★ Recurrence Theorem

Let $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$, for constants $a > 0$, $b > 1$, $d \geq 0$.

then

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

$1 + a + a^2 + \ldots + a^{\log_b n} \leq O(a^{\log_b n}) = O(n^{\log_b a})$

★ Three parts of a typical divide and conquer algorithm.

Divide: Convert the i/p instance into one or more smaller instances of the same problem.

Conquer: 1. If an input instance is really small then solve it by "inspection"
2. Otherwise solve it using this self same algorithm.

Combine: Use the solution(s) of the smaller instances to construct a solution for the original i/p.

★ **Sorting**

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

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

$A = [82, 44, 39, 36, 23, 91, 59]$, $n = 7$

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

Sort L, and R and merge.

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

Sort LL and LR and merge.

$LLL = [82]$ $LLR = [44]$

sort LLL and LLR and merge.

→ $[44, 82]$ (sorted LL)

	Mergesort(A):	
1.	n = length(A)	$D(n)$
2.	if $n \leq 1$:	$D(n)$
2.5.	return A.	$D(n)$
3.	$mid = \lceil \frac{n}{2} \rceil$	$D(n)$
4.	$L = A[0 \ldots (mid-1)]$.	$O(n)$
5.	$R = A[mid \ldots (n-1)]$	$O(n)$
6.	S_L = Mergesort(L)	$T(\lceil n/2 \rceil)$
7.	S_R = Mergesort(R)	$T(\lfloor n/2 \rfloor)$
8.	M = Merge(S_L, S_R)	$O(n)$
9.	return M.	$O(n)$

$$T(n) = \begin{cases} O(n) & \text{if } n \leq 1 \\ 2T(n/2) + O(n) & \text{o.w.} \end{cases}$$

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

★ The Merge Procedure

Input: Two sorted arrays L, R

Output: An array M that is sorted and contains each element of L & R exactly once. (A "merge" of L & R).

```
Merge (L,R):
1.  l = length(L)   r = length(R)
2.  m = l + r.
3.  M = An array of length m, filled with NIL value
4.  i = 0, j = 0.
5.  for k = 0 to m-1:
6.      if j >= r :   || Have we run out of R?
7.          M[k] = L[i] ; i = i+1
8.      else if i >= l :
9.          M[k] = R[j] ; j = j+1
10.     else if L[i] < R[j]:
11.         M[k] = L[i] ; i = i+1
12.     else :
13.         M[k] = R[j] ; j = j+1
14. return M.
```

* Given two n-bits numbers in an array x & y multiply these two numbers and store their product in another array.

* **Integer Multiplication**

Input: Two n-bit numbers x, y given as arrays $x[0, \ldots n-1]$, $y[0, \ldots n-1]$ where $(n-1)$ bit is the least significant.

Output: The product $z = xy$ on an array Z of the same format.

$$(3132) \times (1234) = (31 \times 10^2 + 32) \times (12 \times 10^2 + 34)$$
$$= (31 \times 12 \times 10^4 + 32 \times 12 \times 10^2 + 31 \times 34 \times 10^2 + 32 \times 34)$$

~~Let y be two n-bit~~

Let x, y be two n-bit integers. Let x_L be the binary number consisting of the first $\lceil \frac{n}{2} \rceil$ bits of x, and x_R the binary number consisting of the remaining bits of x. Let y_L, y_R be similarly defined. Then

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

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

		# steps()
	Multiply (X, Y):	
1.	n = length (X)	O(n)
2.	if n == 1 :	O(1)
3.	if X[0] == 0 : return X	O(n)
4.	else : return Y	O(n)
5.	mid = $\lceil \frac{n}{2} \rceil$ // ceil (n/2)	O(n)
6.	X_L = X [0 .... (mid-1)]	O(n)
7.	X_R = X [mid .... n-1]	O(n)
8.	Y_L = Y [0 ..... (mid-1)]	O(n)
9.	Y_R = Y [mid ... (n-1)]	O(n)
10.	P_1 = multiply (X_L, Y_L)	T(n/2)
11.	P_2 = Multiply (X_L, Y_R)	T(n/2)
12.	P_3 = Multiply (X_R, Y_L)	T(n/2)
13.	P_4 = Multiply (X_R, Y_R)	T(n/2)
14.	S = ADD (P_2, P_3)	O(n)
15.	$F_1 = 2^{2\lfloor n/2 \rfloor} \cdot P_1$	O(n)
16.	$F_2 = 2^{\lfloor n/2 \rfloor} \cdot S$	O(n)
17.	F_3 = ADD (F_1, F_2)	O(n)
18.	P = ADD (F_3, P_4)	O(n)
19.	return P.	O(n).

