(n) = O(n^2)$
min-heap : $T(n) = \Theta(n \log n)$

If we want to find maximum moves create max-heap and combine two maximums

4. Huffman Coding

★ IMP. 5 times/10 years

15 April 2021 18:39

- Data Encoding Technique.
- Data compression Technique.
- Non-uniform coding
- More freq → less bits
  less freq → More bits

Q

A	=	35
B	=	12
C	=	45
D	=	5
E	=	3
F	=	4

Total char = 104

In ASCII every character will be stored as 8 bits

∴ Total bits needed = 816

Bits : Char = 8 bits/char

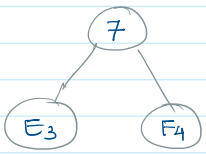
To convert them into Huffman code.

Algorithm:

① sort them according to frequencies in ascending order

E_3 F_4 D_5 B_{12} A_{35} C_{45}

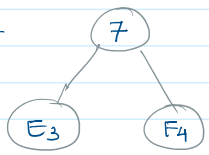
combine first two



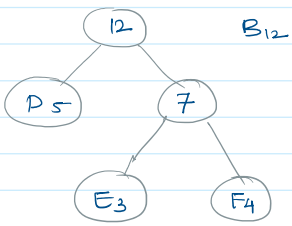
D_5 B_{12} A_{35} C_{45}

② Repeat sort and combine

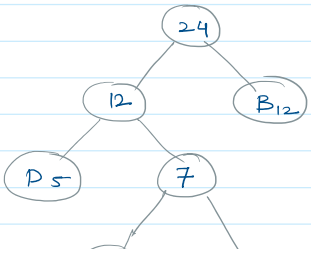
D_5



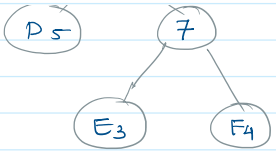
B_{12} A_{35} C_{45}

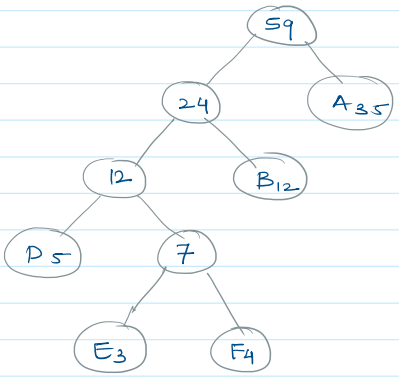


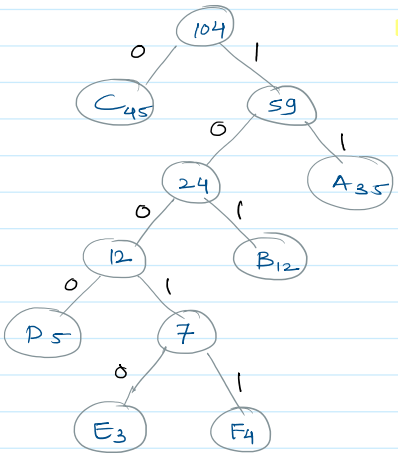
B_{12} A_{35} C_{45}



A_{35} C_{45}







After creating the Huffman Tree code every left node = 0
right node = 1
} Not a standard you can do viceversa

A	=	11
B	=	101
C	=	0
D	=	1000
E	=	10010
F	=	10011

} Huffman codes (encoding)

So, considering above representation and their frequencies.
Total bits = 199
∴ Bits : char = 199 : 104 = 1.913 ← Do not round to 0th decimal.

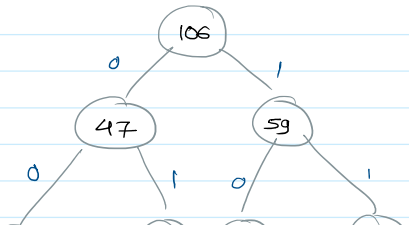
- Here, everytime two nodes are combined
Hence, also called as Two-way-Encoding

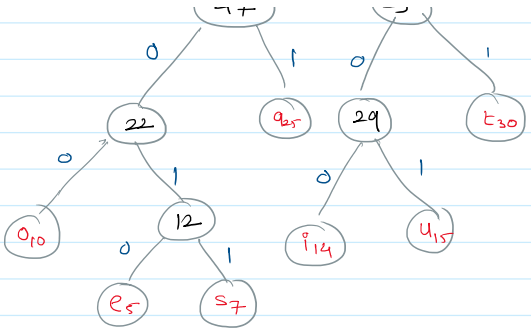
8 bits/char ⟹ 1.913 bits/char
≈ 80% compression
} compression

Q m = (a, e, i, o, u, s, t)

a	=	25
e	=	5
i	=	14
o	=	10
u	=	15
s	=	7
t	=	30

Two-way encoded tree





Huffman codes

a	=	01
e	=	0010
i	=	100
o	=	000
u	=	101
s	=	0011
t	=	11

Bits : char ratio : 2.570 bits/char

Built min-heap

$T(n) = n + (n-1).3.\log n$ ← 2 deletion + 1 insertion (n-1) times

$= O(n \log n)$

$T(n) = O(n^2)$ ← In array.

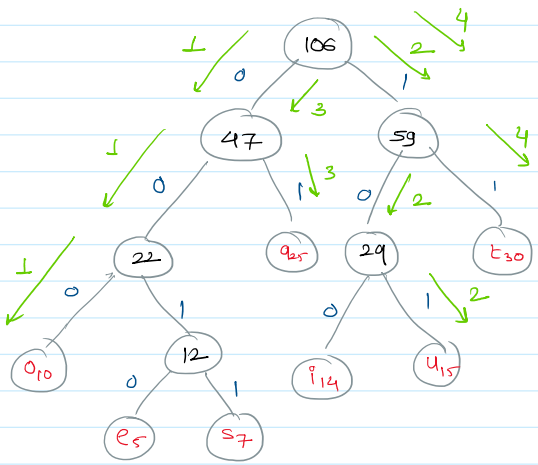
Trace the code from root

when leaf node come, again start from root

- Encoded message: 0 0 0 1 0 1 0 1 1 1 0 1 0 1 0 0 0 0 1 0

  decoded message: o u a t u a o e

  1 2 3 4 ...



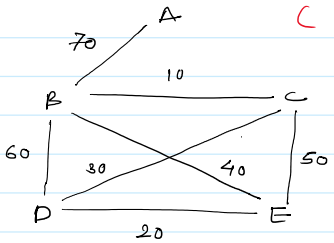
5. Minimum Cost Spanning Tree

18 April 2021 10:54

Every year 1 question (prims very Imp)

Minimum cost spanning tree is a spanning with lowest cost edges

① Kruskal Algorithm: (Always selects minimum edge) doesn't care about adjacent edges)
(Picks minimum edges to form connected graph without forming cycle)



n vertices ; E edges
n = 5 ; E = 7

① Build min-heap of edges
$T_c = O(E)$

② Delete minimum element and add to MST
$T_c = O(\log E)$

and make their parent same

③ Select minimum again, check if parents are different. i.e. no cycle
add to MST.

and make their parents same

④ Repeat above until V−1 times.

Best Case

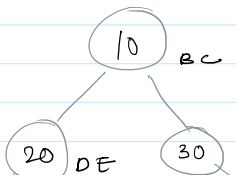
$T(n) = E + (V-1)\log E$ $\quad |E| = O(V^2)$
$= \Theta(V \log E)$ $\quad \log E = O(\log V)$
$= \Theta(V \log V)$

Worst case

$T(n) = E + E \log E$
$= \Theta(E \log E) = \Theta(E \log V)$

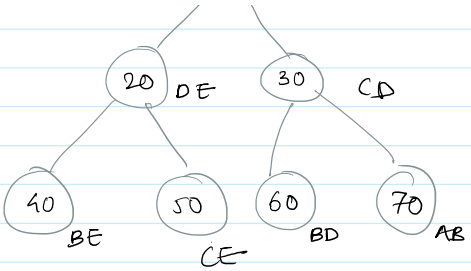
$\therefore T(n) = \Omega(V \log V) = \Omega(V \log E)$
$= O(E \log V) = \Omega(E \log E)$

