Lecture - 1

Properties of Algorithms

1. Input - 0 or more objects from a specified set.
2. Output - 1 or more objects. Must have a specified relation to the input.
3. Finiteness - Must terminate after a finite no. of steps.
4. Definiteness - Each step must be precisely defined.
5. Effectiveness - Each step must be feasible.

Lecture 2

Recursion

"To iterate is human, to recursive divine."

- Euclidean division

Input - Non-negative integer x, positive integer y.

Both expressed as binary numbers.

Output - The quotient q and the remainder r, obtained when x is divided by y.

E.g.:- $x = 105, y = 10$ | $q = 10, r = 5$
$x = 105, y = 9$ | $q = 11, r = 6$
$x = 60, y = 5$ | $q = 12, r = 0$

$x = (q \times y) + r$, where $0 \le r < y$

```
def euc_div(x, y):
    if x < y:
        return (0, x)
    else:
        return (1, 0) + euc_div(x-y, y)
```

Base Case

(1) $q = 0, r = x$

$$x = \underbrace{(0 \times y)}_{q} + \underbrace{x}_{r}$$

(2) Also, $0 \le x < y$

$\Rightarrow 0 \le r < y$

Induction over x values, for $x \ge y$

For induction, we assume $x \ge y$

Let $(q_1, r_1) = euc_div(x-y, y)$

Assume "inductively" that:

(1) $x - y = q_1 y + r_1$

(2) $0 \le r_1 < y$

Now, $x = q_1 y + y + r_1$

$x = (q_1 + 1) y + r_1$

We have already - $0 \le r_1 < y$

$\Rightarrow euc_div(x, y) = (1 + q_1, r_1) = (1, 0) + (q_1, r_1)$

$= (1, 0) + euc_div(x-y, y)$

Divide (x, y):

if $x = 0$: return $(0, 0)$ — Base Case

$z = \left\lfloor \frac{x}{2} \right\rfloor$

$(q, r) =$ Divide (z, y)

Set $q = 2q$; $r = 2r$

if x is odd: $r = r + 1$

if $r \geq y$:

$r = r - y$

$q = q + 1$

return (q, r)

Why should the above code work?

<u>Induction step</u>

Assume x is even, say $x = 2k$

Then $z = k$

Let $(q_1, r_1) =$ Divide (k, y)

Inductively, $z = q_1 y + r_1$, $0 \leq r_1 < y$

$\Rightarrow 2z = (2q_1)y + 2r_1$

$\Rightarrow x = 2z = q_2 y + r_2$

where $q_2 = 2q_1$

$r_2 = 2r_1$

But r_2 might be more than y.

Case 1: $r_2 < y$

$\rightarrow$ Trivial

Condition for $0 \le r_2 < y$ is satisfied

Case 2: $r_2 \ge y$

Set $r_3 = r_2 - y$

$q_3 = q_2 + 1$

$x = (q_2+1)y + r_2 - y$

$= q_3 y + r_3$, $0 \le \overset{r_2 - y}{\overset{\shortparallel}{r_3}} < r_2$

$x = (2q_1+1)y + (2r_1 - y)$

Why $r_3 < y$?

$r_1 < y \Rightarrow 2r_1 < 2y$

$\Rightarrow 2r_1 - y = r_3 < 2y - y = y$

Why $r_3 \ge 0$?

We have $r_2 \ge y$ (case 2)

$\Rightarrow 2r_1 \ge y$

$\Rightarrow 2r_1 - y \ge 0$

$\Rightarrow r_3 \ge 0$

Hence $0 \le r_3 < y$

Euclid's Algorithm for GCD

Input - Non-negative integers x, y of which at least one is positive.

Output - The gcd of x, y

```
Euclid (x, y):
    if y = 0: return x
    (q, r) = Divide (x, y)
    gcd = Euclid (y, r)
    return gcd
```

Lecture - 3

```
Divide (x, y):
    if x < y:
        return (0, x)
    else:
        x = x - y
        (q, r) = Divide (x, y)
        return (q+1, r)
```

Proof by induction

Induct on q (unknown) that satisfies

$$x = qy + r, \quad 0 \le r < y$$

Page No.:
Date:

Base Case: $q=0$

$x = r < y$

i.e., $x<y$

$\Rightarrow$ $(\cancel{q,x})$ Divide$(x,y) = (0,x)$

$= (q,r)$

Inductive step

Consider inputs (x,y) for which the correct q value for some $q \geq 1$

$x = qy + r,\ q \geq 1 \Rightarrow x \geq y$

$x_1 = x - y = qy + r - y$

$= (q-1)y + r$

By induction we have

Divide(x, y) return $(q-1, r)$

Our code returns

$((q-1)+1, r)$

$= (q, r)$

The GCD problem

<u>Input</u>: Non-negative integer x, y, at least one of which is positive.

<u>Output</u>: The Greatest common Divisor (GCD) of x and y.
- The largest integer z such that z divides both x and y "evenly" (i.e., without remainder).

$x = 43, y = 10$	1
$x = 45, y = 18$	9
$x = 66, y = 9$	3
$x = 60, y = 36$	12
$x = 95, y = 38$	19

The current fastest algorithm for factorizing an n-bit number takes $\approx 2^{n/4}$ steps

```
Euclid (x, y):
    if y = 0:
        return x
    (q, r) = Divide (x, y)
    gcd = Euclid (y, r)
    return gcd.
```

Page No.:
Date:

Proof of correctness of Euclid

By induction on the value of y.

Base Case: $y = 0$

$\Rightarrow$ Euclid $(x, y) = x$ correct by Line 1 & 2

Inductive step: Let (x, y) be the input arguments, and let $y \geq 1$

Line 3 computes q & r such that
$$x = qy + r, \quad 0 \leq r < y$$

The recursive call is Euclid(y, r)

We want:
$$\gcd(x, y) = \gcd(y, r)$$
$$\Leftrightarrow$$
$$\gcd(qy + r, y) = \gcd(y, r)$$

Suppose s divides y and r. Does s divide $qy + r$ and y?

Say $y = s \cdot y'$, $r = s \cdot r'$

$$qy + r = qs \cdot y' + s \cdot r'$$
$$= s(qy' + r')$$

$\Rightarrow$ s divides $qy + r$ & trivially divides y.

Suppose t divides $qy+r$ & y. Does t divide y and r?

~~Say $qy+r$~~

~~Say $y = y'' \cdot t$, $qy+r =$~~

$r = (qy+r) - qy$

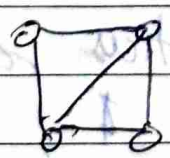
$\Rightarrow$ t divides r, and trivially y.

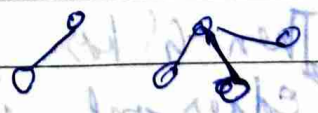
Hence they have the same set of divisors.

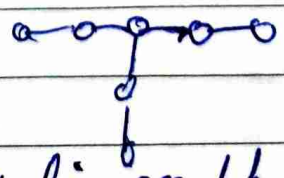
$\Rightarrow$ $\gcd(x,y) = \gcd(y,r)$

We have $r < y$, hence by induction we already ~~know~~ know $\gcd(y,r)$.

Graph $G=(V,E)$, vertices, edges, paths, connected, acyclic, tree







Tree - connected & acyclic graph.

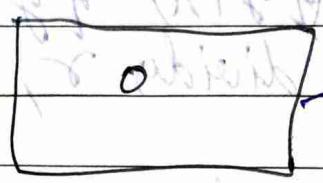
Date:

Claim: Any connected graph $G = (V, E)$ with $|V| = |E| + 1$, must be acyclic.

Proof: By induction on $|V|$

Base Case: $|V| = 1$

Then clearly, $|E| = 0$



No edges
Just 1 vertex

Induction Case: The claim holds for all graphs with $\leq k$ vertices for some $k \geq 1$.

Inductive step

Let $G = (V, E)$ be a connected graph with $|V| = k = |E| + 1$

By inductive hypothesis this has no cycles

Consider an arbitrary graph G' obtained from G by (i) adding a new vertex v, and (ii) an arbitrary edge from v to some vertex of G.

Then G' has $k+1$ vertices and k edges and is acyclic. ∎

Hence proved.

Lecture 4

	# steps
Divide (x,y):	
if $x=0$:	2 check
return $(0,0)$	return
$z = \lfloor \frac{x}{2} \rfloor$	2 remove last bit copying
$(q,r) =$ Divide (z,y)	$T(n-1)+1$
$q = 2\times q$	n
$r = 2\times r$	n
if x is odd:	1
$r = r+1$	n
if $r \geq y$:	n
$r = r-y$	n
$q = q+1$	n
return (q,r)	1

~~No. of steps~~

Let $T(n)$ be the time taken by the call Divide(x,y), when x has n bits.

$$T(n) = (7+6n) + T(n-1)$$

$$\Rightarrow T(n) \leq n(6n+7) = 6n^2+7n$$

$$\cancel{\Rightarrow T(n) \in O(n^2)}$$

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

Euclid(x,y):	# steps
if $y=0$: return x	2
$(q,r) =$ Divide (x,y)	$6n^2+7n$
gcd = Euclid (y,r)	$T(2n-1)$
return gcd	1

~~Assume x,y are n-bit numbers,~~

Let $T(2n)$ be the time taken by Euclid(x,y) when x, y together have 2n bits.

We know $r < y$

Claim: the no. of bits in r is at most one fewer than the number of bits in n.

$$\boxed{r < \frac{n}{2}}$$

$$T(n) = \boxed{6n^2+7n+3} + T(2n-1)$$

$$\Rightarrow T(2n) \leq 2n(6n^2+7n+3) = 12n^3+14n^2+6n$$

$$\Rightarrow T(n) = O(n^3)$$

Lecture 5

Assumptions about $T(n)$

1. $T(n)$ is a non-decreasing function of n.

2. $T(n) \leq c$ when $n \leq d$,
   where d is a constant of our choice.

   — d is a single fixed constant of our choice.

Sorting

Input: An array $A[0, 1, \ldots, (n-1)]$ of n integers.

Output: An array $B[0, 1, \ldots, (n-1)]$ which contains a permutation of the elements of A in non-decreasing order.

"Simplify & conquer"

~~Input: A = [8 2, 111, 8~~

Input: $A = [82, 111, 822, 115, 449, 397, 361, 111, 91, 593]$

Assume: Someone has done <u>most</u> of the sorting for you.

e.g.: $A = [82, 111, 115, 361, 397, 449, 822, 111, 91, 593]$

The "small" step towards completion" problem

Input: An Array $A[0, \ldots, (n-1)]$ of n integers & an integer $0 \le i \le (n-1)$ ~~set~~ such that the part $A[0, \ldots, (i-1)]$ is sorted.

Output: An array $B[0, \ldots, (n-1)]$ where:

- $B[0, 1, \ldots, (i)]$ is sorted version of $A[0, \ldots, i]$
- $B[(i+1), \ldots, (n-1)]$ is identical to $A[(i+1), \ldots, (n-1)]$.

Let ~~small~~ SMALLSTEP(A, i) be an Algorithm that solves <u>this</u> problem.
What is an algorithm that solves Sorting?

Page No.:
Date:
YOUVA

```
InsertionSort(A) :
    n = len(A)
    i = 1
    while i ≤ (n-1):
        SMALLSTEP (A, i)
        i = i + 1
    return A

SMALLSTEP (A, i):
    next = A[i]
    j = i-1
    while j ≥ 0 & A[j] > next
        A[j+1] = A[j]
        j = j-1
    A[j+1] = next
```

Lecture - 6

```
InsertionSort(A):   // A = A{0,1,..,(n-1)} is an array of n integers
i = 1
while i <= (n-1):
    next = A[i]
    & j = i-1
    while j >= 0 & A[j] > next:
        A[j+1] = A[j]
        j = j-1
    A[j+1] = next
    i = i+1
return A
```

A loop invariant for a while (or for, or ...) loop is any statement that is true immediately before each iteration of the loop.

A useful loop invariant is any statement P that satisfies:

1. Initialization: P holds immediately before the first iteration of the loop.
2. Maintenance: If P holds immediately before one iteration of the loop then P holds immediately before the very next iteration of the loop.
3. Termination: Immediately after control exists the loop, the fact that P is an invariant lets us help showing that the algorithm is correct.

```
1  i = 1
2  while i ≤ (n-1):
3      next = A[i]
4      j = i-1
5      while j ≥ 0 & A[j] > next:
6          A[j+1] = A[j]
7          j = j-1
8      A[j+1] = next
9      i = i+1
10 return A
```

From 2 to 9

<u>Invariance</u>: The current array prefix ~~A~~ $A[0, 1, \ldots, (i-1)]$ is a sorted version of the original array (input). Prefix $A[0, 1, \ldots, (i-1)]$

- Initialization is trivially true
  - single element is always sorted.

Let i_1 $(i_1 \leq n-1)$ be the value of i immediately before an iteration of the loop when P holds. That is, $A[0, \ldots, (i_1-1)]$ is a sorted version of the original $A[0, \ldots, (i_1-1)]$

We argue (informally) that lines 3-8 executes the "small step" and placed $A[i_1]$ at the correct location in the current prefix $A[0, \ldots, i_1]$.

Line 9 sets $i = i_1 + 1$

Just before the next iteration, the value of $i = i_1 + 1$
This proves the Maintenance step.

When control exits the loop, $i = n$

	# steps
1	1
2	n
3	$n-1$
4	$n-1$
5	$\sum_{i=1}^{n-1} t_i$
6	$\sum_{i=1}^{n-1} (t_i - 1)$
7	$\sum_{i=1}^{n-1} (t_i - 1)$
8	$n-1$
9	$n-1$
10	1

Let t_i be the number of times that line 5 is executed when the variable i has value i.

$$\cancel{3\sum_{i=1}^{n-1} t_i + 5n - 2}$$

$$3\sum_{i=1}^{n-1} t_i + 3n$$

The ~~worst~~ largest possible value for $t_i = i + 1$

In this case # steps $= 3\sum_{i=1}^{n-1} (i+1) + 3n$

$$= 3\sum_{i=1}^{n-1} i + 6n - 3$$

$$= 3 \times \frac{n(n-1)}{2} + 6n - 3 = 3\left[\frac{n^2 - n}{2} + 2n - 1\right]$$

$$= 3\left[\frac{n^2 - n + 4n - 2}{2}\right] = \frac{3(n^2 + 3n - 2)}{2}$$

The least value of $t_i = 1$

Substituting, we get

$\#$ ~~sub~~ steps $= 6n - 3$

Best Case no. of steps

We can make all the coefficients as 1

$n^2 + n + 1 \leftarrow$ worst case

$n + 1 \leftarrow$ best case

Now for very large values of n,

$$n^2 + n + 1 \approx n^2$$

$$n + 1 \approx n$$

$O(n^2) \leftarrow$ worst case

$O(n) \leftarrow$ best case

C-I: Fast computer, 10^{10} steps per second

C-II: Slow computer, 10^7 steps per second

Insertion Sort: Implemented in tight assembly code i $2n^2$ steps

Merge Sort: Exam solution: $50 n \log_2 n$ steps.