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

$$\Downarrow$$

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

To compute the four products it is enough to compute $x_L \cdot y_L$, $x_R \cdot y_R$ and $(x_L + x_R) \cdot (y_L + y_R)$ and do some additions / subtractions.

Now

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

$$\Downarrow$$

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

$$\cong O(n^{1.6})$$

* **Matrix Multiplication**

Input: Two $n \times n$ integers matrices A, B.

Output: Then product $C = A \times B$

$$C[i,j] = \sum_{k=0}^{n-1} A[i,k] \times B[k,j] \rightarrow O(n^3)$$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

$n/2 \times n/2$.

$\forall A_{ij}$ are $\frac{n}{2} \times \frac{n}{2}$ matrices.

Similarly

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

__/__/__

$$C = \begin{bmatrix} \overbrace{A_{11}B_{11} + A_{12}B_{21}}^{C_{11}} & \overbrace{A_{11}B_{12} + A_{12}B_{22}}^{C_{12}} \\ \underbrace{A_{21}B_{11} + A_{22}B_{21}}_{C_{21}} & \underbrace{A_{21}B_{12} + A_{22}B_{22}}_{C_{22}} \end{bmatrix}$$

$$T(n) = 8\,T(n/2) + O(n)$$

$$\Rightarrow O(n^{\log_2 8}) \Rightarrow O(n^3).$$

★ **Strassen's matrix multiplication method** (1969)

$$P_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$

~~$P_2 = (A_{21} \times$~~

~~$P_2 = (A_{11}$~~

$$P_2 = (A_{21} + A_{22}) \times B_{11}$$

$$P_3 = A_{11} \times (B_{12} - B_{22})$$

$$P_4 = A_{22} \times (B_{21} - B_{11})$$

$$P_5 = (A_{11} + A_{12}) \times B_{22}$$

$$P_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$$

$$P_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

$$C_{11} = P_1 + P_4 - P_3 + P_7$$

$$C_{12} = P_3 + P_5$$

$$C_{21} = P_2 + P_4$$

$$C_{22} = P_1 - P_2 + P_3 + P_6$$

$$T(n) = 7\,T(n/2) + O(n)$$

$$\Downarrow$$

$$T(n) = O(n^{\log_2 7}) = O(n^{2.81}).$$

★ $\Omega(n \log n)$ worst-case time

Any comparison based sorting algorithm. ⇒ (Complexity theory)

$\log(n!)$??

$\approx \Theta(n \log n)$

$n^{n/2} < n! < n^n$

★ **Counting sort**

Input : Array $A[0, \ldots (n-1)]$ of integers where each integer $A[i]$ satisfies $0 \le A[i] \le k$ for some integer k (Intuitively $k << n$)

Output : A sorted version of A.

Rough idea : 1. For each $A[i]$, find the number c_i of other elements in A that are less or equal to $A[i]$.

2. Put $A[i]$ at $B[c_i]$

Counting Sort (A, k) :

1. n = length (A)
2. B = An (empty) array of length n.
3. C = An (empty) array of length k+1
4. for i = 0 to k :
5. C[i] = 0.
6. for i = 0 to (n-1) :
7. C [A[i]] = C [[A[i]] + 1.

```
8.  for  i = 1 to k:
9.      C[i] = C[i] + C[i-1]      // Cumulative count.
10. for i = n-1  down to 0:
11.     B[C[A[i]] - 1] = A[i]
12.         C[A[i]] = C[A[i]] - 1
13. return B.
```

Meaning of C[A[n-1]] just before the first iteration of the loop lines 10-12.

= the total numbers of elements that are ≤ A[n-1], and have not yet been placed at their correct position in B.

So place A[n-1] at B[C[A[n-1]] - 1]

★ ~~Stable sort~~

★ Stable sort : preserves order of the elements even if the ~~xx~~ numbers are numerically same. So sorting preserves the order in which the number arrives.

★ Radix Sort

The keys are written down in some "base".
Each key has width d
·Pad with blanks if needed.

Input: Array A[0, 1, ... (n-1)]
where each element A[i] has 'd' digits numbered 1 (leftmost digit) to d (rightmost digit). Digit d is the least significant digit.
Output: A sorted version of array A.