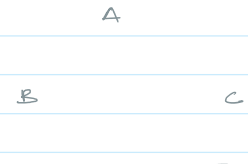
$E = V^2$ can be replaced



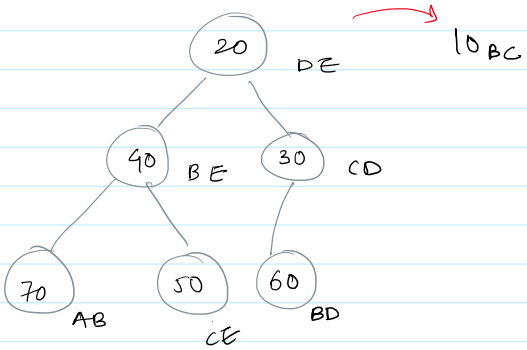
{A} {B} {C} {D} {E}

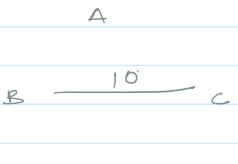
A



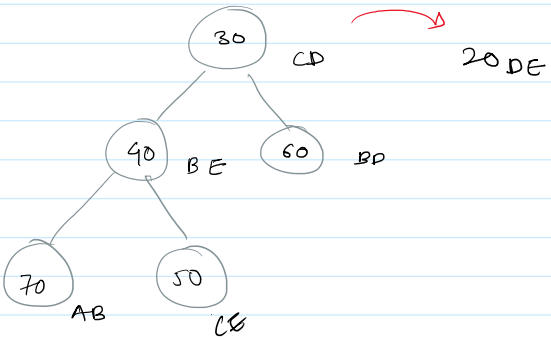


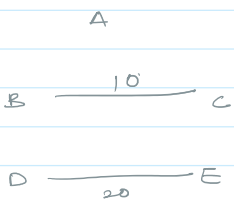
{A} {B,C} {D} {E}



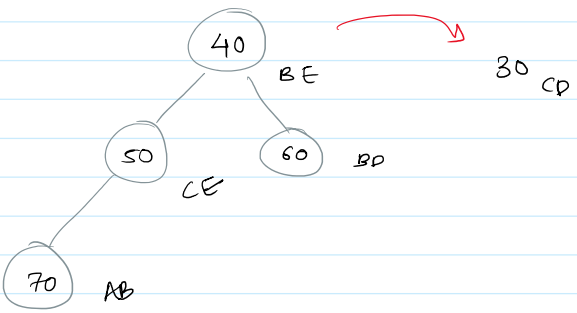


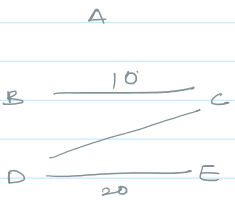
{A} {B,C} {D,E}

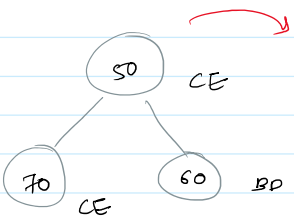


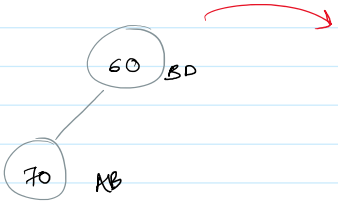


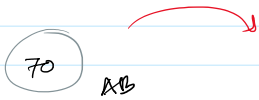
{A} {B,C,D,E}







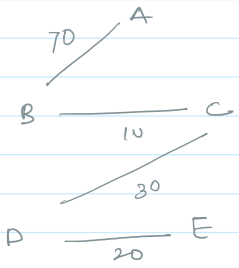




60_{BD} belongs to same group, discarded.

{ A B C D E }

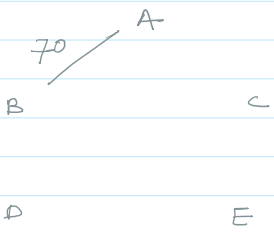
70_{CD}



② Dijkstra's Algorithm : (Travel's to adjacent vertices from previously visited vertices only from minimum path, avoiding cycle)

① select any vertex and find adjacent of that vertex and take minimum path

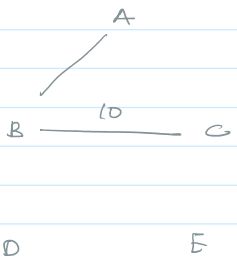
{ A, B } { C, D, E }



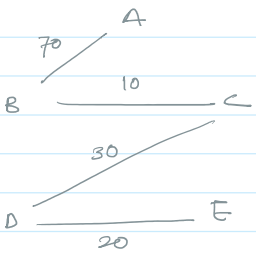
② select adjacent to A and B and select minimum edge

{ A, B, C } { D, E }

New edge should be between elements of two set.



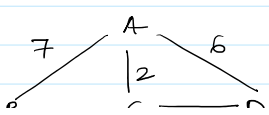
③ Repeat above step without forming cycle



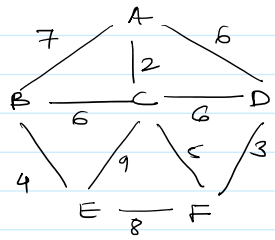
{ A, B, C, D, E } { }

Kruskal & Prims may / may not give same spanning tree.
More than one spanning tree can have same effective cost.

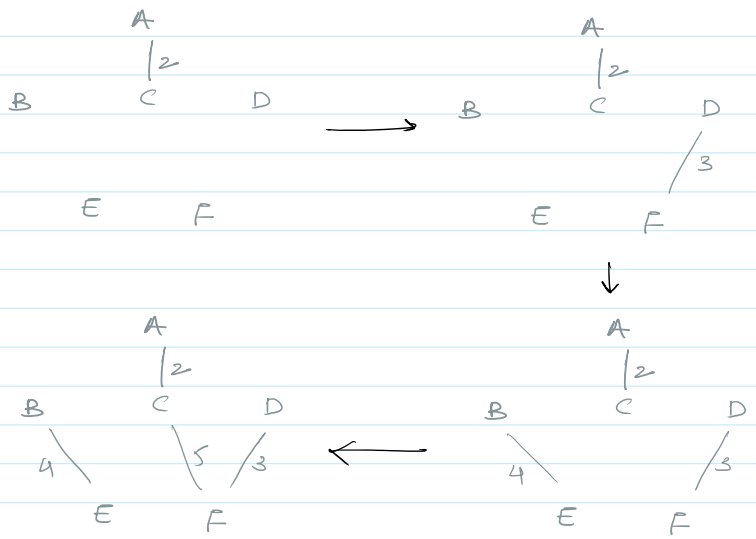
Q

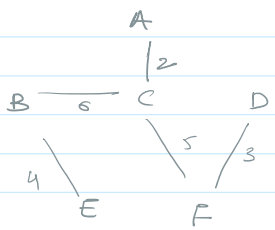


Q



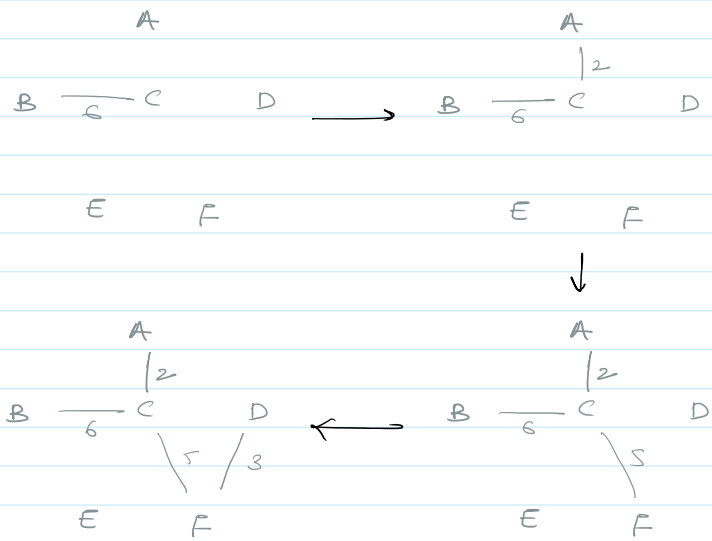
By kruskal

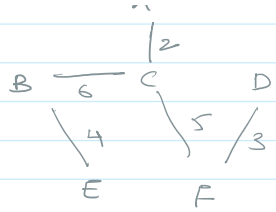




cost = 20

By prims





Note: In a graph,

weights distinct ⟶ Maximum 1 MST

weights repeating ⟶ May be more than 1 MST

Space complexity for Graph representation

Adjacency matrix representation:

Best case = $\Theta(V^2)$

Worst case = $\Theta(V^2)$

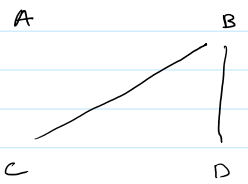
$S(V) = \Theta(V^2)$

Adjacency list representation

Best case = $\Omega(V)$

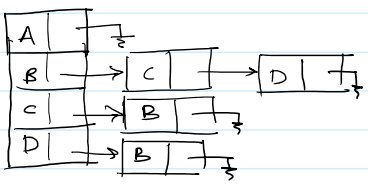
Worst case = $O(V+2E) = O(V^2)$

$S(V) = O(V+E)$



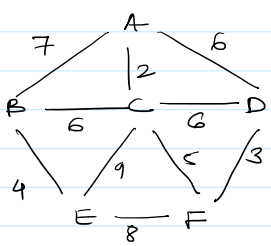
	A	B	C	D
A	0	0	0	0
B	0	0	1	1
C	0	1	0	0
D	0	1	0	0

Adjacency Matrix



Adjacency list

prisms Algorithm in min heap.