Sorting 1 crore numbers, in the worst case:
C-I: 5.5 hours
C-II: 20 minutes

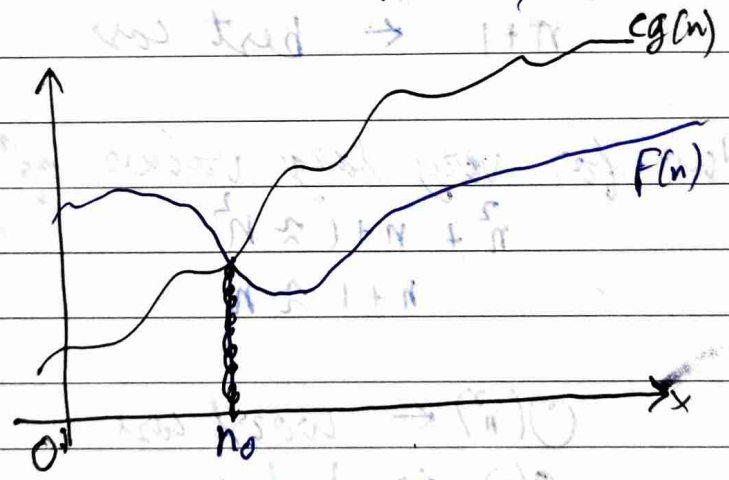
Asymptotic Notation

All functions are from $\{0, 1, 2, \ldots\}$ to real numbers

- since running times (functions) are of this type

- The O notation (read as "big Oh")

- $f = O(g)$ if $\exists$ +ve integers c, n_0 such that $\forall\, n \geq n_0$, $f(n) \leq c\,g(n)$



- Is $n = O(n)$?
  Yes. Eg: $c = 1, n_0 = 1$

- Is $n = O(5n)$?
  Yes. Eg: $c = 1, n_0 = 1$

- Is $5n = O(n)$?

  Yes. Eg: $c = 5$, $n_0 = 1$

- Is $2^{100} \cdot n = O\left(\frac{n}{3^{100}}\right)$?

  Yes. Eg: $c = 6^{100}$, $n_0 = 1$

- Is $n = O(n-100)$?

  Yes. but $c = 1$ won't work $\forall n_0$

  Eg: $c = 2$, $n_0 = 200$

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

then $f = O(h)$

- Is $n = O(n - \log_2 n)$?

  There is some n_0 such that $\forall n \geq n_0$,

  $\frac{n}{2} \geq \log_2 n$ holds

  $\Updownarrow$

  $-\frac{n}{2} \leq -\log_2 n \iff n - \frac{n}{2} \leq n - \log_2 n$

  $\iff n \leq 2(n - \log_2 n)$

Lecture - 7

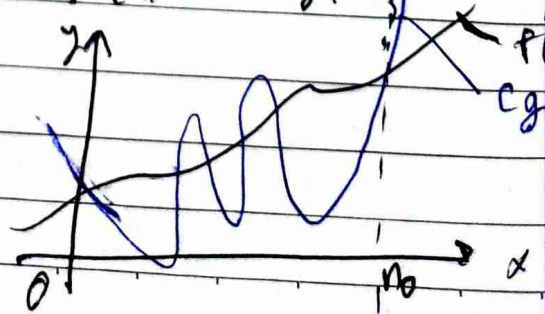
Insertion Sort (A) // A [0, 1, .. (n-1)] : array of n integers

```
1   i = 1
2   while i ≤ (n-1)
3       next = A [i]
4       j = i - 1
5       while j ≥ 0 and A[j] > next:
6           A[j+1] = A[j]
7           j = j - 1
8       A[j+1] = next
9       i = i + 1
10  return A
```

A useful loop invariant for the <u>while</u> loop in lines 2-9:

The current array prefix A[0, 1, .. (i-1)] is a sorted version of the ~~input~~ input array prefix ~~A[0, ..., i]~~ A[0, ..., (i-1)]

$f = O(g)$ if there exist positive constants c, n_0 such that $f(n) \le c\,g(n)$ holds for all $n \ge n_0$



- Is $5n = O(n+100)$?

Yes. $c = 5$, $n_0 = 1$

$5n \le 5(n+100) = 5n + 500 \quad \forall n \ge 1$

- Is $5n = O(n-100)$?

Yes. $c = 6$, $n_0 = 600$

$5n \le 6(n-100) = 6n - 600 \quad \forall n \ge 600$

- Is $n = O(n - \log_2 n)$?

Yes. ~~$c = 5$, $n_0 = 1$~~

$\exists\, n_0$ s.t.

$\frac{n}{2} \ge \log_2 n \quad \forall n \ge n_0$

$\Rightarrow 2\log_2 n \le n \Rightarrow n \le 2n - 2\log_2 n$

$\Rightarrow n \le 2(n - \log_2 n) \quad \forall n \ge n_0$

Set $c = 2$, n_0

- ~~Is~~ Is $n^2 = O(n)$?

No.

Suppose $\exists\, c, n_0$ such that

$n^2 \le cn \quad \forall n \ge n_0$

$\Rightarrow n \le c \quad \forall n \ge n_0$

But for $n > \max(c, n_0)$, we get $n > c$

A contradiction.

- Is $3n = O(2n)$?

Yes. Choose $c = \frac{3}{2}$, $n_0 = 1$

$3n \le \frac{3}{2} \cdot 2n = 3n \quad \forall n \ge 1$

- Is $2^{3n} = O(2^{2n})$?

No.

Suppose $\exists c, n_0$ s.t.

$2^{3n} \le c \cdot 2^{2n} \quad \forall n \ge n_0$

$\Rightarrow 2^n \le c \quad \forall n \ge n_0$

Choose $n > \max(n_0, \log_2 c)$

Then $n > \log_2 c$

$\Rightarrow 2^n > \log c$

This is a contradiction.

---

* The "big Omega" notation

$f = \Omega(g) \Leftrightarrow g = O(f)$

The "big Theta" notation

$f = \Theta(g) \Leftrightarrow f = O(g)$ and $f = \Omega(g)$

$$T(n) = T(n-1) + 10n + 5$$
$$= T(n-2) + 10(n-1) + 5 + 10n + 5$$
$$= T(n-3) + 10(n-2) + 5 + 10(n-1) + 5 + 10n + 5$$
$$= T(n-3) + 10[(n-2) + (n-1) + n] + 3 \cdot 5$$
$$= T(n-4) + 10[(n-3) + (n-2) + (n-1) + n] + 4 \cdot 5$$
$$\vdots$$
$$= T(n-i) + 10[(n-i+1) + (n-i+2) + \cdots + (n-1) + n] + i \cdot 5$$

Set $i = n-1$

$$= T(1) + 10[2 + 3 + \cdots + n] + (n-1)5$$
$$\leq C + 10(2 + 3 + \cdots + n) + 5(n-1)$$
$$= C + 10\left[\frac{n(n+1)}{2} - 1\right] + 5n - 5$$
$$= C + 5n(n+1) - 10 + 5n - 5$$
$$= C + 5n^2 + 10n - 15$$

We estimate that $T(n) = O(n^2)$

Step 2: Verify the estimate

Our claim: There exists positive constants c, n_0 s.t. $T(n) \leq cn^2$ holds for all $n \geq n_0$.

Proof by induction:

Base case:- $n = 1$

To show, $T(1) \leq c$

Proof: Assumption: $T(1) \leq d$

Choose $c \geq d$

Page No.:
Date:

Inductive hypothesis

$$T(n-1) \le c(n-1)^2$$

Induction step:

$$T(n) = T(n-1) + 10n + s$$
$$\le c(n-1)^2 + 10n + s$$
$$= cn^2 + (10-2c)n + c + s$$
$$= cn^2 - [(2c-10)n - c - s]$$

We get $T(n) \le cn^2 - [(2c-10)n - c - s]$

We want $T(n) \le cn^2$

Enough to have:

$$(2c-10)n - c - s \ge 0 \quad \forall n \ge n_0$$

We need $2(c-s) > 0$

Choose $c = \max(6, d)$

Then $n \ge \dfrac{c+s}{2c-10} = n_0$

Lecture - 8

$$T(n) = T(n-1) + 10n^2 + 5n$$
$$= T(n-2) + 10(n-1)^2 + 5(n-1) + 10n^2 + 5n$$
$$= T(n-2) + 10(n^2 + (n-1)^2) + 5(n + (n-1))$$
$$= T(n-3) + 10(n^2 + (n-1)^2 + (n-2)^2) + 5(n + (n-1) + (n-2))$$
$$\vdots$$
$$= T(n-i) + 10[n^2 + (n-1)^2 + \cdots + (n-i+1)^2]$$
$$+ 5[n + (n-1) + \cdots + (n-i+1)]$$

Setting $i = n-1$,

$$= T(1) + 10[n^2 + (n-1)^2 + \cdots + 3^2 + 2^2]$$
$$+ 5[n + (n-1) + \cdots + 3 + 2]$$

Assuming $T(1) \le d$ ~ some constant

$$\le d + 10\left[\frac{n(n+1)(2n+1)}{6} - 1\right] + 5\left[\frac{n(n+1)}{2} - 1\right]$$

$$= d + \frac{5n(n+1)(2n+1)}{3} - 10 + \frac{5n(n+1)}{2} - 5$$

$$= \frac{10n^3}{3} + \frac{15}{2}n^2 + \frac{25}{6}n + d - 15$$

Our estimate : $T(n) = O(n^3)$

Step 2: Verify the estimate.

To prove: There are positive constants c, n_0 such that $T(n) \le cn^3 \ \forall\, n \ge n_0$

Page No.:
Date:

By induction on n

Inductive hypothesis: $T(n-1) \le c(n-1)^3$ holds $\forall n \ge n_0$

$$T(n) = T(n-1) + 10n^2 + 5n$$
$$\le c(n-1)^3 + 10n^2 + 5n$$
$$= cn^3 - 3cn^2 + 3cn - c + 10n^2 + 5n$$
$$\cancel{= cn^3 + (10-3c)n^2}$$
$$= cn^3 - (3c-10)n^2 + (3c+5)n - c$$
$$= cn^3 - [(3c-10)n^2 - (3c+5)n + c]$$

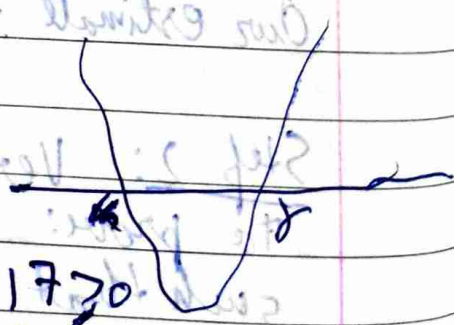
We want:

$(3c-10)n^2 - (3c+5)n + c \ge 0 \quad \forall n \ge n_0$ for some fixed choice of n_0

derivative: $2(3c-10)n - (3c+5)$

Say $c = 4$

$2n^2 - 17n + 4$



derivative $= 4n - 17 \ge 0$ if $n > \frac{17}{4}$

Choose $n_0 > \max\left(\frac{17}{4}, \gamma\right)$

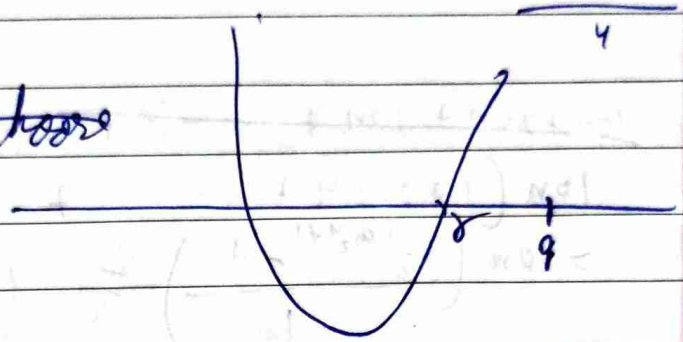
Then $2n^2 - 17n + 4 > 0 \quad \forall\, n \geq n_0$

$$r = \frac{17 + \sqrt{289 - 32}}{4} \cancel{= \sqrt{257}}$$

$$= \frac{17 + \sqrt{257}}{4} < \frac{17 + 16}{4}$$

$$< \frac{17 + 19}{4} = 9$$

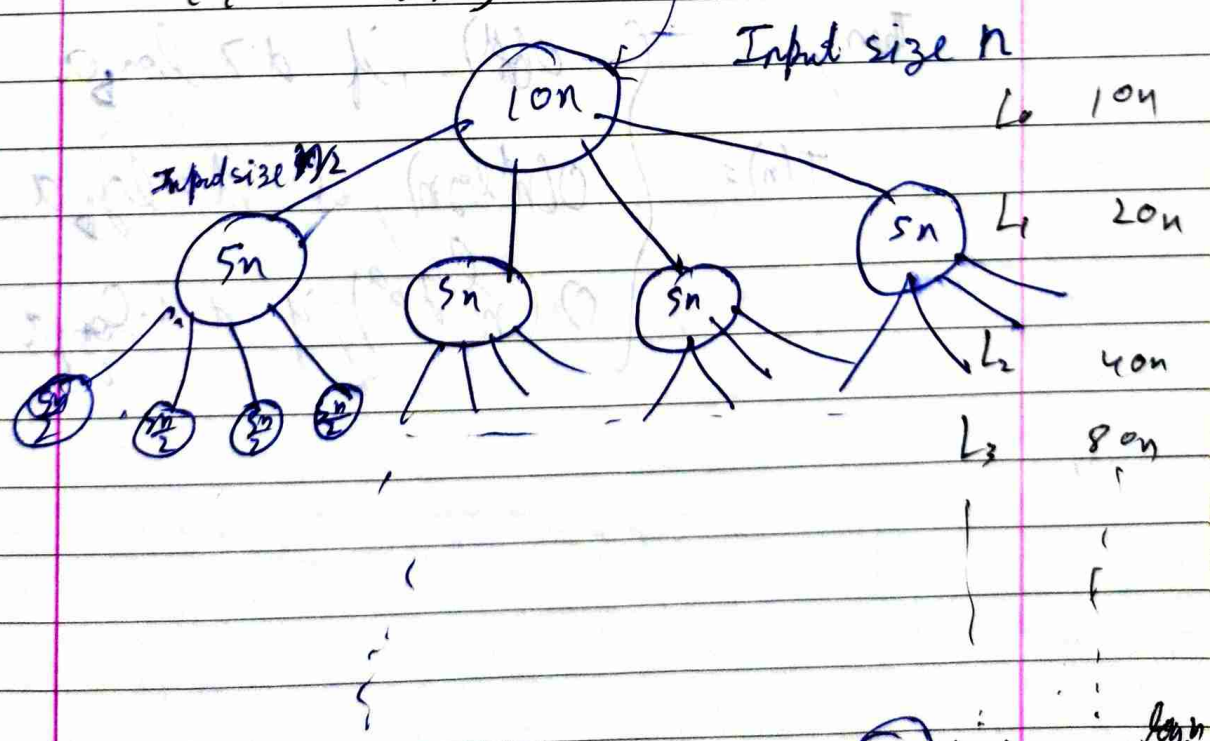
~~Hence choose~~



Choose $n_0 > \max\left(\frac{17}{4}, 9\right)$

Say $n_0 = 10$

$T(n) = 4 + (n/2) + 10n$



Solve (n) ← solve the base case

$10n$

x_1, x_2, x_3, x_4 : four inputs of size $\lfloor n/2 \rfloor$ each

$y_1 = \text{solve}(x_1)$, $y_2 = \text{solve}(x_2)$, $y_3 = \ldots$

answer = Combine (y_1, y_2, y_3, y_4)