RadixSort (A, d):

1. for i = d down to 1:
2. Sort array A on the digit i using CountingSort
3. Let B be the array returned by Step 2.
4. A = B.
5. return A.

★ Stable algorithm

Suppose digit d can take k values.

$$O(d(n+k))$$

For Aadhar

d = 12
n = n
k = 10

$O(12(n+10)) \doteq O(n)$.

* **Dynamic Programming**

① Recursive algorithm

* $f_0 = 1,\ f_1 = 1 \quad f_{i+1} = f_i + f_{i-1}$ for $i \geq 1$.

1, 1, 2, 3, 5, 8, 13, 21, ...

Problem: Given $n \in \mathbb{N}$ as input compute f_n.

```
Simplefib(n):
1.    if n ≤ 1 ;
2.       return 1.
3.    Fn = Simplefib (n-1) + Simplefib (n-2)
4.    return Fn.
```

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

Assume $T(n) = \alpha^n$ for some const. α.

$$\alpha^n = \alpha^{n-1} + \alpha^{n-2}$$

$$\alpha^2 = \alpha + 1$$

$$\alpha^2 - \alpha - 1 = 0$$

~~$\alpha^2 - 2\alpha - \alpha - 1 = 0$.~~

$$\alpha = \frac{1 \pm \sqrt{1+4}}{2}.$$

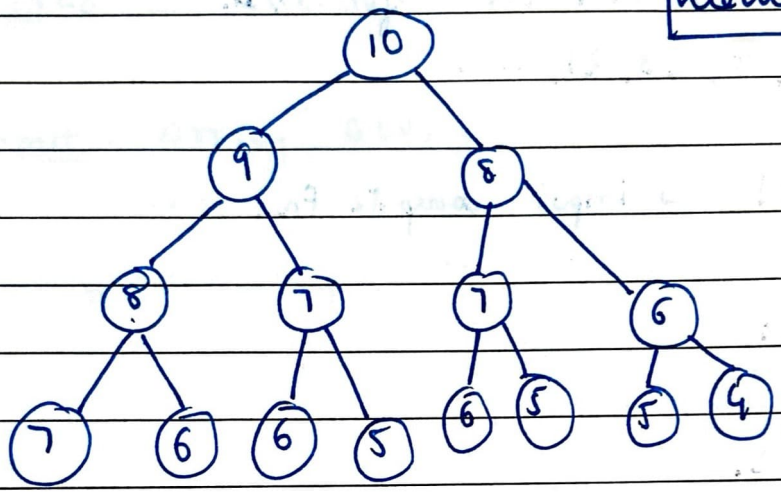
$$\alpha = \frac{1 \pm \sqrt{5}}{2} \approx (1.618)^n.$$

$$\alpha = \frac{1 - \sqrt{5}}{2}$$

DP

$n = 35$: 2.2s	$n = 35$ 8.6×10^{-5} s
$n = 40$: 24s	$n = 40$ 9.1×10^{-5} s
$n = 45$: 267s	$n = 45$ 9.9×10^{-5} s
$n = 50$. : 2900s.	$n = 50$ 10^{-4}.

memoization



Simple Fib(n):

1. if $n \leq 1$: $\Omega(1.618)^n)$
2. return 1
3. F_1 = Simplefib(n-1)
4. F2 = Simplefib(n-2)
5. return F1 + F2

SmartFib(n):

1. if $n \leq 1$: return 1
2. F = an array of length (n+1).
3. F[0] = F[1] = 1.
4. for i = 2 to n:
5. F[i] = 0.
6. F = FibHelper (n, F)
7. return Fn.

else:

Top-down

```
FibHelper(i, F):
1.  if F[i] ≠ 0 :
2.      return F[i]
3.  F1 = FibHelper(i-1, F)
4.  F2 = FibHelper(i-2, F)
5.  F[i] = F1 + F2
6.  return F[i].
```

$T(i) \geq T(i-1) + T(i-2)$

$T(i-2)$ → c

$T(i-1)$ already computes $T(i-2)$.

2) $T(n) \leq O(n)$

$T(n) \leq T(i-1) + c$

$\leq T(n-2) + T(n-1) + 2c$

$\leq T(1) + n \cdot c.$

SmartFib(5)

1. —

2. F = | 1 | 1 | 0 | 0 | 0 | 0 | (indices 0 1 2 3 4 5)

3. 