	A	B	C	D	E	F
Root of min heap	∞	∞	∞	∞	∞	∞
	-	—	—	—	—	—
A visited	A	7	2 (Root)	6	∞	∞
	—	A	A	A	—	—
C visited	A	6	2	6	9	5
	—	C	A	A	C	C
F visited	A	6	2	3	8	5
	—	C	A	F	F	C
D visited	A	6	2	3	8	5
	—	C	A	F	F	C
B visited	A	6	2	3	4	5
	—	C	A	F	B	C
E visited	A	6	2	3	4	5
	—	C	A	F	B	C

← distances from A

↓ Updated because less distance from C

Degree of visited element

$3 + 3\log V$

1 deletion + 2 Adjustment in min heap

$5 + 4\log V$

5 → Degree; 4 → 1 deletion + 3 updation

$3 + 3\log V$

$3 + \log V$

$3 + \log V$

$3 + 0\log V$

$TC = V + 2E + V\log V + E\log V$

- V → Build min heap
- $2E$ → comparing lengths (Sum of degree)
- $V\log V$ → Deleting all elements
- $E\log V$ → Adjusting min heap

$T_n = O[(V+E)\log V]$

If connected graph

$T_n = O[E\log V]$

If complete graph

$T_n = O[V^2\log V]$

If Adjacency matrix used & min heap

$T_n = O(V^2\log E)$

Using Adjacency list/matrix with unsorted Array

$T_n = O(V^2)$

Graphs

16 April 2021 07:05 PM

Input : Graph $G(V, E)$

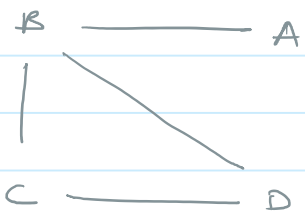
E : set of edges

V : set of vertices

Ex. $G(V, E)$

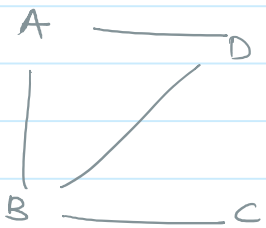
$V = \{A, B, C, D\}$

$E = \{AB, BC, CD, BD\}$



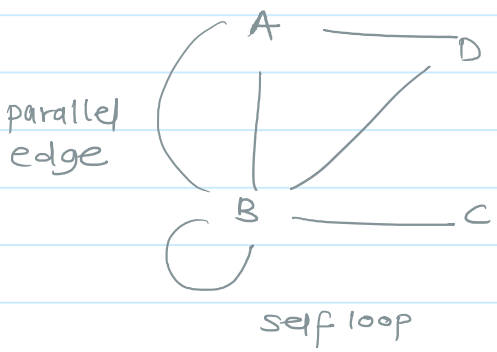
- Types of Graph

Simple Graph



loop	✓
parallel edge	✗
self loop	✗

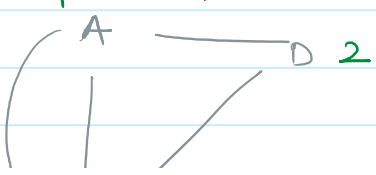
Multigraph



loop	✓
parallel edge	✓
self loop	✓

- Degree of a vertex

4 ← parrallel edge also adder



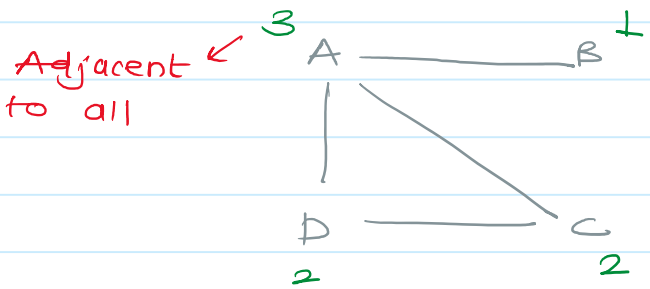
Number of edges connected to a vertex is the degree of that vertex

if there are two edges to same vertex, it is counted again

degree (r) = Numbe of edges

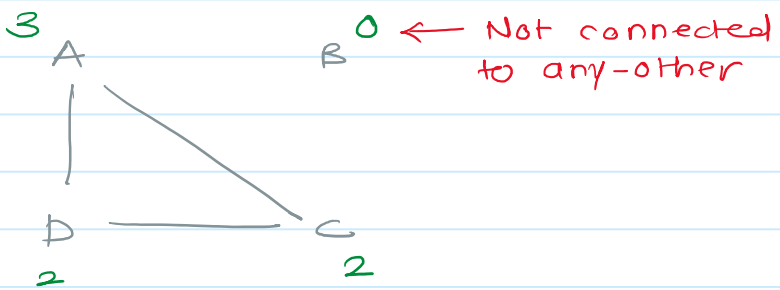
Q. Simple graph with n-vertices

(1) Maximum degree



∴ Max degree = n-1

(2) Minimum degree

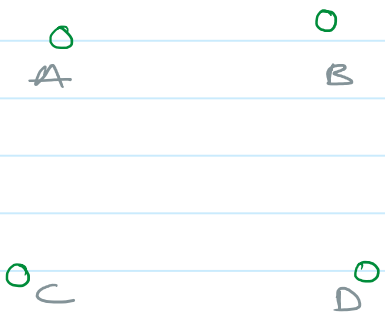


Minimum degree = 0

- For multigraph
  Minimum degree - 0
  Maximum degree = ∞

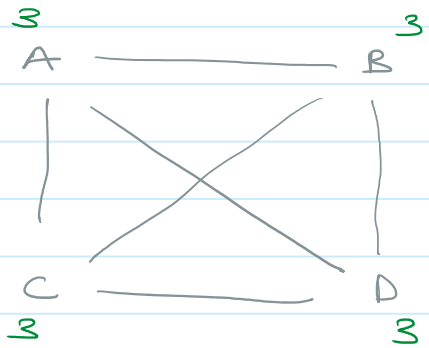
- Simple Graph.

① Null graph:



Every vertex with degree = 0

② Complete Graph



Every vertex with maximum degree $(n-1) = 3$

- Representation: K_n ← n-vertices

$$\sum_{i=1}^{n} \text{degree}(v_i) = 2 \cdot e$$ → twice the edges

Each degree adds 2 degrees

$$|E| \leq \frac{V(V-1)}{2}$$

$$|E| = O(V^2)$$

Applying log

$\log E = O(\log V)$ ← Also correct

Spanning Tree

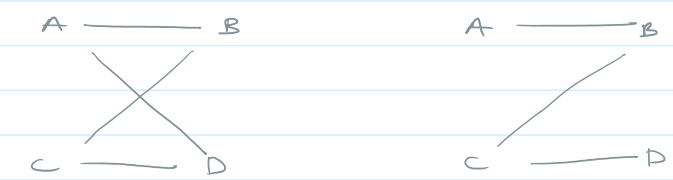
17 April 2021 06:48 PM

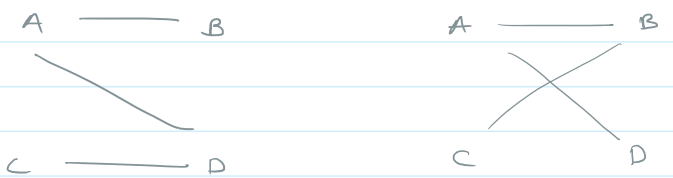
Refer Graphs First

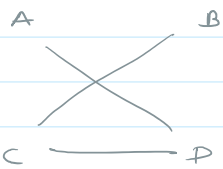
A subgraph S of the given graph G is said to be spanning tree iff

1. Vertices of S = Vertices of G
2. $|E| = V - 1$
3. No cyclic path

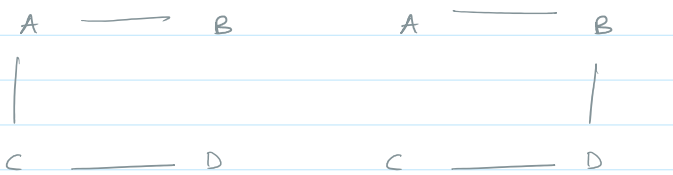
Q. Find number of spanning trees

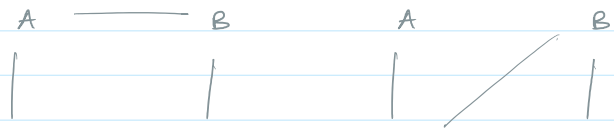


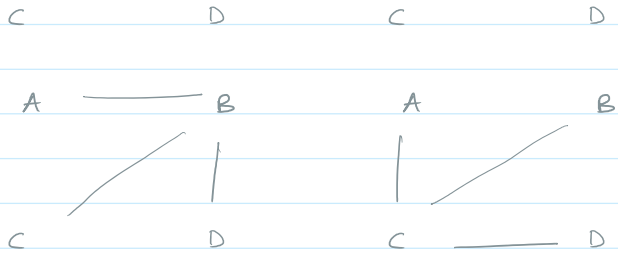


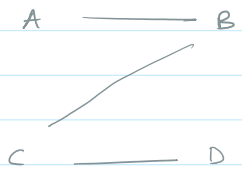


2.