return answer

$\cancel{10n + 20n + 40n + \cdots +}$

$10n\left(1 + 2 + 4 + \cdots + 2^{\log_2 n}\right)$

$= 10n\left(\dfrac{2^{\log_2 n + 1} - 1}{1}\right) = 10n(2n-1)$

$\approx O(n^2)$

Let $T(n) = aT(n/b) + O(n^d)$, for some constants $a \geq 0, b > 1, d \geq 0$

Then,

$$T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b a \\ O(n^d \log n), & \text{if } d = \log_b a \\ O(n^{\log_b a}), & \text{if } d < \log_b a \end{cases}$$

Lecture - 9

Divide & Conquer

Divide: Convert the input instance into one or more instances of the same problem.

Conquer: - If the input instance is small enough, solve it "by inspection".
- otherwise: Solve it using this self same algorithm.

Combine: Use the solution(s) of the smaller instance(s) to construct a solution for the original input.

Sorting:

Input: An array $A[0, 1, \ldots, (n-1)]$ of n integers

Output: An array $B[0, \ldots, (n-1)]$ which is the sorted version of A, in non-decreasing order.

~~A = [92, 44, 39, 36, 23, 9~~

Date:

A = [82, 44, 39, 36, 23, 91, 59]

Sort(A):

L = [82, 44, 39, 36] , R = [23, 91, 59]

Sort(L), Sort(R), combine

Sort(L):

LL = [82, 44] LR = [39, 36]

Sort(LL), Sort(LR), combine

Sort(LL):

LLL = [82] , LLR = [44]

Combine [82], [44] → [44, 82]

Sort(LR):

LRL = [39] , LRR = [36]

Combine [39], [36] → [36, 39]

Combine [44, 82], [36, 39] → [36, 39, 44, 82]

Sort(R):

RL = [23, 91] , RR = [59]

Merge Sort (A): — $T(n)$

		# steps
	$n = \text{length}(A)$	n
Conquer 1	if $n \le 1$: return A	$n+1$
Divide	$mid = \lceil \frac{n}{2} \rceil$	1
	$L = A[0, \ldots, (mid-1)]$	$n/2$
	$R = B[mid, \ldots, (n-1)]$	$n/2$
Conquer 2	$S_L =$ MergeSort(L)	$T(n/2)$
	$S_R =$ MergeSort(R)	$T(n/2)$
	$M =$ Merge (S_L, S_R)	n
	return M	

Merge(L, R)

$l = \text{length}(L)$, $r = \text{length}(R)$

$m = l + r$

M = an empty array of length m

$i = j = 0$

~~while for k=0 to (m-1):~~

for k=0 to m-1:

if $j \ge r$: $M[k] = L[i]$, $i = i+1$

else if $i \ge l$: $M[k] = R[j]$; $j = j+1$

else if $L[i] < R[j]$: $M[k] = L[i]$; $i = i+1$

else: $M[k] = R[j]$, $j = j+1$

return M

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

$$\le 2T(n/2) + 5n \quad ; \quad n \ge n_0$$

Page No.:
Date:

Master's theorem —

$$a\,T(n/b) + O(n^d)$$

$a = b = 2,\ d = 1$

$\log_b a = \log_2 2 = 1 = d$

$\Rightarrow T(n) = O(n^d \log n) = O(n \log n)$

Tutorial 1

Practice sheet - 1

11

Will use induction on $b-a$

```
IsFib (a, b):
    if (b-a) < 0 : False
    elif (b-a) == 0 ∧ b != 1 : False
    elif (b-a) == 0 ∧ b == 1 : True
    else:
        return IsFib((b-a), a)
```

Claim: IsFib(a, b) returns True if a & b are consecutive Fibo numbers

Base: $b - a = 1$

i) $a = 1,\ b = 2$: True

ii) $a = 2,\ b = 3$: ~~False~~ True

iii) $a \geq 3$:

$(a, a+1)$

$\hookrightarrow (1, a)$, $a \geq 3$

False

Hypothesis : For ~~some~~ $\forall i \le k \in \mathbb{N}$, the claim holds

Assume $\boxed{k > 1}$ ($k = 1$ is already covered)

Step a, b $\boxed{\text{s.t. } b = a + k + 1}$

i) a, b are consecutive Fib numbers
After ~~some iter~~ k iterations, we come on this
Is $Fib(a-k-1, k+1)$

$T \to (a, b)$

$T \to (k+1, a)$

$(a-k-1, k+1) = T$

12 Claim: DFS terminates by marking all vertices D.

Inducting on n - the number of vertices
$n = |V|$

Base case: $n = 1$

Inductive hypothesis: $|V| \le k$ claim works

Date:

Lecture - 10

Integer Multiplication

Input: Non-negative integers x, y represented as binary arrays $X[0, \ldots, (n-1)]$ & $Y[0, \ldots, (n-1)]$ where $X[n-1]$ is the least significant bit of x, and similarly for y.

Output: A binary array Z that contains a binary representation of the product $z = xy$, with the same endianness as X.

~~56 78~~

	56	78
×	12	34

$$(56 \times 10^2 + 78) \times (12 \times 10^2 + 34)$$
$$= 56 \times 12 \times 10^4 + (56 \times 34 + 78 \times 12) \times 10^2 + 78 \times 34$$

Divide and conquer

Observation: Let x, y be two n-bit numbers. Let x_L be the binary number consisting of the first $\lceil \frac{n}{2} \rceil$ bits of x and let x_R be the binary number corresponding to the remaining bits of x.

$$\lfloor \tfrac{n}{2} \rfloor + \lceil \tfrac{n}{2} \rceil = n$$

Let y_L, y_R be similarly defined. Then

$$xy = \left(2^{\lfloor \frac{n}{2} \rfloor} \cdot x_L + x_R\right) \cdot \left(\left(2^{\lfloor \frac{n}{2} \rfloor} \cdot y_L\right) + y_R\right)$$

$$= 2^{2\lfloor \frac{n}{2} \rfloor}(x_L \cdot y_L) + 2^{\lfloor \frac{n}{2} \rfloor}(x_L \cdot y_R + x_R \cdot y_L) + x_R \cdot y_R$$

```
Multiply (x, y):
    n = length(X)
    if n = 1:
        if Y[0] = 0: return X
        else: return Y
    mid = ⌈n/2⌉

    X_L = X[0, ..., (mid-1)]
    X_R = X[mid, ..., (n-1)]
    Y_L = Y[0, ..., (mid-1)]
    Y_R = Y[mid, ..., (n-1)]
    P_1 = Multiply(X_L, Y_L)
    P_2 = Multiply(X_L, Y_R)
    P_3 = Multiply(X_R, Y_L)
    P_4 = Multiply(X_R, Y_R)
    S = Add(P_2, P_3)
    F_1 = 2^⌊n/2⌋ · P_1
    F_2 = 2^⌊n/2⌋ · S
    F_3 = Add(F_1, F_2)
    P = Add(F_3, P_4)
    return P
```

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

Master's Theorem

$$\Rightarrow T(n) = n^{\log_2 4} = n^2$$

<u>Claim</u>: If we know the ~~numbers~~ products

- $x_L \cdot y_L$
- $x_R \cdot y_R$
- $(x_L + x_R)(y_L + y_R)$

Then we can compute xy with only additions, subtractions, and multiplication by powers of two.

$$T(n) = 3T(n/2) + O(n)$$

$$\Rightarrow T(n) = O(n^{\log_2 3}) \approx O(n^{1.6})$$

Lecture - 11

Counting Sort

For sorting numbers in a bounded range

Input: Array $A[0, \ldots, (n-1)]$ of n integers, each of which is in the range $\{0, 1, \ldots, k\}$ for some integer k.

Output: An array $B[0, \ldots, (n-1)]$ which is the sorted version of A, in non-decreasing order.

The idea – For each $A[i]$, count the number c_i of elements that are smaller-or-equal to $A[i]$

- Place $A[i]$ at position $B[c_i - 1]$

```
CountingSort (A, k):
    n = length(A)                                   - n
    B = An empty array of length n                  - n
    C = An empty array of length (k+1)              - k+1
    for i=0 to k: C[i]=0                            - k+1
    for i=0 to (n-1): C[A[i]] = (C[A[i]] + 1)       - n
    for i=1 to k : C[i] = C[i] + C[i-1]             - k
    for i = (n-1) down to 0:                        - n
        B[C[A[i]] - 1] = A[i]                       - n
        C[A[i]] = C[A[i]] - 1                       - n
    return B
```

$$\boxed{T(n) = O(n+k)}$$

Date:

- What is $C[A[n-1]]$?

This is the number of elements in A that are
- less than or ~~equal~~ equal to $A[n-1]$,
and
- have <u>not</u> yet been placed into B.

This means we should place $A[i]$ at the location
$$B[C[A[i]]-1]$$

So $n=100$, $A[99]=75$

$$C[A[99]] = C[75] = 25$$

<u>Radix Sort</u>

<u>Input</u>: Array $A[0, \ldots, (n-1)]$ of n integers, each of which has d digits. Digit 1 is the <u>least</u> significant digit.

<u>Output</u>: A sorted version of 'A'.

```
RadixSort(A, d)
    for i = 1 to d
        sort A on the digit i
            using Counting Sort (or some stable sort)
        Let B be the result
        A = B
    return A
```

Say each digit can take k different values.
Each call to Counting Sort takes $\Theta(n+k)$ time.

$$\Rightarrow T(n) = \Theta(d(n+k))$$

For Aadhaar, $d = 12$, $k = 10$

$$\Theta(12(n+10)) = \Theta(n)$$

Lecture - 12

Dynamic Programming

Fibonacci numbers - 1, 1, 2, 3, 5, 8, 13, ...

$F_0 = F_1 = 1$, $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$

Input: Non-negative integer n

Output: The Fibonacci number n

```
SimpleFib(n)
    if n ≤ 1:
        return 1
    F1 = SimpleFib(n-1)
    F2 = SimpleFib(n-2)
    return F1 + F2
```

$$T(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ T(n-1) + T(n-2), & \text{if } n \geq 2 \end{cases}$$

$T(n) = \Theta(1.618^n)$

```
SmartFib(n)
    if n ≤ 1:
        return 1
    P = a global array of length (n+1).
    F[0] = F[1] = 1
    for i = 2 to n:
        F[i] = 0
    Fn = FibHelper(n)
    return Fn
```

2

2

$\}\, n$

```
FibHelper(i)
    if F[i] ≠ 0 : return F[i]                2
    F1 = FibHelper(i-1)                      T(n-1)
    F2 = FibHelper(i-2)                      1
    F[i] = F1 + F2                           1
    return F[i]                              1
```

(constant time)

Smart Fib(5)

FibHelper(5)

- FibHelper(4) return 5
  - FibHelper(3) return 3
    - FibHelper(2) return 2
      - FibHelper(1) return 1
      - FibHelper(0) return 1
    - FH(1) return 1
  - FH(2) return 2
- FibHelper(3) return 3

Ex 8

```
SimpleFibDP(n)
    if n ≤ 1
        return 1
    F = an array of length (n+1)
    F[0] = F[1] = 1
    for i = 2 to n
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

Computing the binomial coefficient - ^nC_k

Input: Positive integer n, non-negative integer k.

Output: $^nC_k = \frac{n!}{k!\,(n-k)!}$

```
Choose(n, k)
    if k = 0 or k = n
        return 1
    C1 = Choose(n-1, k-1)
    C2 = Choose(n-1, k)
    return C1 + C2
```

```
SmartChoose(n, k)
    if k = 0 or k = n
        return 1
    C = a global (n+1) x (k+1) array
    Set all values in C to nil.
    for i = 0 to n:
        C[i][0] = 1
        if i <= k: C[i][i] = 1

    nCk = ChooseHelper(n, k)
    return nCk
```

```
ChooseHelper(n, k)
    if  C[n][k] ≠ nil
            return C[n][k]
    C[n][k] = ChooseHelper(n-1, k-1)
                + ChooseHelper(n-1, k)
    return C[n][k]
```

Lecture-13

Rod Cutting

Length (i)	1	2	3	4	5	6	7	8	9	10
Price (P_i)	1	5	8	9	10	17	17	20	24	30

For $0 \le i \le 10$, integers, let r_i be the (best) max revenue that can be obtained by (possibly) cutting up & selling a rod of length i.

$r_0 = 0$

i	r_i	length of rod
1	1	1
2	5	2
3	8	3
4	10	2+2
5	13	2+3
6	17	6
7	18	(6+1) or 3+2+2 or 5+2
8	22	6+2
9	25	6+3
10	30	10

$P[0, \ldots, n]$ is the list of price

$P[0] = 0$ & $P[i] = P_i$, $i = 1, \ldots, n$