4.

5. Fn = FibHelper(5, F)

Let **S** be the no. of times that FibHelper(,) is called recursively. Then the total no. of calls to FibHelper(,) is : (S+1).

Total time that line 6 of Smartfib takes = O(S+1)

At line 6 of SmartFib( ), (n-1) element of F are zeroes.

Each time that lines 3,4 of FibHelper() are executed, one more element of f becomes non-zero.

The code never changes a non-zero in F to zero
So lines 3, 4 of FibHelper get executed = (n-1) times

$\Rightarrow S = 2.(n-1)$

Bottom-up.

```
SimplerFibDP(n):
1. if n≤1 : return 1
2. F = an array of length n+1
3. F[0] = F[1] = 1.
4. for i = 2 to n:
5.     F[i] = F[i-1] + F[i-2]
6. return F[n]
```

→ ^nC_k

$^nC_k = {}^{n-1}C_{k-1} + {}^nC_{k-1}$

$^0C_0 = 1$ $\qquad$ $^nC_n = 1.$

```
Choose(n,k):                          |
1. if k==0 or k==n:                   | 1
2.     return 1                       | 1
3. C1 = Choose(n-1, k-1)              | T(n-1)
4. C2 = Choose(n-1, k)                | T(n-1)
5. return C1+C2.                      | 1.
```

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

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

```
SmartChoose (n,k):
1. if k==0 or k==n:
2.     return 1
3. C = an (n+1) x (n+1) array
4. Set all values in C to NIL
5. for i= 0 to n:
6.     C[i][0] = 1
7.     if i ≤ k : C[i][i] = 1.
8. nCk = ChooseHelper (n, k, C)
9. return nCk

ChooseHelper (i,j, C):
1. if C[i][j] ≠ NIL:
2.     return C[i][j]
3. C[i][j] = ChooseHelper (i-1, j-1, C) + ChooseHelper (i-1, j, C)
4. return C[i][j].
```

$\therefore O(nk)$.

* **The Rod Cutting Problem**
(from the CLRS book).

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

Input: Input is an array $P[0, \ldots n]$ of prices $P[0] = 0$, $P[i] = p_i$ for $i > 0$.

For $0 \le i \le n$, r_i is the max^m revenue that can be obtained by (possibly) cutting up and selling a rod of length i. Then $r_0 = 0$, and we want to find r_n.

r_1	1	No cutting
r_2	5	No cutting
r_3	8	No cutting
r_4	10	cut into 2 pieces of length 2.
r_5	13	2+3.
r_6	17	No cutting.
r_7	18	6+1
r_8	22	6+2
r_9	25	6+3
r_{10}	30	No cutting.

__/__/____

→ inducting on

```
CutRod (n, P):
1.    if (n == 0) :                                   1
2.        return 0                                    1
3.    maxRevenue = -1.                                1
4.    for i from 1 to n:                              n
5.        iFirstRevenue = P[i]                        T(n-i)
6.                  + CutRod (n-i, P).
7.        maxRevenue = max (maxRevenue, iFirstRevenue)   1
8.    return maxRevenue.                              1
```

Let $T(n)$ be the number of times that cutRod() is called. Start with the call CutRod(n, p)

$$T(n) = \sum_{i=1}^{n} T(n-i) + 5 + n.$$

$n - i = j$

$$= 1 + \sum_{j=0}^{n-1} T(j) + \quad \text{if } n > 0.$$

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

P_i: Price of a piece of length i

r_i: Maximum revenue from a piece of length i.

We want r_n.

Property: The length of "some" piece in the solution for $1 \le i \le n$, let S_i be the set of all solutions which contain a piece of length i. The S_n = {the entire rod}

$$S = S_1 \cup S_2 \cup \cdots S_n$$

In general, $S_1 \cap S_2 \neq \phi$

Let maxRevenue (S_i) be the maximum revenue from any element of S_i

Then,

$$r_n = \max_{i=1}^{n} (\text{maxRevenue } (S_i))$$

$$\text{maxRevenue } (S_i) = P_i + r_{n-i}$$

$P_i = P[i]$ selling price of a length of pieces i

r_j = maxm revenue realizable from a rod of length j.

we want r_n.