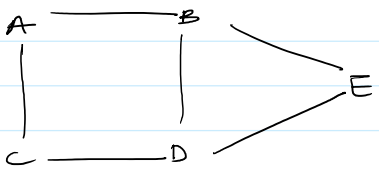


Shortcut = total edges $C_{(V-1)}$ − (find all cycles of length 3 to V-1)

$= {}^{5}C_{3} - 2$

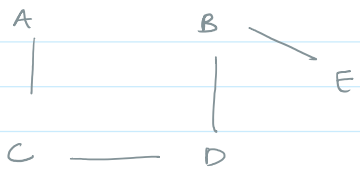
$= 8$

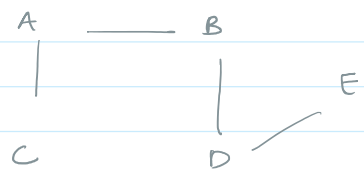
③

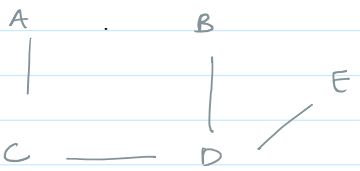


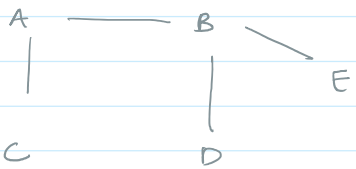
But don't use

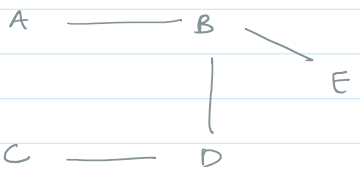
$= 5$ ← rotate edges

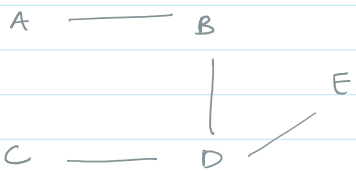




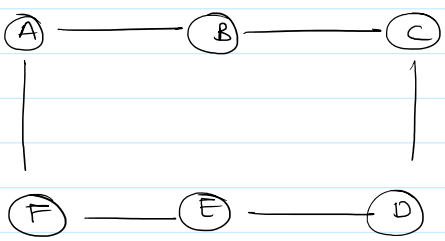








④




A
B
C
= 6
Rotate
F
E
D

6. Single source shortest path

22 April 2021 05:45 PM

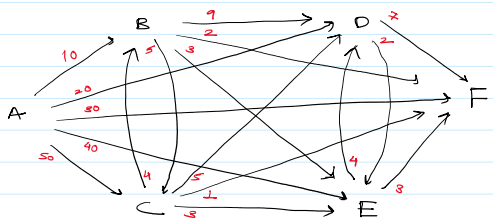
- Dijkstra's Algorithm
- Bellman Ford Algorithm

Dijkstra's Algorithm

23 April 2021 05:46 PM

Revisit lecture & do Tc

- Dijkstra's Algorithm:



For deleting each node from min heap — log V

Updating adjacent vertices by decrease key is called relaxation ← relaxation only once at each vertex.

O(n) creating min heap

For adjusting each decrease key

A	B	C	D	E	F	
0	∞	∞	∞	∞	∞	O(n)
A	—	—	—	—	—	
0	0+10	0+50	0+20	0+40	0+30	5 + 5 log V
A	A	A	A	A	A	
10		10+5	10+9	10+3	10+2	4 + 4 log V ← Relaxation at B
B		B	B	B	B	
12		15	19	13		0 + 0 log V ← Relaxation at F.
F		B	B	B		
13		15	13+4			2 + 1 log V ← Relaxation at E
E		B	E			
15			17			4 + 0 log V ← Relaxation at C
C			E			
17						
D						

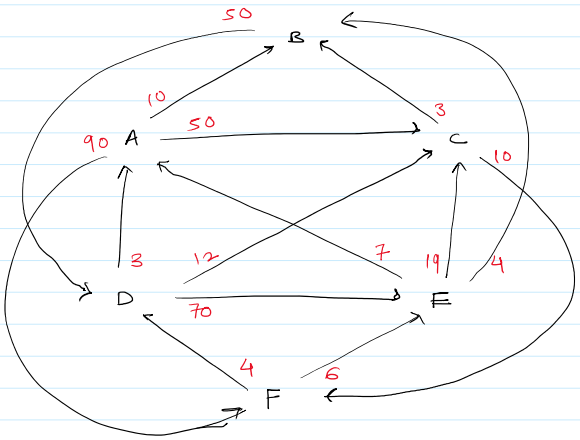
(5 = outgoing degree (making offer); 5 log V = 5 decrease key operation)

T_C = Build heap + Offering (Σ outgoing Degree) + Decrease key operations + Deletion from heap

= O(n) + E + E log V + V log V

= O[(V+E) log V]

Q.



A	B	C	D	E	F	
0	∞	∞	∞	∞	∞	
A	—	—	—	—	—	
0	10	50	∞	∞	90	
A	A	A	—	—	A	

10	50	60	∞	90
B	A	B	—	A
50		60	∞	60
C		B	—	F
60			130	60
D			D	D
60			66	
F			F	
66				
E				

66 ← cost from source

E ← order of selecting.

Q. output sequence of vertices identified by Dijkstra's

→ A – B – C – D – F – E

Q. Cost from A → E

→ 66

Q Path from A → E

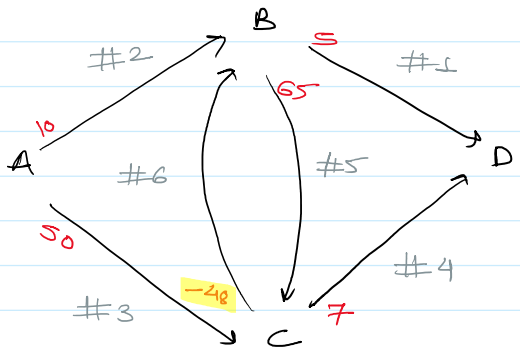
→ A → B → D → F → E

– If graph contains all positive Dijkstra's will always be right.
Other than that, it may fail.

Bellman Ford Algorithm

23 April 2021 05:47 PM

For negative weights



Give numbers # to each edge

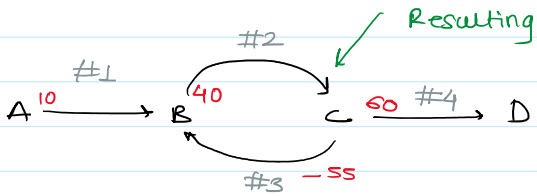
Update the costs by considering each edge

A	B	C	D	
0	∞	∞	∞	— E → Relaxation of all vertices at each pass
A	—	—	—	
0	10	50	∞	— E
A	A	A	—	
0	2	50	15	— E
A	C	A	B	
0	2	50	7	— E
A	C	A	B	

No change further.

$\therefore T_c = O(V.E)$

Q.



Resulting sum of cycle is negative. Goes in infinite loop

A	B	C	D
0	∞	∞	∞
A	—	—	—
0	10	∞	∞
A	A	—	—
0	10	50	∞
A	A	B	—
0	-5	35	110

0 A	−5 C	35 B	110 C
0 A	−20 C	20 B	95 C
⋮	⋮	⋮	⋮
0 A	−∞ C	−∞ B	−∞ C

- If graph contains positive edge weight.
  1. Dijkstra's ← Passed
  2. Bellman Ford. ← Passed

- If graph contains negative edge weights but no negative edge weight cycle
  1. Dijkstra's ← May Fail.
  2. Bellman Ford. ← Passed

- If graph contains negative edgeweight cycle
  1. Dijkstra's ← Always fail.
  2. Belman Ford ← Wrong answers for vertex associated with cycle (Undefined answer)

Dynamic Programming

03 May 2021 07:32 AM

Follows Principle of optimality.

Dynamic Programming	Greedy Technique
– Gives optimal soln	– Given optimal solution
– More time	– less Time
– Covers all possibilities	– Covers Few posibilities
– Alway correct	– Not always correct.