```
def CutRod (n, P):
    if n == 0:
        return 0
    maxrevenue = -1
    for i = 1 to n:
        ifirstRevenue = P[i] + cutRod(n-i, P)
        if ifirstRevenue > maxrevenue:
            maxrevenue = ifirstRevenue
    return maxrevenue
```

Let $T(n)$ be the time of the func. runs.

$$T(n) = 1 + \sum_{i=1}^{n} T(n-i)$$

P_i is the selling price of a piece of length i. r_i denotes the max. revenue that can be realised from a piece of length i.

We want to find r_n.

<u>Step I</u>: Choose an attribute of the sol^n's & express the entire sol^n as a union of subsets, where each subset has a different value for this attribute

<u>e.g.</u>: Say, we choose the "length of any one piece in a sol^n" as the attribute of that sol^n for $1 \le i \le n$.
Let S_i be the set of all sol^n that have attribute i.

$S = S_1 \cup S_2 \cup \ldots \cup S_n \quad , \quad |S_i| = 1$

Let maxRevenue (S_i) be the max. revenue that can be realised from one element of set S_i.

Then r_n = max maxRevenue (S_i)

For a given i, maxRevenue $(S_i) = P_i + r_{n-i}$

$P[0, 1, \ldots, n]$ are the selling price.

$P[0] = 0$ & $P[i] = P_i$

```
def CutRodDP (n, P):
    R = an array of length n+1
        // R[i] will eventually
           contain r_i
    R[0] = 0
    for j = 1 to n:
        jmaxRevenue = -1
        for i // go over all pos
```

```
R[0] = 0
for j = 1 to n:
    jmaxrevenue = -1
        // go over all possible
           length of first
           piece
    for i = 1 to j
        irevenue = P[i] + R[j-i]
        if irevenue > jmaxrevenue:
```

smax revenue = i revenue
R[i] = smax revenue
return R[n]

Lecture - 14

```
SmartFib (n)
    if n ≤ 1: return 1                          1
    F = a global array of length (n+1)          n+1
    F[0] = F[1] = 1                             2
    for i = 2 to n                              }
        F[i] = 0                                } n
    F[n] = FibHelper (n)                        L(n)
    return Fn                                   1
```

$L(i)$

```
FibHelper (i)
    if F[i] ≠ 0: return F[i]                    1
    F1 = FibHelper (i-1)
    F2 = FibHelper (i-2)
    F[i] = F1 + F2                              1
    return F[i]                                 1
```

Except recursive calls, all steps take constant time

t: total number of calls to FibHelper (), when started off with a call to FibHelper(n)

Then, FibHelper(n) runs in $O(t)$ time

$$t \le \underbrace{2(n-1)}_{} + \underbrace{1}_{\text{Main call in SmartFib}}$$

Because each time FibHelper is being called an element is being added to F[ ] and none of its values is being changed back to zero.

Initially, F has (n-1) zero elements.

For 2 recursive calls, we make 1 ~~process~~ ~~to~~ progress to F.

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

```
SmartChoose(n, k)
    if k = 0 or k = n: return 1
    C = a global (n+1) × k array
    Set all values in C to NIL
    for i = 0 to n:
        C[i][0] = 1
        if i ≤ k: C[i][i] = 1
    nCk = ChooseHelper(n, k)
    return nCk
```

$O(nk)$

```
ChooseHelper(n, k)
    if C[n][k] ≠ NIL:
        return C[n][k]
    C[n][k] = ChooseHelper(n-1, k-1) +
                ChooseHelper(n-1, k)
    return C[n][k]
```

~~Once again~~

Except recursive calls, each part takes constant time (in ChooseHelper).

Every recursive call makes a NIL to non-NIL, Hence each recursive call makes some progress in C.

$\Rightarrow t \leq 2k(n+1) + 1$

$\Rightarrow T(n) = O(nk)$

Page No.:
Date:
YOUVA

```
CutRod (n, P)
    if n = 0 : return 0
    maxRevenue = -1
    for i = 1 to n :
        iFirstRevenue = P[i] + CutRod (n-i, P)
        if iFirstRevenue > maxRevenue
            maxRevenue = iFirstRevenue
    return maxRevenue
```

Now memoize this

```
MemCutRod (n, P)
    if n = 0 : return 0
    R = an array of length (n+1)
    R[0] = 0
    for i = 1 to n : R[i] = NIL
    maxRevenue = CutRodHelper (n, P, R)
    return maxRevenue

CutRodHelper (i, P, R)
    if R[i] ≠ NIL : Return R[i]
    imaxRevenue = -1
    for j = 1 to i
        jFirstRevenue = P[j]
                        + CutRodHelper (i-j, P, R)
        if jFirstRevenue > imaxRevenue :
            imaxRevenue = jFirstRevenue
    R[i] = imaxRevenue
    return R[i]
```

Lecture - 15

Matrix Chain Multiplication (Least cost version)

Input: A sequence $\langle A_1, A_2, \ldots, A_n \rangle$ of matrices, where the product $A_i A_{i+1}$ is defined for all $1 \le i < n$

Goal: Using the high-school matrix multiplication algorithm as a subroutine, compute $A_1 A_2 \ldots A_n$ using the least number of arithmetic operations (asymptotically).

Eg:- A_1, A_2, A_3, A_4

1. $A_1(A_2(A_3A_4))$: $(5 \times 50 \times 10) + (100 \times 5 \times 10) + (10 \times 100 \times 10) = 17500$
2. $A_1((A_2A_3)A_4)$: $(100 \times 5 \times 50) + (100 \times 50 \times 10) + (10 \times 100 \times 10) = 85000$
3. $(A_1A_2)(A_3A_4)$: — ~~85000~~ 8000
4. $(A_1(A_2A_3))A_4$: — 80000
5. $((A_1A_2)A_3)A_4$: — 12,500

$A_1 : 10 \times 100$, $A_2 : 100 \times 5$, $A_3 : 5 \times 50$, $A_4 : 50 \times 10$

The parenthesization that corresponds to any optimum solution can be thought of as consisting of:

1. A (break-point) index, $1 \le i < n$,
2. An optimum cost parenthesization of $\langle A_1, A_2, \ldots, A_i \rangle$
3. An optimum cost parenthesization of $\langle A_{i+1}, A_{i+2}, \ldots, A_n \rangle$

For this, if A_1 is not enclosed in parenthesis, then $i = 1$
Otherwise, i is the index such that the outermost right parenthesis that encloses A_1, occurs right after A_i.

<u>Input</u>: Array $D[0, \ldots, n]$, where the dimension of array A_i is $D[i-1] \times D[i]$

P: Universe of all parenthesizations
For $1 \le b < n$, let P_b be the set of all parenthesizations with "<u>break-point</u>" at b

then $P = P_1 \cup P_2 \cup \ldots \cup P_{(n-1)}$

Let $\widehat{OPT}(P_b)$ be the least cost among all the solution in set P_b

$$\boxed{OPT(1,n)} = \min_{b=1}^{n-1} \widehat{OPT}(S_b)$$

$$\widehat{OPT}(S_b) = \boxed{OPT(1,b)} \cancel{+OPT(b,n)} + \boxed{OPT(b+1,n)} + D[0]\cdot D[b]\cdot D[n]$$

No of matrices to optimize for has reduced

OPT [i][j] for $i \le j$ should be the ~~least~~ best cost of multiplying $\langle A_i, \ldots, A_j \rangle$

LeastCostMCM(n, D)

OPT = A 2D array of dimension $(n+1)\times(n+1)$

// OPT[0] is a dummy row, to make indices match the explanation.

~~OP~~

for i = 1 to n : OPT[i][i] = 0

for l = 2 to n : // l = j - i + 1

for i = 1 to n - l + 1 :

j = l + i - 1

OPT[i][j] = ∞

for b = i to (j-1):

val = OPT[i][b] + OPT[b+1][j] + D[i-1] D[b] D[j]

```
        if val < OPT[i][j]:
            OPT[i][j] = val
return OPT[i][n]
```

Pset-3

Q4

Python ✓ $r=2$

Pychorm

T(r) — LCS'(x, y, i, j, r):

c — if r==0: return True

c if (i ≥ m & j < n) or

c (i < m & j ≥ n): return False

c if x[i] != y[j]: return False

c else:

T(r-1) LCS'(x, y, i+1, j+1, r-1)

Claim: $T(r) = O(r)$

Base: $r=0, r=1$ ✓

Inductive Case:

Assume $T(k-1) = O(k-1)$ for some $k<r$

$\cancel{T(k) \le T(k-1)}$

$T(k) \le \max(c, T(k-1))$

$\Rightarrow T(k) \le T(k-1) \cancel{= O(k-1)}$

$+ c'$ constant being added for non-recursive steps

$$\Rightarrow T(k) \le c''(k-1) + c'$$
$$\Rightarrow T(k) = O(k)$$

Q.6 (d) Prove or disprove
$$n^{\log_2(\log_2 n)} = O\left(2^{\sqrt{\log_2 n}}\right)$$

Assume $\exists\ c, n_0$ s.t.

$$n^{\log_2(\log_2 n)} \le c \cdot 2^{\sqrt{\log_2 n}} \quad \forall n \ge n_0$$

$$\Rightarrow \log_2(\log_2 n) \cdot \log_2 n \le \log c + \sqrt{\log_2 n}$$
$$\le c_1 \sqrt{\log_2 n}$$

for some constant c_1

Square bothsides -

$$\left(\log_2(\log_2 n)\right)^2 \log_2^2 n \le c_1^2 \log_2 n$$

$$\Rightarrow \left(\log_2(\log_2 n)\right)^2 \cdot \log_2 n \le c_1^2$$

A contradiction

Q6. (e) $2^{\sqrt{\log_2 n}} = O\left(n^{\log_2(\log_2 n)}\right)$

$n_0 = 16, \; c = 2$

$\forall n > n_0$,

$\log_2 n > \log_2 16 = 4$

$\Rightarrow \log_2(\log_2 n) > \log_2 4 = 2$

$\Rightarrow n^{\log_2(\log_2 n)} > n^2$

Also, $2^{\sqrt{\log_2 n}} < 2^{\log_2 n} \quad \forall n > 16$

$\lfloor \log_2 n \rfloor = k$

$\Rightarrow \log_2 n \geq k$

$\Rightarrow 2^k \leq n$

$\Rightarrow (2^k)^2 \leq n^2$

$\Rightarrow 2^{\log_2 n} \leq 2^k < (2^k)^2 \leq n^2$

$\Rightarrow 2^{\sqrt{\log_2 n}} < n^{\log_2(\log_2 n)}$

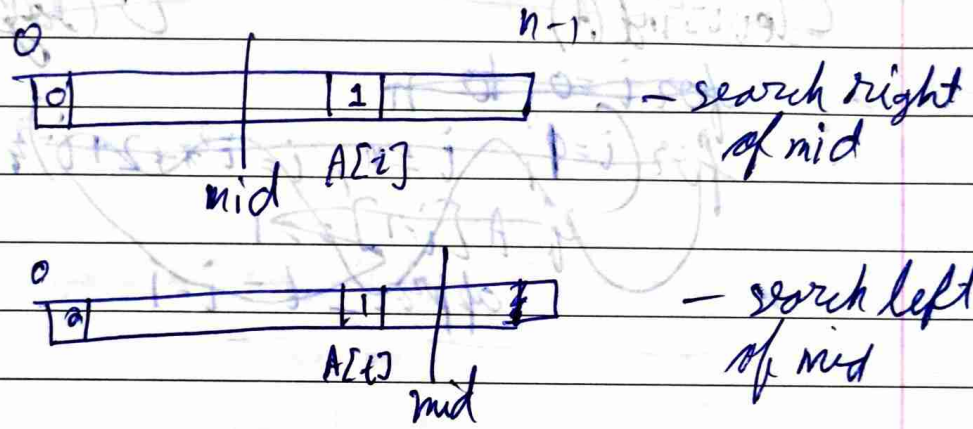
18. $A = [0, \ldots, \underset{t}{1}, \ldots, 1]$

(a) SimpleFindT(A) $\quad O(n)$

```
n = length(A)
for i = 0 to n
    if A[i] == 1
        return i
```

} Linear search

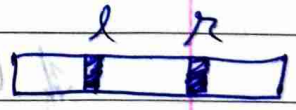
(b) ~~SmartFindT(A)~~ Binary search



SmartFindT(A) $\quad O(\log n)$

```
n = length(A)
BinSearch(A, 0, n-1)
```

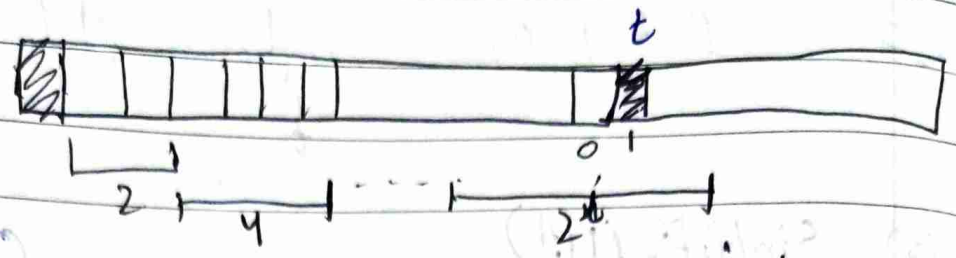
$l \quad r$




```
BinSearch(A, l, r)
    if l == r
        return l
    mid = (l + r)/2
    if A[mid] = 0
        return BinSearch(A, mid+1, r)
    else: return BinSearch(A, l, mid)
```

not mid-1 since for A[mid]=1, that can be the t value that we want

(c)



$2^i + 1, \ldots, 2^{i+1} - 1$

$1 + 2 + \cdots + i \approx O(i^2) = O(\log t)$

$2^i \le t \le 2^{i+1}$

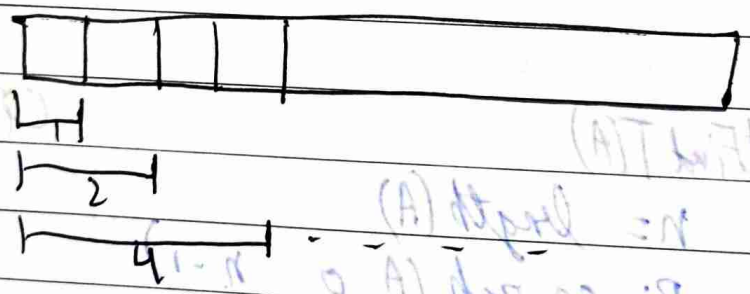
~~CleverFind(A)~~ ~~$O(\log t)$~~

~~for i = 0 to n~~

~~for (i = 1, i ≤ n, i = i * 2 + i)~~

~~if A[i-1] == 1~~

~~approx_t = i-1~~



CleverFind(A) $O(\log t)$

for(i = 1; i ≤ n; i = 2i)

~~if A[i-1]~~

if A[i-1] == 1

~~break~~ approx_t = i-1 break,

return BinSearch(A, 0, approx_t)

(st, lt, BinSearch, A)

(BinSearch, l, A)

19

$A = [\ \ \ \ \ \ \ \ \ \ ]$

To find $\min\limits_{0 \le i < j \le n} |A(i) - A(j)|$

(b) Simple Smallest Diff (A)