Let S be the set of all feasible solutions for $1 \le i \le n$, S_i is all solutions which have atleast one piece of length i.

$$S = S_1 \cup S_2 \cdots \cup S_n$$

Let maxRevenue (S_i) be the most money that can be made by selling any one of element of S_i.

$$r_n = \max_{i=1}^{n} (\text{maxRevenue } (S_i))$$

$$\text{maxRevenue}(S_i) = P_i + r_{n-i}$$

If we compute the r_j values in increasing order of j, we will always know what to do.

To compute for each r_j, use projection onto the length of some piece.

```
CutRodDP (n, P)
1. R = an array of length n+1
       // R[j] will eventually be rj.
2. R[0] = 0.
3. for j = 1 to n:
4.       jMaxRevenue = -1     // will become rj
5.       for i = 1 to j:
6.           iRevenue = P[i] + R[j-i].
7.           jMaxRevenue = max(jMaxRevenue, iRevenue)
8.       R[j] = jMaxRevenue
9. return R[n].
```

★ **LPS**

S[i] == S[j]

① LPS(i, j) = 2 + LPS(i+1, j-1)

② If i == j
1

③ if S[i] ≠ S[j]
max(LPS(i+1, j), LPS(i, j-1))

① E

```
def MinCoins (D, n, memo = None):
    if memo==None:
        memo = [-1] * (n+1)
    if n==0:
        return 0
    if memo[n] != -1
        return memo[n]
    minval = float('inf')
    for d in D:
        if n-d>=0
            minval = min(minval, 1+MinCoins(D, n-d, memo))
    memo[n] = minval
    return memo[n]
```

def

* **Memoization**

① Minimum cost to cli

* **Cutting Rod**

You are given rod of length n and a list of prices for different rod lengths. Find the maxm value you can obtain by cutting the rod and selling the pieces.

n = 4

prices [1, 5, 8, 9]

for i from 0 to n-1

cutRod[

t = max(cutRod[i] p[i], cutRod[n-i]

"abcbda"

abcbda

adbcba

if A[i] == A[j]

LPS[i][j] = 1 + LPS[i-1][j-1]

else

```
CutRod (n, P)
1. if n==0 : return 0
2. maxRevenue = -1
3. for i = 1 to n:
4.     iFirstRevenue = P[i] + CutRod ((n-i), P)
5.     maxRevenue = max (iFirstRevenue, maxRevenue)
6. return maxRevenue
```

S: Universe of all feasible solutions

For $1 \le i \le n$ S_i = set of all solutions that contain a piece of length i.

$$S = \bigcup_{i=1}^{n} S_i \qquad i \neq j \Rightarrow S_i \cap S_j = \phi$$

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

r_j: max revenue obtainable from a rod of length j.

We want r_n.

Observe $OPT(S_i) = P_i + OPT(S_{n-i})$.

$= P_i + r_{n-i}$

```
CutRod_DP(n, P):
1.  R = An array of length n+1.
        // R[j] will eventually be rj
2.  R[0] = 0.
3.  for j = 1 to n:  // compute rj
4.      jMaxRevenue = -1   // will become rj
5       for i = 1 to j:   // project onto length of some piece
6.          iRevenue = P[i] + R[j-i]
7.          jMaxRevenue = max(iRevenue, jMaxRevenue)
8.  R[j] = jMaxRevenue
9.  return R[n].
```

* **Matrix multiplication**

$A_{p \times q}$, $B_{q \times r}$, compute AB

$\langle A_1, A_2, A_3 \ldots A_n \rangle$

A_{i-1}, A_i are conformable for multiplication for $2 \le i \le n$

eg. A_1 2×3 A_2 3×4 A_3 4×5

$(A_1 A_2)(A_3) = (2 \times 3 \times 4) + (2 \times 4 \times 5) = 64$

$A_1 (A_2 A_3) = 2 \times (3 \times 4 \times 5) + (2 \times 3 \times 5)$

$= 90.$

* Least cost Matrix Chain Multiplication

Input : A sequence $A_1, \ldots A_n$ of matrices where A is $d_{(i-1)} \times d_i$ matrix for each $1 \le i \le n$

Output: The least "pqr" cost of finding the product $A_1, A_2 \cdots A_n$.

$d_0 \times d_b \quad\quad d_b \times d_n$

$(A_1 \cdots A_b)(A_{b+1} \cdots A_n)$

$c_1 \quad\quad c_2$

$c_1 + c_2 + d_0 \cdot d_b \cdot d_n$.