Time Complexity = No. of Distinct Function Calls x Tc of each function

Applications

- Fibonacci series
- Longest common sub-sequence
- Matrix chain multiplication
- Sum of subset problems
- 0/1 Knapsack
- All pairs shortest path

1. Fibonacci Series

03 May 2021 07:41 AM

fibonacci series :

Addition of prev two terms.

i.e. = fib(n) = fib(n-1) + fib(n-2)

```
fib(n) {
  if (n==0)
      return 0;
  else if (n==1)
      return 1;
  return fib(n-1) + fib(n-2)
```

Recurrence relation for value

$T(n) = T(n-1) + T(n-2)$ ... $n > 1$

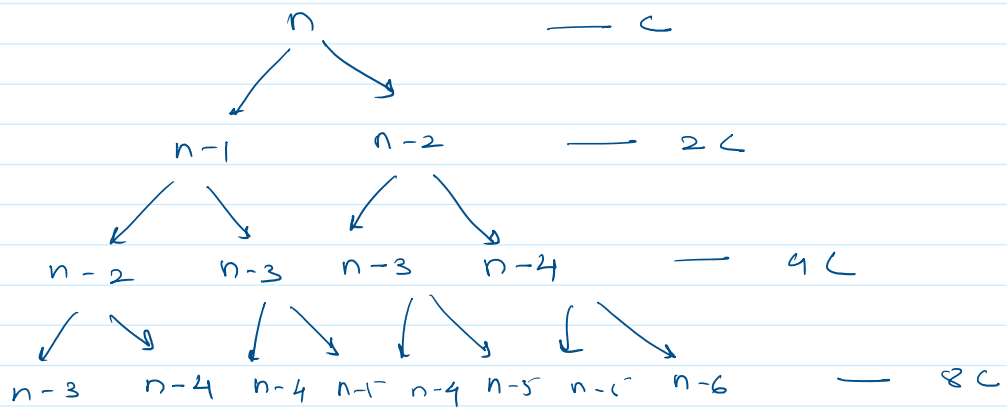
$= n$ ... $n = 1 \;||\; n = 0$

Recurrence relation for time

$T(n) = T(n-1) + T(n-2) + c$ ... $n > 1$

$= c$ ... $n = 1 \;||\; n = 0$

By tree method



$n-K$

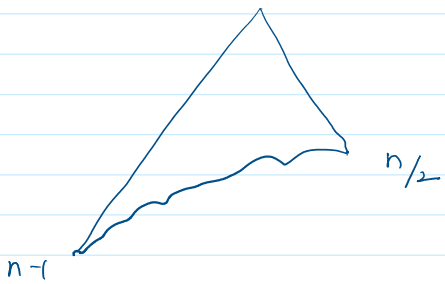
$n-k = 1$

$k = n-1$

$n-2K$

$n-2K = 0$

$k = n/2$



Series $= c[2^0 + 2^1 + 2^2 \cdots 2^k]$

$= c[2^0 + 2^1 + 2^2 \cdots 2^k]$

GP series.

$= c[1(2^k - 1)]$

GP series.

$$= c\left[\frac{1(2^k - 1)}{2-1}\right]$$

$k = n-1$

$$= c\left[2^{n-1} - 1\right]$$

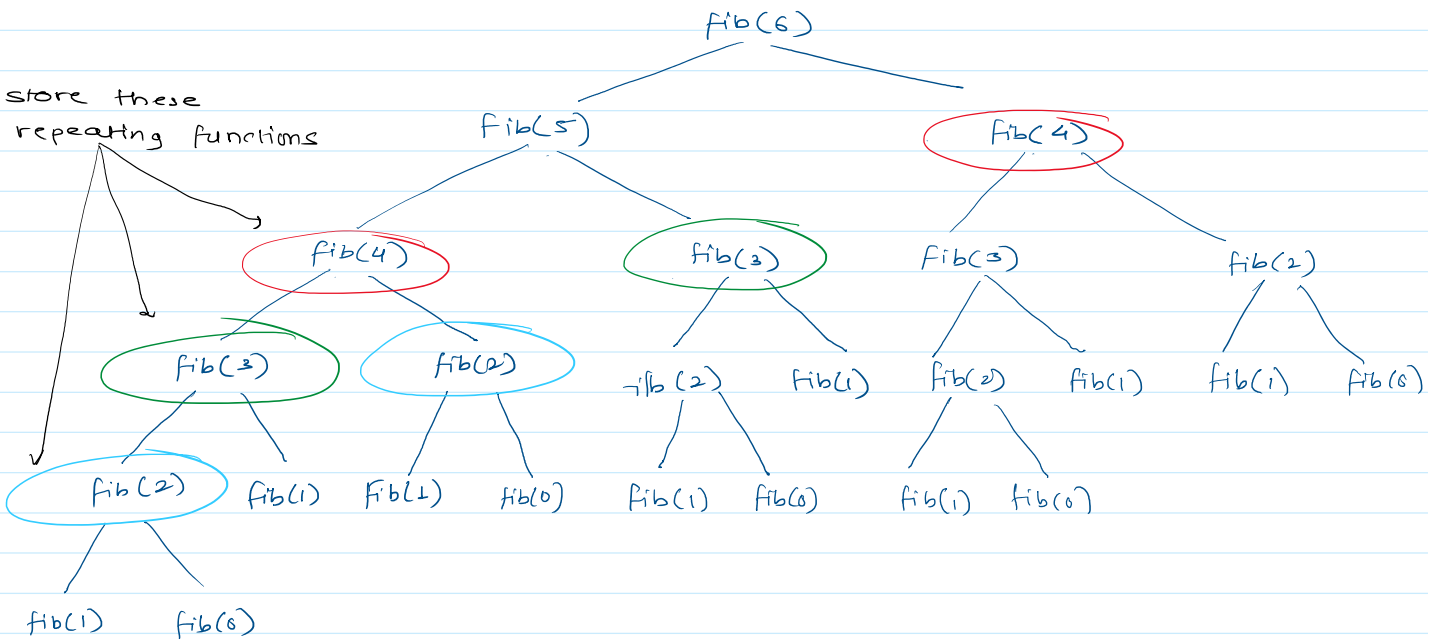
$$T(n) = O(2^n)$$

If $k = n/2$

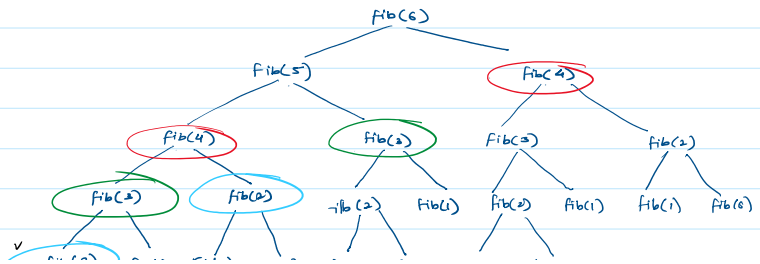
$$= c\left[(2^{n/2}) - 1\right]$$

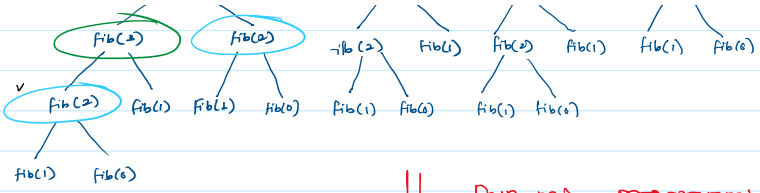
$$= c\left((\sqrt{2})^n\right)$$

let's consider fib(6)

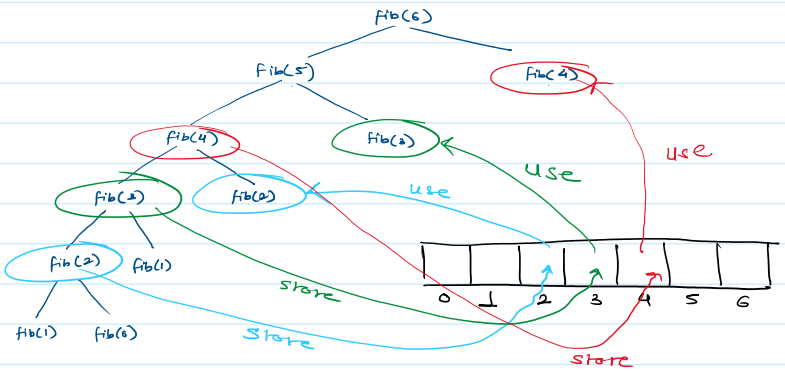


- Here for calculating fib(n) only (n-1) function calls required.
  But by brute force we are calculating 2^n function calls.
- Here overlapping function calls or repeating function calls present.
- Store the already calculated answers into data structure and reuse them.





Dynamic programming



- Psuedocode

```
DP_fib(n) {
    if (n==0 || n==1)
        return n;
    else {
        if (fib_table[n-2]==NULL)
            fib_table[n-2] = DP_fib(n-2)
        if (fib_table[n-1]==NULL)
            fib_table[n-1] = DP_fib(n-1)

        fib_table[n] = fib_table[n-2]
                     + fib_table[n-1]
    }
    return fib_table[n]
}
```

$T(n) = n+1$ ← function calls.

$= O(n)$

Space complexity = stack space + Table space

$= n + n+1$

$= \Theta(n)$

2.Longest common sub-sequence

04 May 2021 08:33 AM

Take any symbols continuous / non-continuous but order shouldn't be changed.