```
n = length (A)
~~for (i~~
ans = ∞
for (i = 0, i < n, i++)
    for (j = i+1, j < n, j++)
        ans = min (ans, AbsDiff (A[i], A[j]))
return ans
```

(c) Smart Smallest Diff (A) $\quad O(n \log n)$

```
n = length (A)
MergeSort (A)          — O(n log n)
ans = ∞
for (i = 1, i < n, i++)
    ans = min (ans, A[i] - A[i-1])
                                — O(n)
return ans
```

Page No.
Date:

1) smallprogress (A, i)

```
min = A[i], index = i          C1
for j = i to n-1               C2
    if A[j] < min              C3
        min = A[j]             C4
        index = j              C5
swap (A[i], A[index])          C6
return A                       C7
```

main ()

```
for i = 0 to n-1
    A = smallProgress (A, i)   Cin
```

asymptotic upper bound $= O(n^2)$

~~Best case analysis~~ worst case

$$C_{1n} + \sum_{i=0}^{n-1} (C_1 + (C_2 + C_3 + C_4 + C_5)(n-i) + (C_6 + C_7)$$

when array is in descending order

Best case - already in ascending order

so no C_4 & C_5

also $O(n^2)$

$$C_{1n} + \sum_{i=0}^{n-1} (C_1 + C_2 + C_3 + C_6 + C_7$$

$$C_{1n} + \sum_{i=0}^{n-1} (C_1 + (C_2 + C_3)(n-i) + (C_6 + C_7)$$

$$= O(n^2)$$

2

(a) ~~(c)~~ isPalindrome (A)

```
if len(A) == 2:                    C1
    if A[0] == A[1]:               C2
        return True                C3
if len(A) == 1:                    C4
    return True                    C5
if A[0] == A[-1]:                  C6
    return isPalindrome(A[1:-1])   T(n-2)
return False                       C7
```

~~(c) iPalin~~ $T(n) = T(n-2) + \Theta(1)$

bounded by lower upper by constants

Claim: $T(n) \le cn - d$

Proof: By induction

Base: $n = 1, 2$

$T(1) = C_1 + C_4 + C_5$

$T(2) = C_1 + C_2 + C_3$

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

$\le c(n-2) + C_1$

$= cn - d + C_1 - 2c$

$\le cn - d$, $2c > C_1$

$T(n) \le C_1 n$ $\forall n \ge n_0$

$n_0 = 1$

$n \ge n_0 + 2$

3. Find Largest (A) $\quad T(n)$

```
if length (A) = 1                    C1
        return A[0]                  C2
k = FindLargest ( A[1:] )            T(n-1)
if A[0] > k :                        C3
        return A[0]                  C4
else :                               C5
        return k                     C6
```

$$T(n) = T(n-1) + C$$
$$T(n) = O(n)$$

Base Case:

$$T(1) = C_1 + C_2$$
$$T(2) = 2C_1 + 2C_2 + C_3 + C_4 + C_5 + C_6$$

~~Induction Hypothesis:~~

Inductive Case:

$$n_0 \geq 2$$
$$T(n) \leq cn - d \quad \forall\, n \geq n_0$$

$$T(n+1) = T(n) + C$$
$$= C_1 n - d + C$$
$$= C_1(n+1) - C_1 + d$$
$$\leq C_1(n+1) - d \quad \text{if } C_1 > C$$

Hence $T(n) = O(n)$

Same for both best case & worst case

Tutorial

Pset-3

(11) HeavySubset (A, d):

(a),(c) $O(n)=O(n)$ freq = [0, 0, ---, 0] # array of size cn

$O(n)$ { for i in A: # for i = 1 to n:
freq [i] += 1

$O(1)$ value = $\left\lfloor \frac{n}{d} \right\rfloor + 1$

anslis = [] $O(1)$

$O(cn)$ { for i = 1 to cn:
if freq [i] ≥ value:
anslis = anslis + [i]

$O(1)$ return anslis

$O(n)$

(b) Claim: $i \in \mathbb{Z}^+$ $i \in$ anslis, $\Leftrightarrow$ count (i, anslis) $\geq \frac{n}{d} + 1$

Date:

(12) max $A[i]$ is not bounded
$i \in [n]$

Heavy Subset(A, d):

$O(n \log n)$

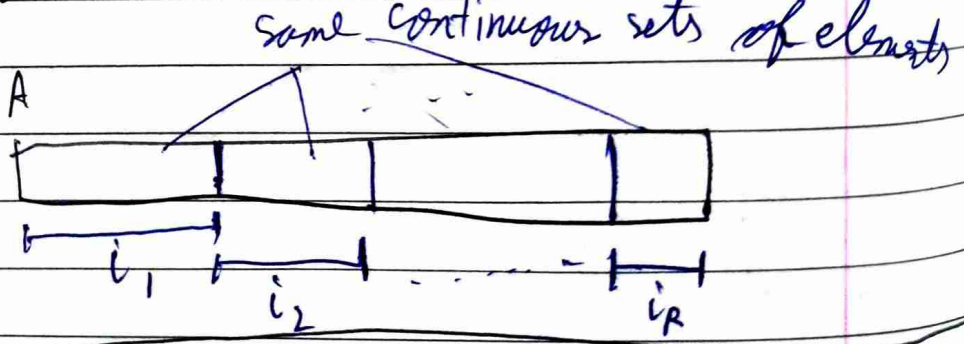
~~new~~ A = mergesort(A)

~~for~~
value = $\lfloor \frac{n}{d} \rfloor + 1$

currentnum = $-\infty$
anslis = []
added = False
count = 0
for i = 1 to n:
    if currentnum == A[i]:
        count += 1
        if count ≥ value & added == False
            anslis += [A[i]]
    else:

Claim:

sorted ~~vector~~ A

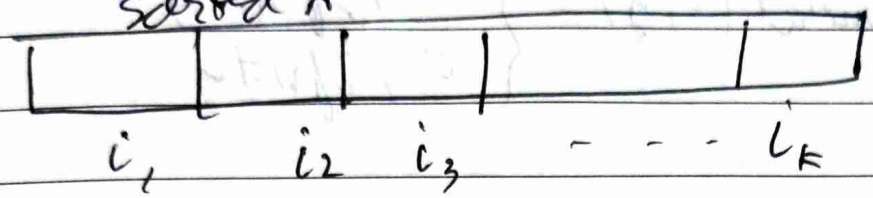


currentnum = A[i]
count = 1
added = False

Page No.:
Date:

Claim: Sorting ensures each integer $z \in A$ clusters in a continuous way

sorted A



$i_1 < i_2 < \cdots < i_k$

Claim

So if $z_1 \neq z_2 \in A$ & $z_1 < z_2$

then $\boxed{\text{if } A[j_1] = z_1 \ \& \ A[j_2] = z_2 \Rightarrow j_1 < j_2}$ S_1

Proof: As $z_1, z_2 \in A$

$\Rightarrow \exists\ j_1, j_2 \in [n] = \{1, \ldots, n\}$

A = sorted (A)

s.t. $A[j_1] = z_1$ & $A[j_2] = z_2$

since A is sorted

$\Rightarrow A[k_1] \leq A[k_2] \quad \forall\ k_1 \leq k_2$

& $z_1 < z_2 \Rightarrow S_1$

(13) [ ] - No ordering

~~Same~~

$$\text{Same}(x, y) = \begin{cases} 1, & \text{if } x = y \\ 0, & \text{if } x \neq y \end{cases}$$

<u>Claim:</u> if $I = C[i]$ $\left(I \text{ occurs} \geq \underbrace{\left\lfloor \frac{n}{2} \right\rfloor}_{mid} + 1 \right)$

(S2) If I is majority element
$\Rightarrow$ It is majority element of either

(S1) $C[1, \ldots, \underbrace{mid}_{k_1}]$ or $C[\underbrace{mid+1}_{k_2}, \ldots, n]$

<u>Proof:</u> $S_1^c \Rightarrow S_2^c$

(a) $\underset{a}{\text{count}(a)} < \left\lfloor \frac{\left\lfloor \frac{n}{2} \right\rfloor + 1}{2} \right\rfloor + 1 + \left\lfloor \frac{n - \left\lfloor \frac{n}{2} \right\rfloor - 1}{2} \right\rfloor + 1$

$< \left\lfloor \frac{n}{2} \right\rfloor + 1$

$$\cancel{k_1 + k_2 \geq \left\lfloor \frac{n}{2} \right\rfloor + 1}$$

$$\cancel{\max(k_1, k_2) \geq \left\lceil \frac{\left\lfloor \frac{n}{2} \right\rfloor + 1}{2} \right\rceil}$$

$$k_1 + k_2 \geq \left\lfloor \frac{n}{2} \right\rfloor + 1$$

$$\max\left(k_1, \left\lfloor \frac{n}{2} \right\rfloor + 1 - k_1\right) \rightarrow \max(X, C - X)$$

$$X + Y = C$$

$$\max(k_1, k_2) \geq \left\lceil \frac{\left\lfloor \frac{n}{2} \right\rfloor + 1}{2} \right\rceil$$

$$\geq \left\lfloor \frac{n}{4} \right\rfloor + 1$$

```
QUEENCAT(C)
        n = length(C)
Base case
O(1)    if n == 0 : return None
        if n == 1 : return C[1]
O(1)    else : mid = ⌊n/2⌋
O(n)        C1 = C[1, ..., mid]
O(n)        C2 = C[mid+1, ..., n]
2T(n/2)     ans1 = QUEENCAT(C1)
            ans2 = QUEENCAT(C2)
```

Page No.:
Date:

Problem (17)

or

$O(n)$ $\begin{cases} count_ans1 = count(ans1, c) \\ count_ans2 = counter(ans2, c) \end{cases}$

$O(1)$
```
if ca1 ≥ ⌊n/2⌋ + 1 : return ans1
if ca2 ≥ ⌊n/2⌋ + 1 : return ans2
else :
    return None
```

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

None use master's theorem
(free to use in midsems)

Q7.

(a) $T(n) = T\left(\frac{n}{2}\right) + C$, Assume $T(1) \le C_1$

$$T(n) = T\left(\frac{n}{2}\right) + C$$
$$= T\left(\frac{n}{4}\right) + C + C = T\left(\frac{n}{8}\right) + C + C + C$$
$$= C \cdot \lfloor \log_2 n \rfloor + T(1)$$

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

(b) $T(n) = T\left(\frac{n}{2}\right) + 3\log_2 n$

$$T(n) = T\left(\frac{n}{2}\right) + 3\log_2 n$$

$$= T\left(\frac{n}{4}\right) + 3\log_2\left(\frac{n}{2}\right) + 3\log_2 n$$

$$= T\left(\frac{n}{8}\right) + 3\log_2\left(\frac{n}{4}\right) + 3\log_2\left(\frac{n}{2}\right) + 3\log_2 n$$

$$\cancel{= \cdots = T(1) +}$$

$$= T\left(\frac{n}{8}\right) + 3\left(\log_2 n - \log_2 4\right) + 3\left(\log_2 n - \log_2 2\right) + 3\log_2 n$$

$$= T\left(\frac{n}{8}\right) + 3\left(\log_2 n - 2\right) + 3\left(\log_2 n - 1\right) + 3\log_2 n$$

$$\cancel{= T(1} = \cdots = \underbrace{T(1) + 3\log_2 n \cdot \lfloor \log_2 n \rfloor}_{O((\log n)^2)}$$

$$= 3\left[1 + 2 + 3 + \cdots + (\lfloor \log_2 n \rfloor - 1)\right]$$

$$O((\log n)^2)$$

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

Date:

(c) $T(n) = 2T\left(\frac{n}{2}\right) + 5n\sqrt{n}$

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

$$= 4T\left(\frac{n}{4}\right) + 2\cdot 5\left(\frac{n}{2}\right)\sqrt{\frac{n}{2}} + 5n\sqrt{n}$$

$$= 8T\left(\frac{n}{8}\right) + 4\cdot 5\left(\frac{n}{4}\right)\sqrt{\frac{n}{4}} + 2\cdot 5\cdot\frac{n}{2}\sqrt{\frac{n}{2}} + 5n\sqrt{n}$$

$$\cancel{= \cdots + \lfloor \log_2 n \rfloor}$$

$$= \cdots = 2^{\lfloor \log_2 n \rfloor} \cdot T(1)$$

$$+ 5n\sqrt{n}\left(1 + \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{4}} + \cdots + \frac{1}{2^{\lfloor \log_2 n \rfloor - 1}}\right)$$

$$\leq n \cdot c_1 + 5n\sqrt{n}\underbrace{\left(1 + \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{4}} + \cdots\right)}_{\text{infinite sum}}$$

$$= nc_1 + 5n\sqrt{n} \times \frac{1}{1 - \frac{1}{\sqrt{2}}}$$

$$\cancel{= n\sqrt{n}}$$

$$= nc_1 + 5n\sqrt{n} \cdot \sqrt{2}(\sqrt{2}+1)$$

$$= O(n\sqrt{n})$$

(d) $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log_2 n}$

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

$$= 4T\left(\frac{n}{4}\right) + \frac{2 \cdot \frac{n}{2}}{\log_2 \frac{n}{2}} + \frac{n}{\log_2 n}$$

$$= 8T\left(\frac{n}{8}\right) + \frac{4 \cdot \frac{n}{4}}{\log_2 \frac{n}{4}} + \frac{2 \cdot \frac{n}{2}}{\log_2 \frac{n}{2^2}} + \frac{n}{\log_2 n}$$

$$= 8T\left(\frac{n}{8}\right) + \frac{n}{\log_2 n - 2} + \frac{n}{\log_2 n - 1} + \frac{n}{\log_2 n}$$

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

$\cancel{\log_2 n - \lfloor \log_2 n \rfloor > 1}$

$$\log_2\left(\frac{n}{2^{\lfloor \log_2 n \rfloor - 1}}\right) = \log_2\left(\frac{2^{\log_2 n}}{2^{\lfloor \log_2 n \rfloor - 1}}\right)$$

$$= \log_2\left(2^{\log_2 n - \lfloor \log_2 n \rfloor + 1}\right)$$

$$\cancel{\Rightarrow T(n) \le 2^{\lfloor \log_2 n \rfloor} T(1) + n\left(1 + \frac{1}{2}\right)}$$

$$\Rightarrow T(n) \le 2^{\lfloor \log_2 n \rfloor} T(1) + n\left[1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\log_2 n}\right]$$

Use: $1 + \frac{1}{2} + \cdots + \frac{1}{k} \le \log_e k$

$$\Rightarrow T(n) \le nc_1 + n \log_e(\log_2 n)$$

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

<u>Proof of this claim</u>

<u>Base Case</u>:

$n = 2$: $n \log_2(\log_2 n) = 0$

$n = 3$: $n \log_2(\log_2 n) =$ "some small constant"

$n = 4$: $n \log_2(\log_2 n) = 4$

We will start our base at $n = 4$
[Can start anywhere generally]

$$T(4) = 2T(2) + \frac{4}{\log_2 4} = 2T(2) + 2$$

$$= 4T(1) + \frac{2}{\log_2 2} + 2$$

$$= 4T(1) + 2 + 2 = 4c' + 2 + 2$$

$$= \cancel{O(1)}$$

$$= O(1) = O(4 \log \log c_1)$$

<u>Induction Hypothesis</u>

$\forall\ 4 \le k < n,$

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

Induction step -

let $n \ge 5$

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

$\frac{n}{2} < n$

$$\Rightarrow T(n) = \cancel{2}c \cdot \frac{n}{\cancel{2}} \log\left(\log \frac{n}{2}\right) + \frac{n}{\log_2 n}$$

$$= \cancel{\ } = cn \log(\log n - \log 2) + \frac{n}{\log_2 n}$$

$$\le cn \log(\log n) + \cancel{c' \log 2 \cdot n} + \frac{n}{\log_2 n}$$

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

Date:

(8.)

$L = [13, 29, 41, 45, 91]$ — i_L

$R = [23, 27, 38, 79, 96]$ — i_R

I II III

$M = [13, 23, 29, \ldots]$

k

Partial sol^n: First k elements of M filled.

first i_L elements from L & first i_R elements from R entered in M.

~~Small step~~

→ Above, $k = 3$, $i_L = 2$, $i_R = 1$

→ $L[i_L]$ is untouched, $R[i_R]$ is untouched

→ Small Step: Enter $\min\{L[i_L], R[i_R]\}$ in M.

$M = [13, 23, 29, \boxed{27}, \ldots]$

→ What if $i_L = l$? → Always fill from R.

~~What if $i_R = r$?~~ " " " L.