Input: An array $D[0, \ldots n]$ of positive integers, where the original array A_i has dimensions $D[i-1] \times D[i]$

Output: The least "pqr" cost of finding the product $A_1, A_2 \cdots A_n$.

f: Universe of all solutions

For $1 \le b \le (n-1)$, let S_b denote all the solutions that "break" after array A_b.

$$f = \bigcup_{b=1}^{(n-1)} S_b \quad \Big| \quad \text{if } i \neq j \text{ then } S_i \cap S_j = \phi$$

$OPT(S_b)$ = the least cost of finding $A_1, A_2 \cdots A_n$ where the first break is after A_b.

★ **Least Cost Matrix chain multiplication**

Input : An array $D[0 \cdots n]$ of positive integers

Goal : Find the least cost (sum of the 'pqr' values) for computing the product $A_1 \cdot A_2 \cdots A_n$ where A_i is a $d_{i-1} \times d_i$ matrices of each $1 \le i \le n$.

For $1 \le s \le t \le n$, let $OPT(s,t)$ be the least cost for finding the product. $A_s A_{s+1} \cdots A_t$

we want $OPT(1,n)$

$OPT(s,s) = 0$.

f = Universe of all feasible solutions.

For $1 \le b \le n-1$,

S_b = set of all solutions where "break-point" is b.

Then, $f = \bigcup_{b=1}^{n-1} S_b$.

$(i \ne j \Rightarrow S_i \cap S_j = \phi)$

For $1 \le b \le n-1$ let $\widehat{OPT}(S_b)$ be the optimum value over all of S_b.

$\widehat{OPT}(S_b) = OPT(1,b) + OPT(b+1,n) + d_0 d_b d_n$.

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

A measure that strictly dropped from OPT(1, n) to each of OPT(1, b) & OPT(b+1, n)

The # of arrays involved in the terms OPT(i, j)
→ n for OPT(1, n)
→ b≤n for OPT(1, b)
→ n-b < n for OPT(b+1, n).

Let (j - i+1) be the "length" of OPT(i, j)

```
LeastCostMCM(n, D)
1.  OPT = OPT[0....n][0....n]
      // OPT[i][j] will eventually contain OPT(i, j).
2.  for i from 1 to n:
        OPT[i][i] = 0.
    // OPT[0] is a dummy row so (that head doesn't explode)
4.  for l = 2 to n:   // l = j-i+1
5.      for i = 1 to n-l+1:
6.          j = i + l - 1
7.          OPT[i][j] = ∞
8.          for b = i to (j-1):
9.              ~~val = OPT[i][b] + OPT[b][j]~~
9.              val = OPT[i][b] + OPT[b+1][j] + D[i-1].D[b].D[j]
10.             OPT[i][j] = min(OPT[i][j], val).
```

Knapsack (0/1) problem

Bag capacity $W = 55$ kg.

Item	I_1	I_2	I_3
Weight (kg)	10	20	30
Value (₹)	60	100	120.

Best solution to

0/1 Knapsack with repetition $5I_1$:) Rs. 300. (wt.=50)

0/1 knapsack without repetition $I_2 + I_3$ =) Rs. 220. (wt.=50)

Capacity $W = 11$ kg.

Item (Itype)	I_1	I_2	I_3	I_4
Weight (kg	6	3	4	2
Value (₹)	30	14	16	9

- with repetition - $I_1 + I_2 + I_4 = 53$ kg Rs. (11 kg)
- without repetition - $I_1 + I_2 + I_4 = 53$. Rs. (11 kg).

CutRod $W = n$

0/1 knapsack with repetition.

(weakly NP-complete) __/__/____

↙ W can be large compared to n.

0/1 knapsack with repetition

Input: (1) Capacity W of the knapsack, $W \in \mathbb{N}^+$

(2) Item types $I_1, I_2, \ldots, I_n$

(3) weights $w_1, w_2 \ldots w_n$ each a positive integer.

(4) Values $v_1, v_2 \ldots v_n$ positive integers.

Goal: Find the maximum sum of values of a collection of items whose total weight is at most W, where each type of item may occur any number of times.

S = universe of all feasible solutions.

For $1 \le j \le n$

S_j = all elements of S which contain atleast one item of type I_j

~~For eg~~

For any $X \subseteq S$, let OPT(X) be the maximum value over all elements of X. We want OPT(S)

$$OPT(S) = \max_{j=1}^{n} (OPT(S_j))$$

$OPT(S_j)$ = v_j + value of an optimum collection that fills in a knapsack of capacity $W - w_j$

Array maxVal[0 .... W]

maxVal[c] = max value of a knapsack with capacity c.

maxVal[0] = 0.