E.g.

$x = (B, A, B, A, B, A)$

$y = (A, B, A, B, A, B)$

$Z_0 = ()$

$Z_1 = (A), (B)$

$Z_2 = (BA), (AB), (AA), (BB)$

$Z_3 = (BAB), (ABA), (AAA), (BBB), (AAB), (BAA), (BBA), (ABB)$

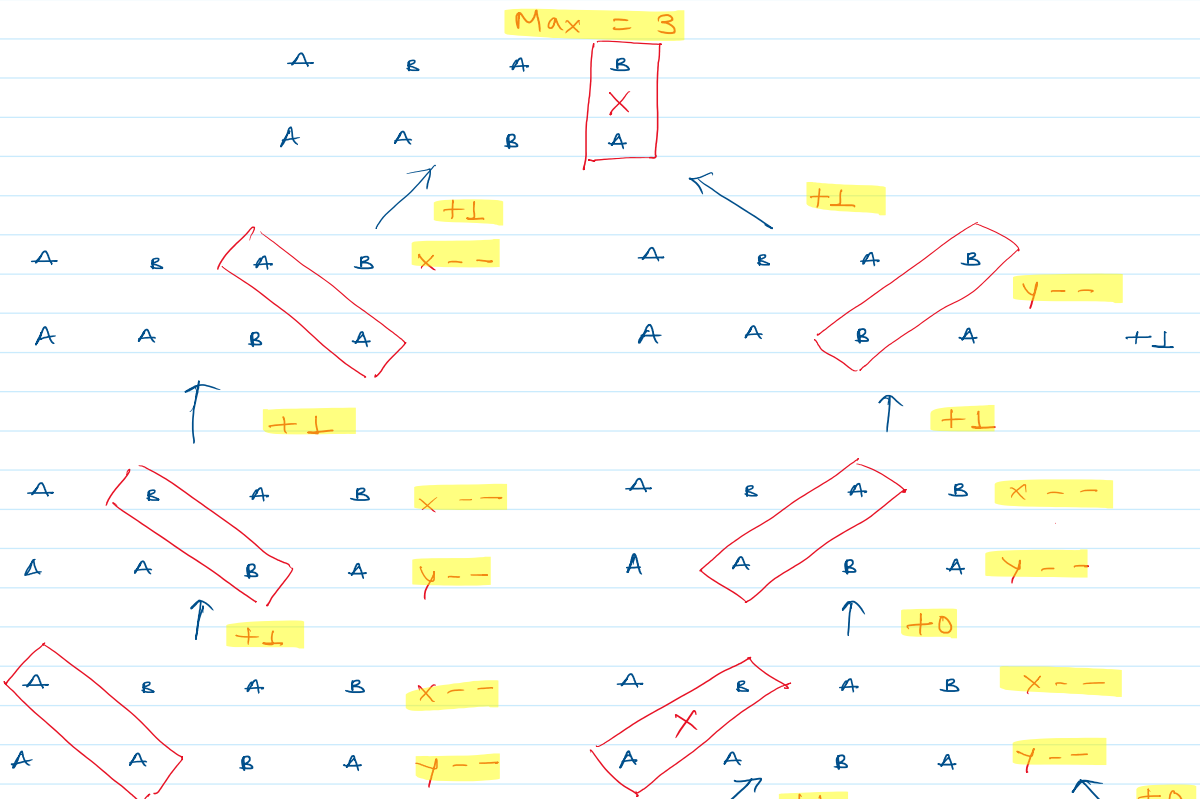
$Z_4 = (BABA), (ABAB), (BBBA), (AABB)$ — (BBBA, AABB: Not possible)

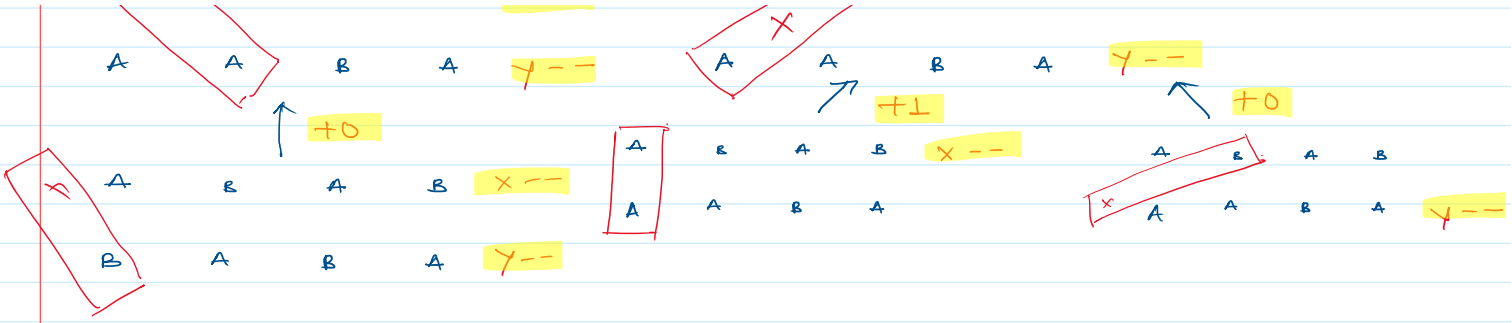
$Z_5 = (BABAB), (ABABA)$ ← longest common subsequence.

$Z_6 =$ No string common.

Algorithm:

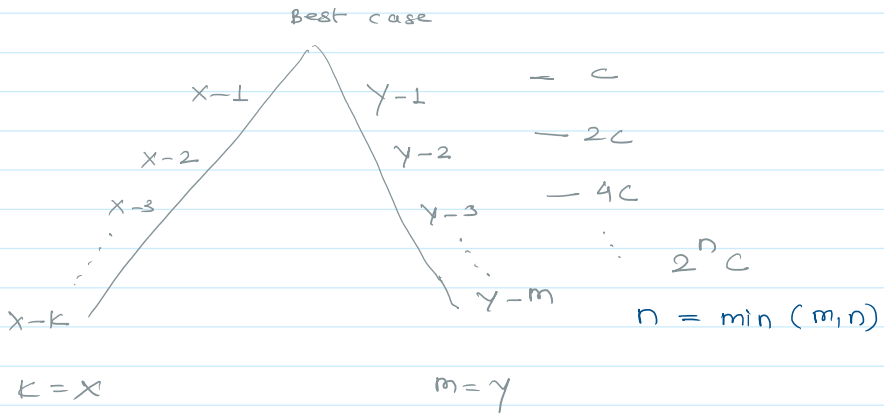
```
LCS (x, y)  ← x, y are indexes
    if (x = 0 || y = 0)
        return 0
    else {
        if a[x] == b[y]
            return 1 + LCS (x-1, y-1)
        else
            return max ( LCS (x-1, y),
                         LCS (x, y-1) )
```