```
MergeHelper (iL, iR, l, r, L, R)
{
    if (iL == l)
        return ~~flag~~ = 1   — indicates go with
                               right subarray
    if (iR == r)
        return ~~flag~~ = 0   — " " "
                               left subarray
    if (L[iL] ≤ ~~return~~ R[iR])
        return 0
    return 1
}

Merge (l, r, L, R)
{
    iL = 0 , iR = 0

    M = empty array
    for (i = 0 ; i < l+r ; i++)
        flag = MergeHelper (iL, iR, l, r, L, R)
        if (flag == 0) :
            M[i] = L[iL]
            iL += 1
        else
            M[i] = L[iR]
            iR += 1
    return M
```

Lecture - 16

Burglar robbing a store, with a knapsack of capacity W kg.
There are ~~items~~ n items $I_1, I_2, \ldots, I_n$, with weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ respectively.

0-1 Knapsack, without repetition
(The burglar can either take an item or not, not fraction of a knapsack.

W = 55 kg

Item	I_1	I_2	I_3
Height	10	22	30
Value	60	100	120

W/O repetition
- $I_2 + I_3$: value = 220

With repetition
- $5I_1$: value = 300

0-1 Knapsack with repetition

Inputs: - The capacity W of knapsack, a positive integer
- Item types $I_1, I_2, \ldots, I_n$
- Weights $w_1, w_2, \ldots, w_n$, each a positive integer
- Value $v_1, v_2, \ldots, v_n$, each a positive integer

Output:

Goal: Find the maximum sum of values of a collection of items whose weights sum up to atmost W. Each type may be chosen more than once.

Some attributes to try projection with:

1. Whether a solution contains an item type I_i
2. The number of different pieces (not types) in a solution.

DP by projection onto item types

S: set of all feasible solutions.

for $1 \le i \le n$: S_i

is the set of all solutions that contain at least one item of type I_i

In general, $S_i \cap S_j \neq \phi$ for $i \neq j$

For $X \subseteq S$

let $OPT(X)$ be the optimum value of a solution in set X.

We want to find $OPT(S)$.

$$OPT(S) = \max_{i=1}^{n} OPT(S_i)$$

$OPT(S_i) = v_i +$ the optimum solution for a knapsack of capacity $(W - w_i)$

---

1. Create an array maxVal [ ] of size $W+1$
   - maxVal [w] = the optimum sol^n for a knapsack of capacity w
2. maxVal [0] = 0
3. Use the recurrence to fill maxVal [w] in increasing order of w, ~~from~~ from 1 to W.

initially filled with NIL value.

```
maxVal[0] = 0

for w = 1 to W:
    j maxvalue = -1
    for i = 1 to j:
        ivalue = Value[i] + maxVal[i - j]
```

```
RepKnapSackDP (W, Values, Weights):
    n = length(Values)
    maxVal = an array of length W+1
             initial values to NIL.

    maxVal[0] = 0
    for w = 1 to W:
        wMaxVal = 0
        for i = 1 to n: if Weights[i] ≤ w {
            iValue = Values[i] +
                     maxVal[w - Weights[i]]
            if iValue > wMaxValue:
                wMaxValue = iValue }
        maxVal[w] = wMaxValue
    return maxVal[W]
```

$O(nW)$ time.

DP by projecting onto the <u>number of pieces</u> in a solution

S : set of all feasible solutions.

For $p \in \mathbb{N}$, let S_p be the set of all solutions that contain <u>exactly</u> p pieces.

Here, $S_i \cap S_j = \phi$ if $i \neq j$

For $X \subseteq S$, let $OPT(X)$ be the optimum value of a solution in set X.

We want to find $OPT(S)$.

The maximum value of p, ~~say~~ say $N = \left\lfloor \frac{W}{\min_{i=1}^{n} w_i} \right\rfloor$

$1 \leq p \leq N$

$$OPT(S) = \max_{p=1}^{N} OPT(S_p)$$

$$OPT(S_p) = \max_{i=1}^{n} \left(v_i + \text{the optimum value of all solutions with exactly } (p-1) \text{ pieces, and knapsack capacity } (W - w_i) \right)$$

The pair (capacity, number of pieces) "dropped" the number of pieces dropped by 1.

So we compute the OPT values in increasing order of the number of pieces, and for each such number p, in increasing order of knapsack capacities.

Given: W, Values, Weights
$n = \ln(\text{Values})$

Output

1. Compute $N = \left\lfloor W / \min_{i=1}^{n} w_i \right\rfloor$
$= \left\lfloor W / \min(\text{Weights}) \right\rfloor$

2. Create an array maxVal $[0 \ldots N][0 \ldots W]$

- maxVal $[i][s]$ will store the maximum value that can be obtained with a knapsack of capacity s, containing exactly i pieces.

3. maxVal $[0][s] = 0$ for $0 \le s \le W$
maxVal $[i][0] = 0$ for $0 \le i \le N$

4. Compute maxVal[p][w] in increasing order of p, and for each p, in increasing order of w.

~~RepN~~

```
RepKnapsack (W, Values, Weights)
    n = len(Values) - 1
    minWt = min(Weight)
    N = floor( W/minWt)

    maxVal = an (N+1) x (W+1) array,
            initialized to 0.

    for p = 1 to N:

        for w = 1 to W,
            preMaxVal = 0
            for i = 1 to n:
                iValue = Values[i]

                iWeight = Weights[i]
                if iWeight ≤ w:
                    newVal = iValue +
                        maxVal[p-1]
                        [w - iWeight