$$T(x,y) = T(x-1,y) + T(x,y-1) + c$$

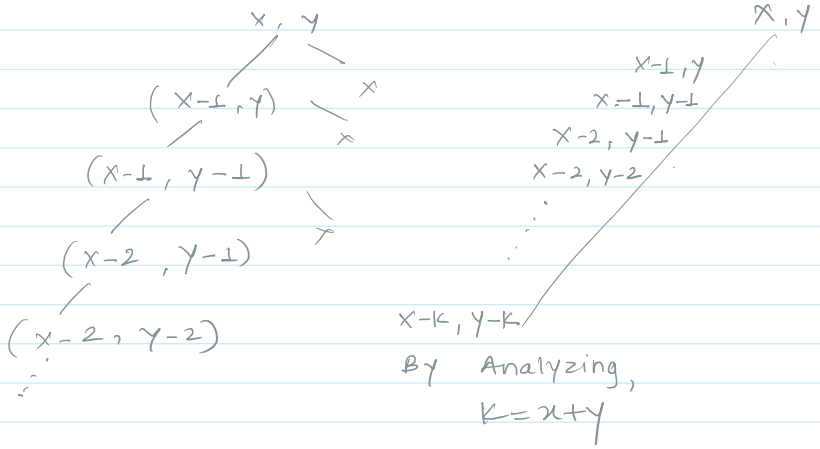
Best case



$n = \min(m,n)$

$k = x$ $\quad$ $m = y$

Worst case



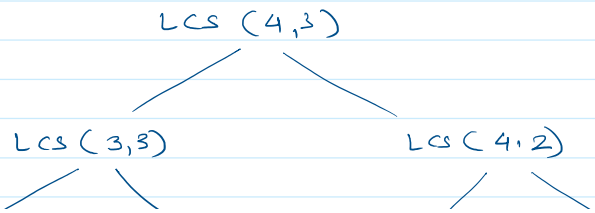
By Analyzing,

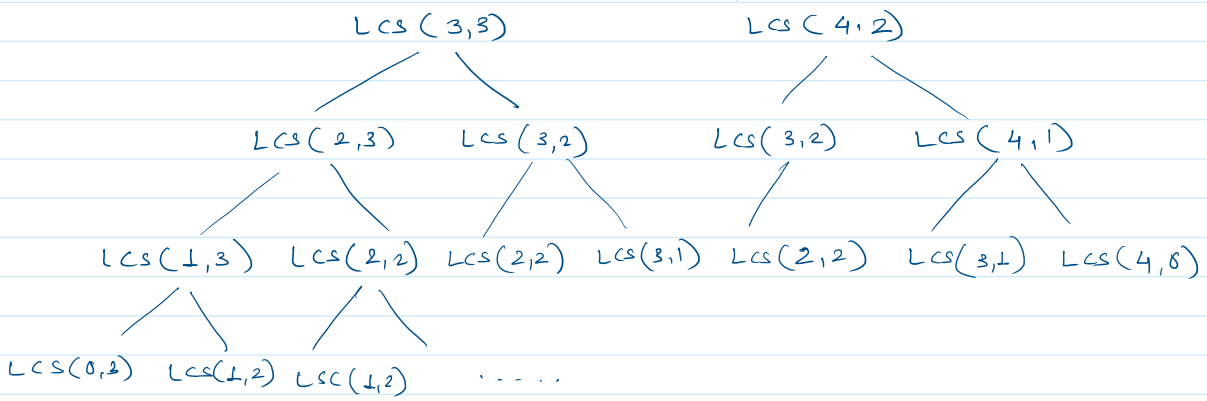
$k = x+y$

$$= 2^n c$$
$$= 2^{m+n}$$
$$= O(2^{m+n})$$
$$= \Omega(2^{\min(m,n)})$$

← Best case, when always matching, and ends when anyone string ends.

Dynamic programming





By Analyzing we can tell that there is lot of repetation We can store the values and cand reuse.

For finding distinct
- LCS(m,n)
  m,n has m×n combinations.

∴ m×n distinct function calls

∴ $T_c = O(mn)$

Space complexity.
- Stack space = m+n
- Array required = m×n

$S_c = O(m.n)$

- Non - Recursive LCS

```
LCS (x, y) {
    for( i = 1 ; i <= x , i++ )  {
        for ( j = 1, j <= y , j++) {
            if ( a[i] == b[j] ) {
                T[i, j]= 1 + T [i-1, j-1]
            else {
                T[i,j] = max( T[i-1, j],
                              T[i, j-1])

                }
            }
        }
    }
```

- Table feeling can be done in column major of row major.
- At a time only two rows / two columns required.

- Recursion
  - Calling : Preorder

- Recursion
  - Calling : Preorder
  - Evaluating : Postorder

3. 0/1 Knapsack

06 May 2021 08:14 AM

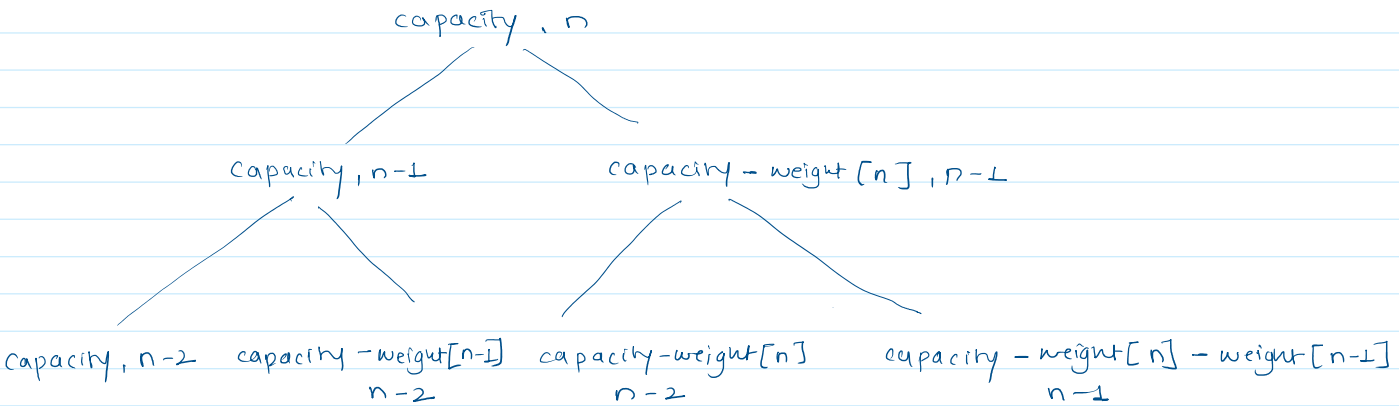
Ex.

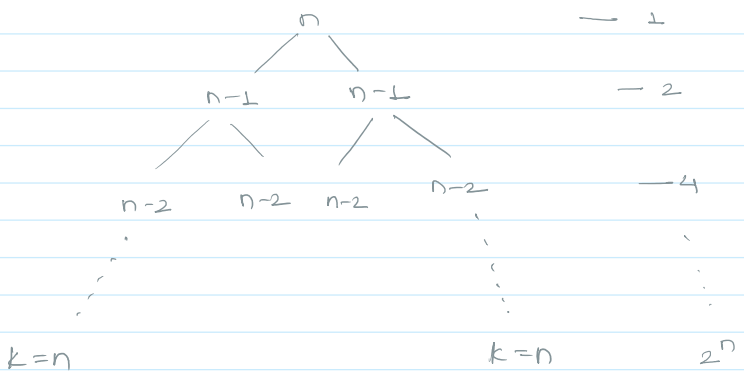
capacity = 10 n = 5

objects	ob_1	ob_2	ob_3	ob_4	ob_5
profits	25	70	50	10	60
weights	5	3	2	4	5

- Psuedocode.

```
Zo_knapsack ( capacity , n) {
    If ( capacity == 0 || n = 0 )
        return 0.
    else if ( weight [n] > capacity)
        return   Zo_knapsack( capacity, n-1)   ← Ignore if weight > capacity
    else
        return   max( profit[n] + Zo_knapsack ( capacity - weight[n] , n-1),
                      Zo_knapsack ( capacity , n-1)
                                  compare profit with considering element and
                                  without considering element.
}
```





$$T_c = 1 + 2 + 4 + 8 \cdots 2^n$$

$$= \left[1 \left(\frac{2^n - 1}{2 - 1} \right) \right]$$

$$= O(2^n)$$

Best case = $\Theta(n)$ ← Every time weight > capacity

Best case = $\Theta(n)$ ← Every time weight > capacity (Unary Tree)

Worst case = $\Theta(2^n)$

If we analyze recursive tree there are repetations at the leaf nodes.
So we can use dynamic programming.

∴ Number of distinct function calls.
= m.n
$T_n = O(mn)$

S_c = Stack space + Table
= n + m.n
= O(mn)

0/1 knapsack using dynamic programming
$T_c = O(m.n) \cong O(2^n)$
because of less repetations

∴ 0/1 knapsack is one of the NP complete.

4. Matrix Chain Multiplication

07 May 2021 14:19

Every year 1 Question. [optimal parenthization problem]

Finding best way to multiplying chain of matrices.

A_{ij} where i = row, j = column.

$A_{3.2} \times B_{2.4} = D_{3.4}$

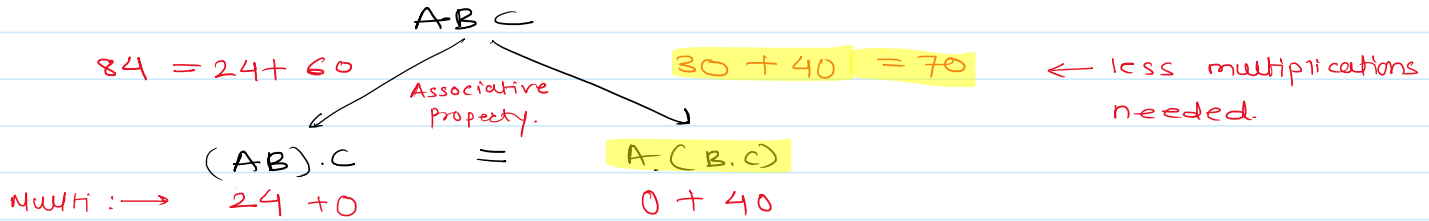
total elements in D $4 \times 3 = 12$

total multications needed = 12×3

$A_{3.2} \times B_{2.4} \times C_{4.5} = D_{3.5}$

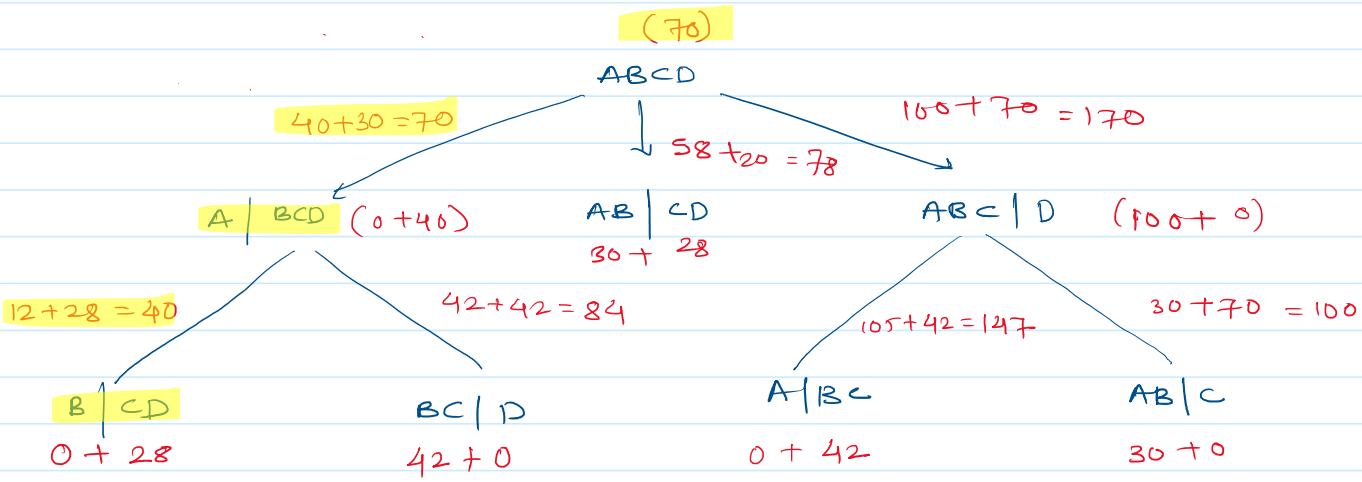
Total elements in D $3 \times 5 = 15$

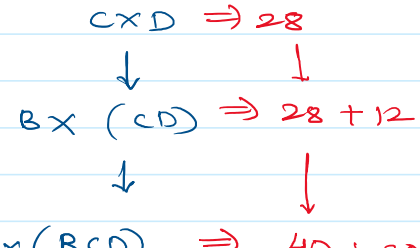
Multiplications needed total = $15 \times 2 \times 4 = 120$

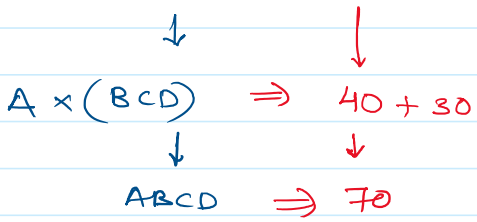


Ex.

$A_{5.3}\ B_{3.2}\ C_{2.7}\ D_{7.2} = ABCD$

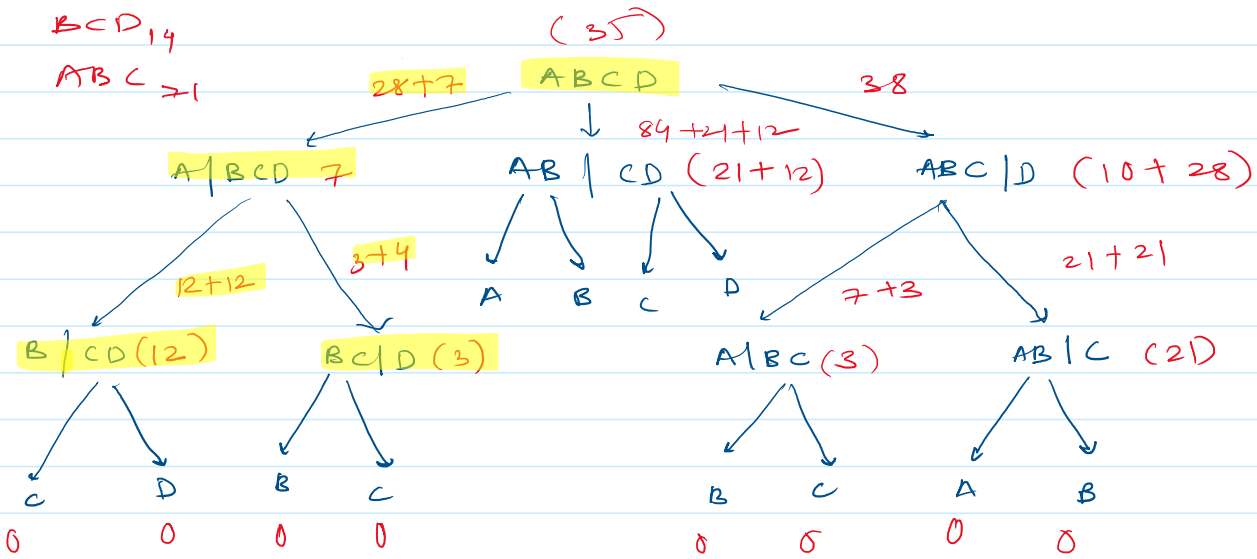






Ex. 2

A 7.1 B 1.3 C 3.1 D 1.4



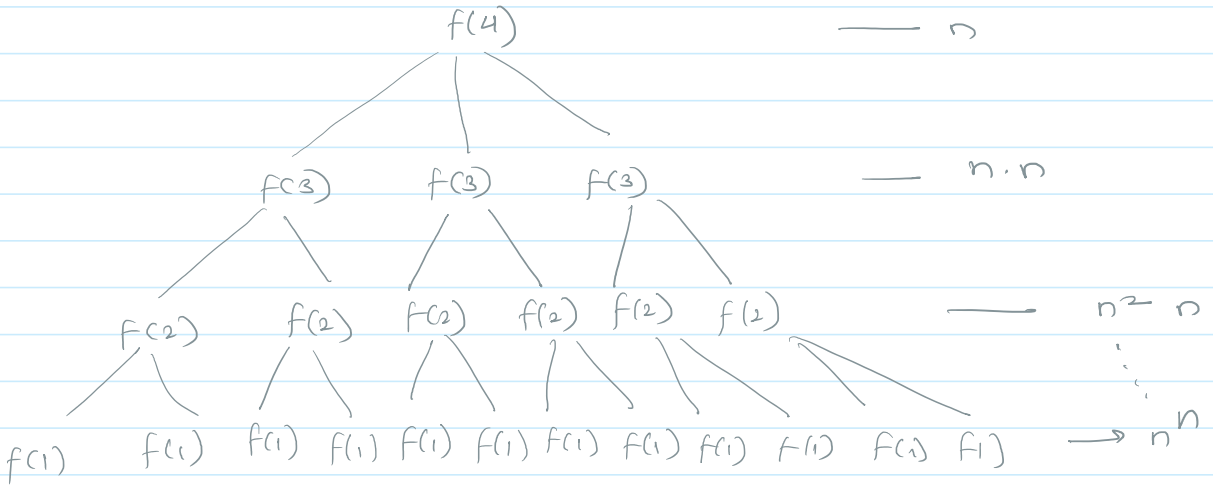
Total levels in tree = |Matrices|

level = n = 4

Tree = n-ary ← n-1 childs / parent

For finding min at each node n-1 comparisons needed

$$MCM(1,4) = \min \begin{cases} MCM(1,1) + MCM(2,4) + 28 \\ MCM(1,2) + MCM(3,4) + 84 \\ MCM(1,3) + MCM(4,4) + 28 \end{cases}$$



$$T_c = n + n^2 + n^3 + n^4 \ldots n^n$$
$$= n\left[1 + n^1 + n^2 + \ldots n^{n-1}\right]$$
$$= n\,n^n$$
$$= \Theta(n^{n+1})$$
$$= \Theta(n^n)$$

Algorithm

P[] x 2 ← Array for storing row & colum
P[i] = column of ith matrix
P[i-1] = row of ith matrix

```
MCM( i, j ) {

  If  i == i
        return 0;
  costs = []
for( k = i ; k < j ; k++) {
   costs += mcm( i, k) + mcm( k+1, j ) + p i-1 * p j * k
}

return   min( costs)
```

In above program many repetations are there.

∴ Distinct Function Calls

for n matrices

mcm (1 , n)

↓ ↓ upto

n 1

n × n

DFC $= n^2$

For finding minimum from n-ele.

$T_C = n^2 \times n = O(n^3)$

neglected

Space complexity $= n +$ Input space $+ n^2$

by Dynamic Prog.

$= O(n^2)$

5. Sum of Subsets

08 May 2021 08:48

Ex.

Input : (50 90 200 45 15 100 70)

O/p : Find any subset with sum m.

Exactly same as 0/1 knapsack problem

- Psuedocode

```
sos ( m, n) {
   if  m = 0
         return  ss
   if  n = 0
         return  -1
   if  a[n] > m
         return  sos(m, n-1)
   else
      return  sos ( m - a[n], n - 1)
      return  sos ( m, n-1)

}
```

Tc without dynamic programming

$T_c = O(2^n)$

With dynamic programming

$T_c = O(m.n)$