                if newVal > preMaxVal
                    ~~newVal = preMaxVal~~
                    preMaxVal = newVal
```

maxVal[p][w] = preMaxVal

return the maximum ~~an~~ among the values

maxVal[i][W] , for $1 \le i \le N$

$O(nW^2)$ time

NO repetition allowed

Projecting onto ~~prefix~~ prefixes of the input

<u>Items:</u> $\underbrace{I_1, I_2, \ldots, I_n}_{S}, \ldots, I_N$

For $1 \le i \le n$ let S_i be the set of all solutions which pick items only from the "prefix" $\{I_1, I_2, \ldots, I_i\}$

- Equivalently: a solution belongs to $S_i \Leftrightarrow$ it contains no item I_j for any $j > i$.

$$OPT(S) = OPT(S_n, W)$$

$$\cancel{OPT(S_n) = \max\{(v_n + OPT(S_{n-1}, W}$$

$$OPT(S_n, W) = \max\{(v_n + OPT(S_{n-1}, W - w_n)), OPT(S_{n-1}, W)\}$$

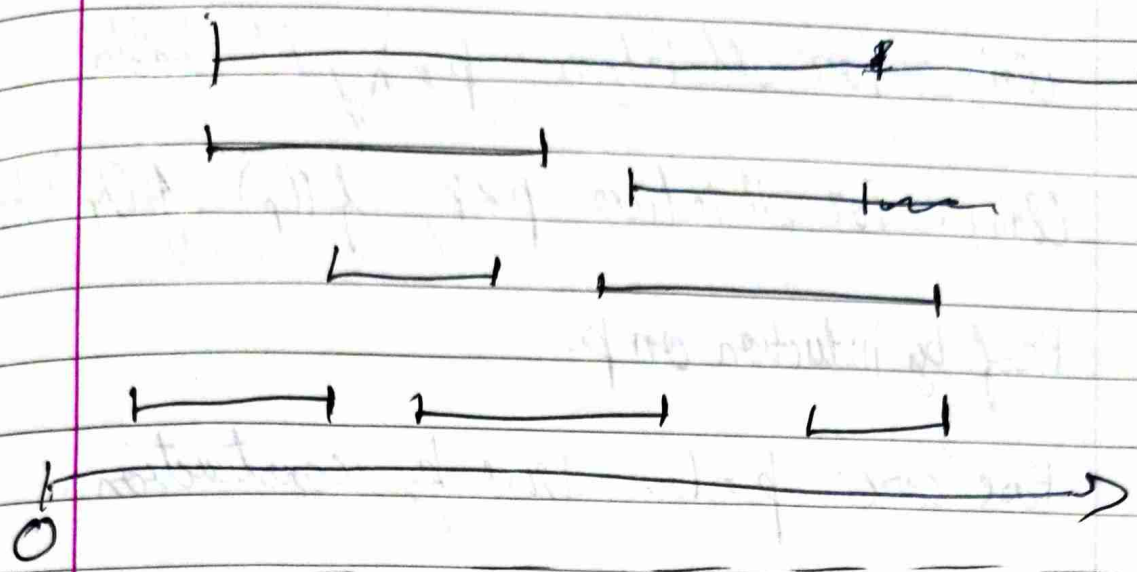
Lecture - 17

Interval Scheduling

Input: A set of n requests, $R = \{(s(1), f(1)), (s(2), f(2)), \ldots, (s(n), f(n))\}$ where the $s(i), f(i)$ are the natural numbers, & $s(i) < f(i)$ holds for all i.

- $s(i), f(i)$ are the starting & ~~fin~~ finishing times for requests.
- Request i can be satisfied by allotting the resource from $s(i)$ to $f(i)$.
- A subset of requests is compatible if they all can be satisfied together.

Goal: Find a largest subset of R that can be satisfied together.



A greedy, optimal algorithm for Interval Scheduling.

1. Let $A = \phi$ // The set of accepted requests
2. While R is not-empty, do
   - Find a request $i \in R$ that has the least $f(i)$ value.
   - $A \leftarrow A \cup \{i\}$
   - Delete all requests in R that are in conflict with i.
3. return A.

Let $A = \{i_1, i_2, \ldots, i_k\}$ in the order in which the algorithm added the requests in A.

If possible, let $O = \{j_1, j_2, \ldots, j_n\}$ be a longer set of compatible requests, sorted in the order of their starting times.

~~Claim: For all indices $p \le k$, it holds~~

Claim: For all indices $p \le k$, $f(i_p) \le f(j_p)$ holds

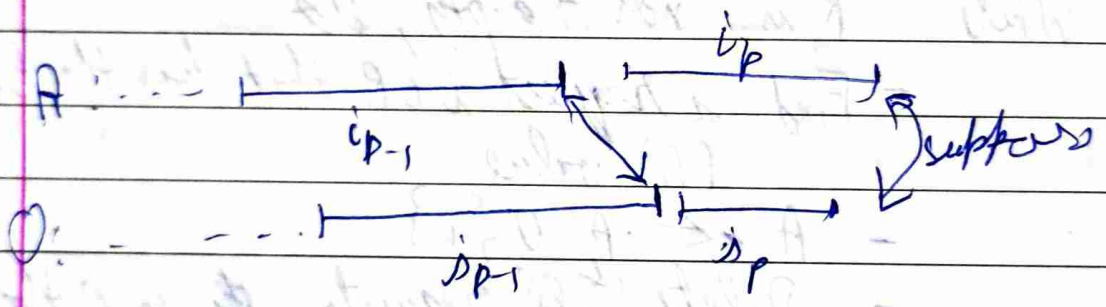
Proof by induction on p.

Base case: $p = 1$. True by construction.

<u>Inductive step</u> - To show: $f(i_p) \le f(j_p)$

From the inductive assumption -

$$f(i_{p-1}) \le f(j_{p-1})$$



~~j_p is available because~~

If $f(j_p) < f(i_p)$, then we would have chosen j_p. Also j_p is available because non-intersecting requests before $f(i_{p-1})$ are available.

$$f(i_{p-1}) \le f(j_{p-1}) < f(j_p) < f(i_p)$$

Then we would have chosen j_p by construction.

What happens ~~a~~ if $m > k$?

$\Downarrow$

~~There are~~

There is a i_{R+1} in O &

~~$s(i_{R+1}) > f(i_R) \geq f(i$~~

$f(i_R) \leq f(i_j) < s(i_{R+1})$

Again i_{R+1} is available because ~~the~~ requests after i_R are still available. So we could have selected i_{R+1} (or ~~there~~ other requests ~~in~~ among existing ones).

Lecture - 18

I_a : the interval which ends first

Let X_a be the set of all intervals which have a conflict with I_a, including I_a.

Claim: Any optimal collection of non-conflicting intervals must pick at least one element of X_a

Suppose Y is an optimal collection & $Y \cap X_a = \phi$
Then we can choose an interval from X_a that extends Y (choosing I_a).

Is it true that any solution can pick at most one interval from X_a ?

Scheduling to minimizing Latency

Input: Non-negative integer S (time from which the resource can be used), ~~and~~ a set of n Requests $R = \{r_1 = (t_1, d_1), \ldots, r_n = (t_n, d_n)\}$. The t_i, d_i are natural numbers and $t_i \le d_i$ holds for all i.

- There is a single resource available to use starting at time S.

~~There is a~~

- (t_i, d_i) are the duration & deadline for request r_i.

Goal: Find an arrangement of all the requests to the resource, which minimizes the ~~maximum latency~~. maximum lateness.

At most one request can be assigned to the resource at any point in time. Each request, once started, has to end before another request can start using the resource.

Let $\{s_1, s_2, \ldots, s_n\}$ be the starting times for an assignment. Then the finishing times are:

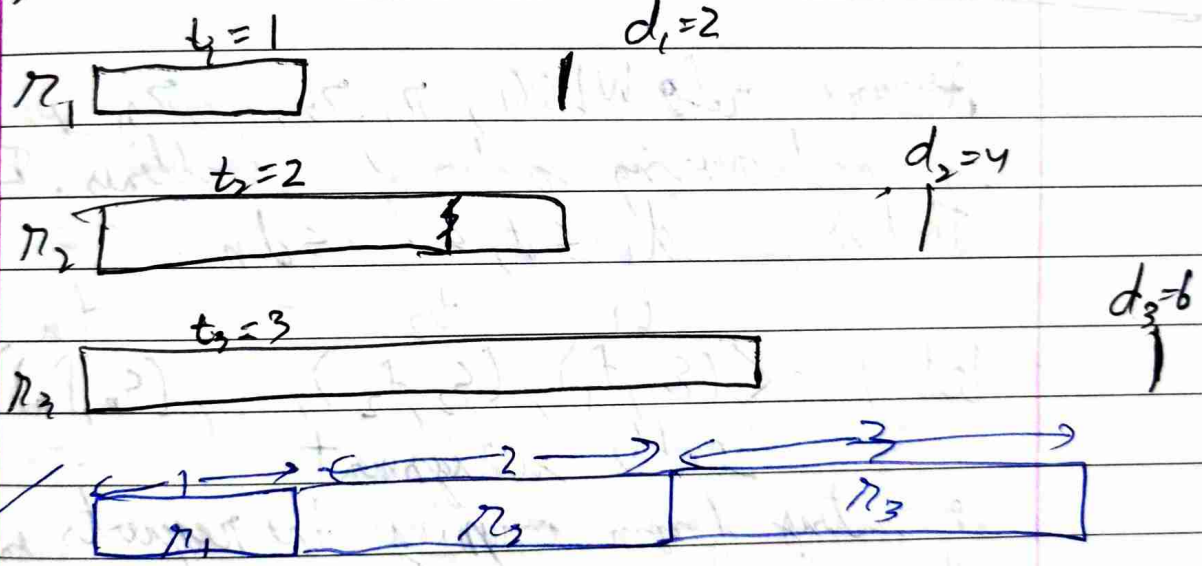
$f_1 = s_1 + t_1$, $f_2 = s_2 + t_2$, ..., $f_n = s_n + t_n$.

Request r_i is <u>late</u> if $f_i > d_i$.

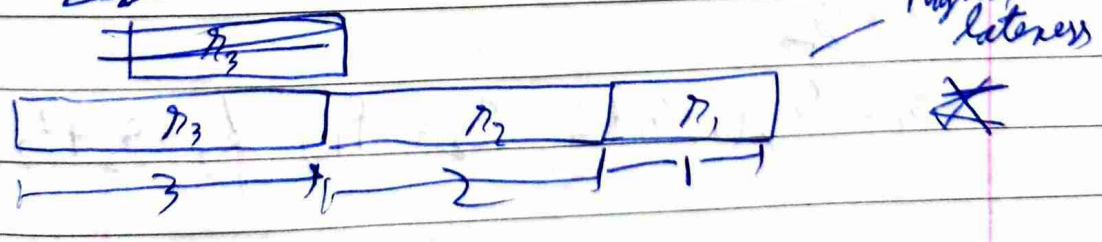
The <u>lateness</u> of such an r_i is ~~$d_i - f_i$~~ $f_i - d_i$.

The lateness of non-late requests is defined to be zero.

$s = 0$



~~But~~



Higher lateness

Picking the <u>shortest request</u>, first does not work.

<u>Eg.</u> $R = \{(1, 100), (10, 10)\}$

Picking the request with the least <u>slack time</u> $(d_i - t_i)$, doesn't work.

$R = \{(1, 2), (10, 10)\}$

Assume ~~wlo~~ WLOG, $r_1, r_2, \ldots, r_n$ is in non-decreasing order of deadlines. R_i
That is, $d_1 \le d_2 \le \cdots \le d_n$.

$t_1 \quad t_2 \quad \cdots \quad t_n$

Let $A = \langle (s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n) \rangle$ be the assignment resulting from assigning the requests in this order.

$s_1 = s, f_1 = s_1 + t_1, s_2 = f_1, f_2 = s_2 + t_2, \ldots$

Claim: There is an optimum assignment ~~which~~ ~~to~~ with ~~no~~ no "idle time".

For every gap in an assignment, we can shift all the "right" requests to the left to

fill that gap & the maximum lateness will either reduce or remain the same.

An <u>inversion</u> in a schedule A' is a pair of requests r_i, r_j where $d_j < d_i$ & r_i is scheduled <u>before</u> r_j.

Claim: Schedule A computed by the greedy algorithm has no inversion.
└ By construction of our algorithm.

<u>Claim:</u> <u>All</u> schedules with no gaps & no inversions have the exact ~~so~~ same maximum lateness.

These two schedules differ only in relative order of ~~requests~~ requests with same deadlines.

All those requests with the same deadline can be interchanged among them ~~a~~, since they all have the same deadline.

O: An optimal schedule with no gaps.

Claim: If O has an inversion, then there is a pair of requests r_i, r_j such that $d_j < d_i$, and r_j is immediately after r_i

in O.

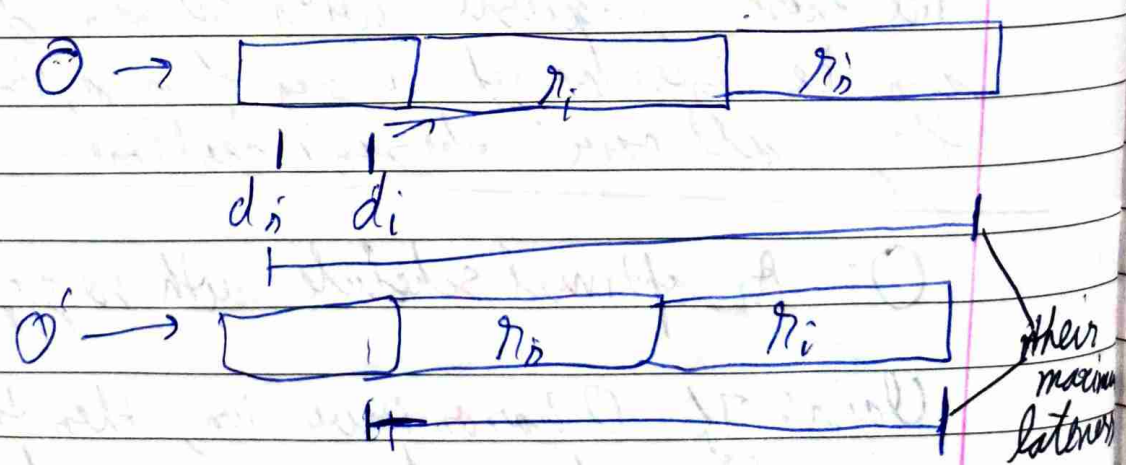
Proof: Let r_a, r_b form an inversion in O, with $d_a > d_b$, and r_a being somewhere before r_b in O.
Scan from r_a to the right, to find the first pair of requests ~~the~~ where the deadline drops.

$$d_i > d_j$$

$$\text{---} \; r_i, r_j, \text{---}$$

Claim: Let O' be the schedule obtained by exchanging the order of r_i, r_j from O above. Then -

(1) O' has one fewer inversion than O
(2) The maximum lateness of O' is ~~the~~ at most as that of O.



Lecture -19

Randomized algorithms

Monte Carlo: Runs fast, may give wrong output.

Las Vegas: Always gives correct output, sometimes take a long time.

Randomized Quick Sort runs in $O(n \log n)$ time on any input size n, with high probability.

A stronger statement

~~Deter~~

Simple Deterministic Quick Sort runs in $O(n \log n)$ on randomly ordered input, with high probability.

Hire Assistant (n)

```
best = 0 // Dummy candidate, least qualified
for i = 1 to n:
    interview candidate i
    if candidate i is better than candidate best:
        best = i
        hire candidate i // after firing the existing assistant.
```

The total cost, if m candidates are hired: $c_i \cdot n + c_h \cdot m$

c_i :- cost of interviewing a candidate
c_h :- cost of firing + hiring

$c_i \cdot n$ is constant, cannot be optimized.
The worst cost is $c_i \cdot n + c_h \cdot n$ $(m = n)$

X_i : the indicator random variable that candidate i is hired.

- 0 if candidate i is <u>not hired</u>
- 1 if candidate i is <u>hired</u>.

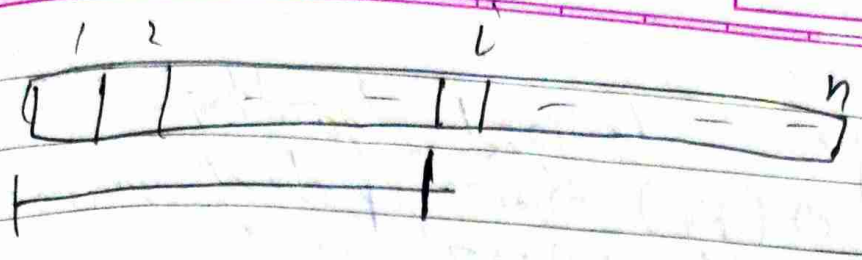
Let X be the random variable whose value is the total number of candidates that are hired.

$$\text{Then } X = \sum_{i=1}^{n} X_i$$

$$E[X] = \sum_{i=1}^{n} E[X_i]$$

$$= \sum_{i=1}^{n} (\text{The probability that } X_i = 1)$$

What is $P(X_i = 1)$?



$X_i = 1$ only when i is better than $1 \le j \le i-1$.

We are doing this over all permutations

Hence $Pr(X_i = 1) = \frac{1}{i}$

$$\Rightarrow E[X] = \sum_{i=1}^{n} \frac{1}{i} = \left(\frac{1}{1} + \frac{1}{2} + \dots + \frac{1}{n}\right)$$
$$= O(\log n)$$

Lecture - 20

Verifying Matrix Multiplication

(Probability & Computing, Chapter 1, by Mitzenmacher & Upfal)

Input: Three $n \times n$ integer matrices A, B, C.

Qn: Is $A \cdot B = C$?

Simple algorithm :- $O(n^3)$ time

Fancy (impractical) :- $O(n^{2.7})$, $O(n^{2.3})$

Simple randomized algorithm –

$O(kn^2)$ time, prob. of success $= 1 - \frac{1}{2^k}$

Freivald, 1977

1. Pick an n-bit array vector $\vec{r}$ uniformly at random from $\{0,1\}^n$. $\leftarrow O(n)$
2. Compute the product $AB\vec{r}$

$O(n^2)$ – Compute $X = B\vec{r}$

– Compute $AX = A(B\vec{r}) = AB\vec{r}$

3. Compute $C\vec{r}$ $\leftarrow O(n^2)$
4. If $AB\vec{r} = C\vec{r}$: return Yes $\leftarrow O(n)$

else : return No.

Let α be an arbitrary, but fixed, vector in $\{0,1\}^n$.

Let x be a vector picked uniformly at ~~be~~ randomly from $\{0,1\}^n$ (somehow)

Let y be an n-bit vector that we picked by tossing a fair coin n times.

Then $P[\alpha = x] = P[\alpha = y]$

$\qquad\qquad\qquad\qquad \overset{\shortparallel}{\frac{1}{2^n}} \qquad\qquad\qquad \overset{\shortparallel}{\left(\frac{1}{2}\right)^n}$

Entirely it takes $O(n^2)$ ~~FT~~

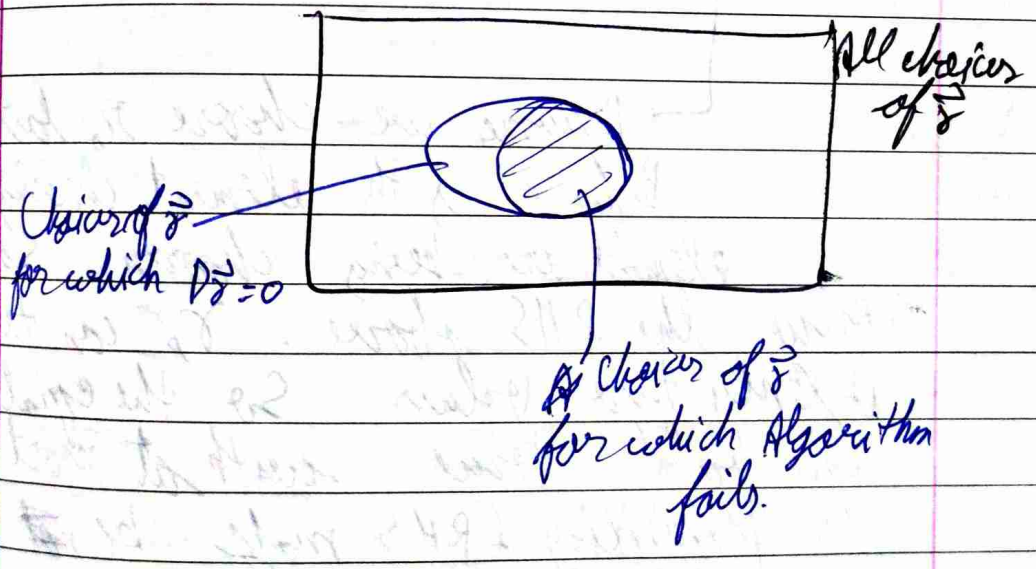
If answer comes out to be No, then $AB \neq C$ is correct everytime. So failure ~~happens~~ can happen for Yes.
This is called a one-sided error.

Failure only when $AB \neq C$, and algorithm returns Yes.

Let $D = AB - C$ (we don't know D)

Then $D \neq 0$ and the algorithm fails only when $D\vec{r} = 0$.

$$\Pr[\text{Algorithm fails}] \leq \Pr[D\vec{r} = 0]$$



Since $D \neq 0$, there is some i, j s.t. $d_{ij} \neq 0$.

$$\cancel{D\vec{r} = 0 \iff \sum_{k=1}^{n} d_{ik} r_k = 0}$$

$$D\vec{r} = 0 \iff \sum_{k=1}^{n} d_{ik} r_k = 0$$

$$\Updownarrow$$

$$r_j = \frac{-\sum_{\substack{k=1 \\ k \neq j}}^{n} d_{ik} r_k}{d_{ij}}$$

$$\Pr[D\vec{r} = 0] = \Pr\left[r_j = \frac{-\sum_{\substack{k=1 \\ k \neq j}}^{n} d_{ik} r_k}{d_{ij}}\right]$$

$$\leq \frac{1}{2}$$

Assume we choose r_j last. Before j th element choosing, all element are being chosen. So we know the RHS above. r_j can take only two values. So the equality can hold true with at most 1/2 probability (RHS may be non-binary).

Repeat the algorithm k times, with a fresh $\vec{r}$ each time.

Probability of failing all k times $\leq \frac{1}{2^k}$

$\Rightarrow$ Probability of ~~to~~ passing $\geq 1 - \frac{1}{2^k}$

Lecture 21

Say we run the round a 100 times, and all rounds return Yes.

We prove: the probability that the algorithm gives a wrong answer is

$$\leq \frac{1}{2^{100}}$$

Given these outputs (100 times Yes), what is the probability that $AB = C$?

- Is this the same as the probability that the algorithm gives a correct answer after 100 rounds?

Baye's Law: Let $E_1, E_2, \ldots, E_n$ be mutually exclusive and exhaustive events. Let B be an arbitrary event. Then for any i:

$$P[E_i \mid B] = \frac{P[E_i \cap B]}{P[B]} = \frac{P[B \mid E_i] \times P[E_i]}{\sum_{i=1}^{n} P[B \mid E_i] \cdot P[E_i]}$$

3 coins C_1, C_2, C_3. Two are fair, one is biased. The biased coin turns up head $2/3$ of the time.

~~We know~~

We toss each coin once

C_1 & C_2 turns up heads each

C_3 turns up tails

~~let E_i be the (prior) pro~~

Let E_i be the event that C_i is biased.

Then $P[E_i] = \frac{1}{3}$ (prior)

Let B be the event that C_1, C_2 turn up heads & C_3 turns up tails.

We want to find:- $P[E_1|B]$

$$P[B|E_1] = \cancel{\tfrac{2}{3}} \times \frac{2}{3} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{6}$$

$$P[B|E_2] = \frac{1}{2} \times \frac{2}{3} \times \frac{1}{2} = \frac{1}{6}$$

$$P[B|E_3] = \frac{1}{2} \times \frac{1}{2} \times \frac{1}{3} = \frac{1}{12}$$

$$P[E_1|B] = \frac{P[B|E_1] \cdot P[E_1]}{P[B]}$$

$$= \frac{P[B|E_1] \cdot \cancel{P[E_1]}}{P[B|E_1] \cdot \cancel{P[E_1]} + P[B|E_2] \cdot \cancel{P[E_2]} + P[B|E_3] \cdot \cancel{P[E_3]}}$$

$$= \frac{\frac{1}{6}}{\frac{2}{6} + \frac{2}{6} + \frac{1}{12}} = \frac{\frac{1}{6}}{\frac{5}{12}}$$

$$= \frac{1 \times 12}{6 \times 5} = \frac{2}{5}$$

$$> \frac{2}{6} = \frac{1}{3}$$

$$AB \overset{?}{=} C$$

Let E be the event that $AB = C$ holds, and $\bar{E}$ be the event that $AB \neq C$ holds.

$$P[E] = P[\bar{E}] = \frac{1}{2}$$

Let F be the event that one round of Freivald returned Yes.

We want to find :- $P[E|F]$

$P[F|E] = 1$

$P[F|\bar{E}] \leq \frac{1}{2}$

$$P[E|F] = \frac{P[F|E] \cdot P[E]}{P[F|E] \cdot P[E] + P[F|\bar{E}] \cdot P[\bar{E}]}$$

$$\geq \frac{\frac{1}{2}}{1 \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2}}$$

$$= \frac{\frac{1}{2}}{\frac{3}{4}} = \frac{2}{3}$$

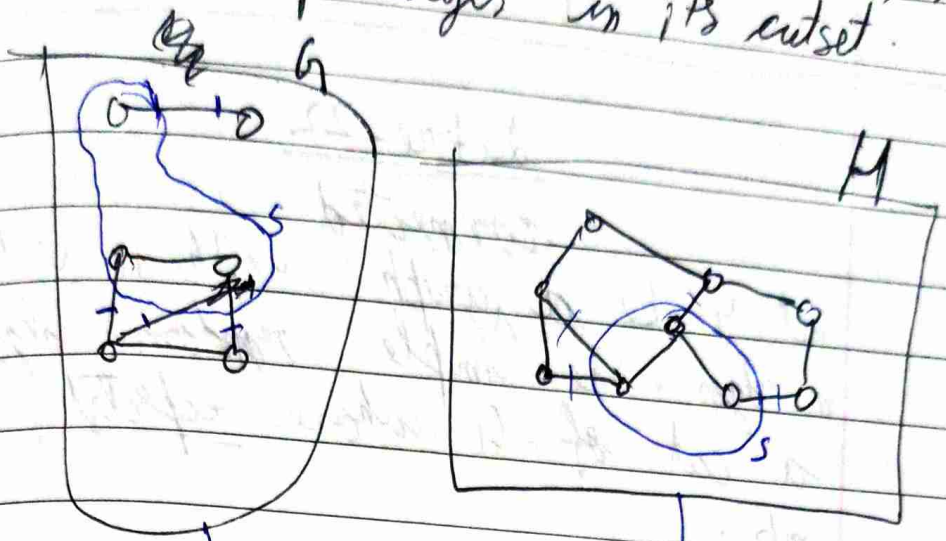
After i rounds of the algorithm returns Yes, the probability that $AB = C$ holds is $\geq 1 - \frac{1}{2^i + 1}$

Let $G = (V, E)$ be an undirected graph with n vertices, m edges.

- A <u>cut</u> of G is <u>any</u> partition of V into two <u>non-empty</u> parts, Say S & $V \setminus S$.
  Denoted $(S, V \setminus S)$

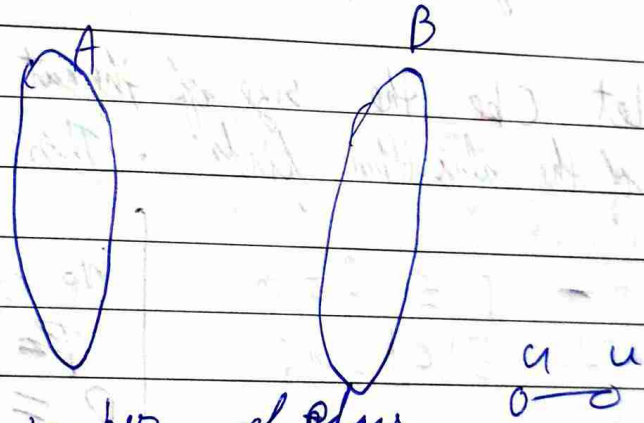
- The <u>cutset</u> corresponding to the cut $(S, V \setminus S)$ is the set of all edges that <u>cross</u> the cut.

- The size of cut $(S, V \setminus S)$ is the number of edges in its cutset.



Size = 4 of cut

Size = 4 of cut



Expected number of edges that cross the cut $= \frac{m}{2}$

Let the edges be $e_1, e_2, \ldots, e_m$. For edge e_i, ~~that~~ let X_i be an indicator random variable that is 1 if edge e_i cross the cut, and 0 otherwise.

Then $E[X_i = 1] = \frac{1}{2}$

Let $C = \sum_{i=1}^{m} X_i$. Then C is a random variable that denotes no of edges that cross the cut.

$$E[C] = \sum_{i=1}^{m} E[Y_i] = \sum_{i=1}^{m} \frac{1}{2} = \frac{m}{2}$$

Lecture - 22

Let G be a connected graph with n vertices and m edges. The simple random algorithm gives a cut of G whose expected size is $m/2$.

What is the probability that one round of this algorithm gives us a cut of size $\geq \frac{m}{2}$

Let C be the size of the cut that one round of the algorithm finds. Then:

- $1 \leq C \leq m$
- $E[C] = m/2$

We want to find:

$$P = P\left[C \geq \frac{m}{2}\right]$$

$$\cancel{E[C] = \sum_{i=1}^{\left(\frac{m}{2}-1\right)}}$$

$$E[C] = \sum_{i=1}^{\left(\frac{m}{2}-1\right)} i \cdot P[C=i] + \sum_{i=\frac{m}{2}}^{m} i \cdot P[C=i]$$

$$\sum_{i=1}^{\left(\frac{m}{2}-1\right)} i \cdot P[C=i] \leq \left(\frac{m}{2}-1\right) \sum_{i=1}^{\left(\frac{m}{2}-1\right)} P(C=i)$$

$$= \left(\frac{m}{2}-1\right)(1-P)$$

$$\Rightarrow E[C] = \sum_{i=1}^{\left(\frac{m}{2}-1\right)} i\,P[C=i] + \sum_{i=\frac{m}{2}}^{m} i \cdot P[C=i]$$

$$\leq \left(\frac{m}{2}-1\right)(1-P) + mP$$

$$= \frac{m}{2} - 1 - \frac{mP}{2} + P + mP$$

$$= \frac{m}{2} - 1 + P + \frac{mP}{2}$$

$$\Rightarrow \cancel{\frac{m}{2}} \leq \cancel{\frac{m}{2}} - 1 + P + \frac{mP}{2}$$

$$1 \leq P\left(1 + \frac{m}{2}\right) \Rightarrow \boxed{P \geq \frac{1}{1+\frac{m}{2}}}$$

$\Rightarrow$ The probability of failure $\leq 1 - \dfrac{1}{\left(\frac{m}{2}+1\right)}$

"The most useful inequality in randomized algorithms"

$$\boxed{1 - x \leq e^{-x}}$$

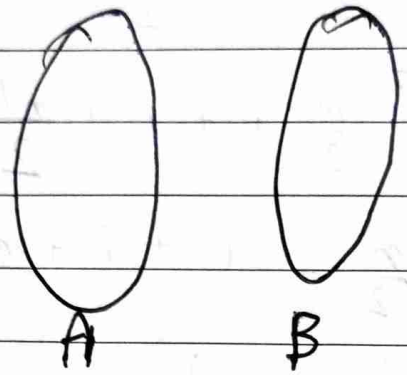
$$Pr[1 \text{ failure}] \leq 1 - \frac{1}{\frac{m}{2}+1} \leq e^{-\frac{1}{\frac{m}{2}+1}}$$

$$Pr\left[\left(\frac{m}{2}+1\right) \text{ failure}\right] \leq \left(e^{-\frac{1}{\frac{m}{2}+1}}\right)^{\frac{m}{2}+1} = \frac{1}{e} < \frac{1}{2}$$

$$\Rightarrow Pr\left[k\left(\frac{m}{2}+1\right) \text{ failure}\right] \leq \frac{1}{2^k}$$

Let $v_1, v_2, \ldots, v_n$ be the order in which the randomized algorithm picks up the vertices. Let $x_1, x_2, \ldots, x_n$ the decisions that the algorithm makes (after tossing coins. Then,

$$x_i \in \{A, B\}$$



Let C be the size of the final cut

$$E[C \mid \underbrace{x_1, x_2, \ldots, x_k}_{\text{first } k \text{ decisions made}}]$$

= the expected value of the final cut C, given that the algorithm has placed the first k vertices at $x_1, x_2, \ldots, x_k$

$$= \frac{1}{2} E[C \mid x_1, \ldots, x_k \,\&\, x_{k+1} = A] + \frac{1}{2} E[C \mid x_1, \ldots, x_k \,\&\, x_{k+1} = B]$$

$$2E[C \mid x_1, \ldots, x_k] = E[C \mid x_1, \ldots, x_{k+1} = A] + E[C \mid x_1, \ldots, x_{k+1} = B]$$

$$\cancel{\Rightarrow E[C \mid x_1, \ldots, x_k] = \max}$$

$$\Rightarrow E[C \mid x_1, \ldots, x_k] \le \max\left\{ \begin{array}{l} E[C \mid x_1, \ldots, x_k, x_{k+1} = A] \\ E[C \mid x_1, \ldots, x_k, x_{k+1} = B] \end{array} \right\}$$

$$\cancel{E[C \mid x_1, x_2, \ldots, x_n] \ge E[C \mid x_1]}$$

$$\cancel{E[C \mid x_1], \; E[C \mid x_1, x_2], \; \ldots}$$

$$\underline{E[C \mid x_1]} \le \cdots \le E[C \mid x_1, x_2, \ldots, x_n]$$

$$\parallel$$

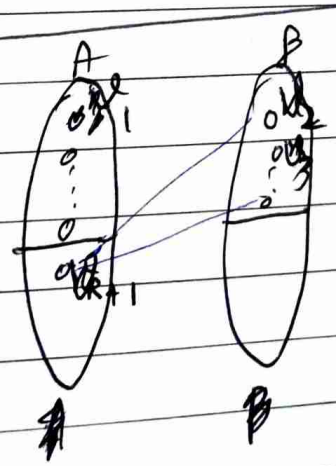
the size of the final cut

$$E[C \mid x_1 = A] = E[C \mid x_1 = B] = E[C]$$

$$E[C] \le \text{the size of the final cut}$$

$$\parallel$$

$$\frac{m}{2}$$



Derandomized – When $x_{k+1} = A$, count no. of cuts between v_{k+1} & already placed vertices in B. Same when $x_{k+1} = B$. Then choose the one with higher no. of such cut.