* **0/1 knapsack without repetition**

Input: (1) Capacity W of the knapsack, $W \in \mathbb{N}^+$
(2) Itemtypes $I_1, I_2, \ldots I_j, I_n$
(3) Weights $w_1, w_2, \ldots w_n$ each a positive integer.
(4) values $v_1, v_2 \ldots v_n$ positive integers.

Goal: Find the maximum sum of values of a collection of items whose total weight is at most W, where each type of item may occur at most once.

For each $1 \le j \le n$, what is the most value we can get if we only pick items from among
$\langle I_1, I_2, \ldots I_j \rangle$

For $1 \le j \le n$, S_j = all feasible solutions which pick items only from
$\langle I_1, I_2, \ldots I_j \rangle$.

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

~~$OPT(S_n) = \max(v_n +$ or~~

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

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

if $w_n < W$.

Array maxVal $[0 \ldots n]\ [0 \ldots w]$

~~maxval [i][j] is the maximum~~

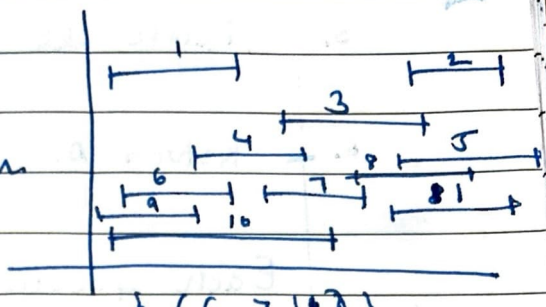
maxval [j] [k] is the maximum value of a knapsack of capacity k, that has items only from $\langle I_1, \ldots I_j \rangle$.

maxVal [0][K] = 0 $\forall$ k

maxval [j][0] = 0 $\forall$ j.

★ **Greedy Algorithms**

Kleinberg & Tardos, Algorithm Design.



★ **Interval Scheduling**

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

Goal Find a Largest collection of compatible requests.

① Pick a request with the earliest starting time. ✗
② Pick a request with the smallest duration $(f(i) - s(i))$ ✗
③ Pick a request that ends first (least $f(i)$). ✓
④ -- latest starting time. ✗
⑤ Pick a request with least no. of conflicts ✗

~~Let R be the set of remaining requests, and A the set of~~

Let L be the set of left-over requests and A be the set of accepted requests

1. Set $L = R$, $A = \phi$
2. while L is not empty
3. Find a request i with least value for $f(i)$.
4. $A \leftarrow A \cup \{s(i), f(i)\}$.

5. Delete all requests j from L where $s(j) < f(i)$.

6. Return A.

Each greedy step is "optimal" in some sense:

Let $A = \langle I_1, I_2, \ldots I_k \rangle$ be the order in which the requests we have added to A.

Let $O = \langle I_1, I_2, \ldots I_n \rangle$ be an unknown optimal solution, with requests sorted in increasing order of starting times

- Is O sorted in increasing order of finishing times?

Claim For each $p \leq k$, it is true that $f(I_p) \leq f(J_p)$ holds

- The p^{th} request in A ends no later than the p^{th} request in any optimal solution.

★ **Proof By Induction**

Base case $p = 1$.

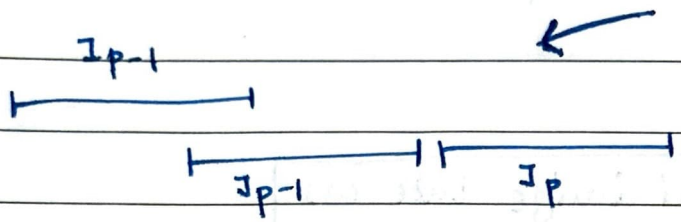
$$f(I_1) \leq f(J_1)$$

Inductive assumption : For some $p \geq 2$

$$f(I_{p-1}) \leq f(J_{p-1}).$$

Now, to show

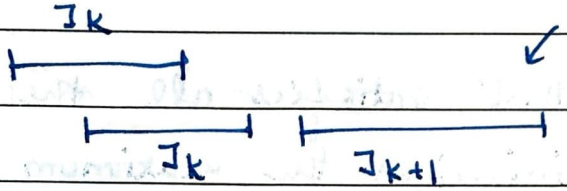
$f(I_p) \leq f(J_p)$.



We want $k \geq m$

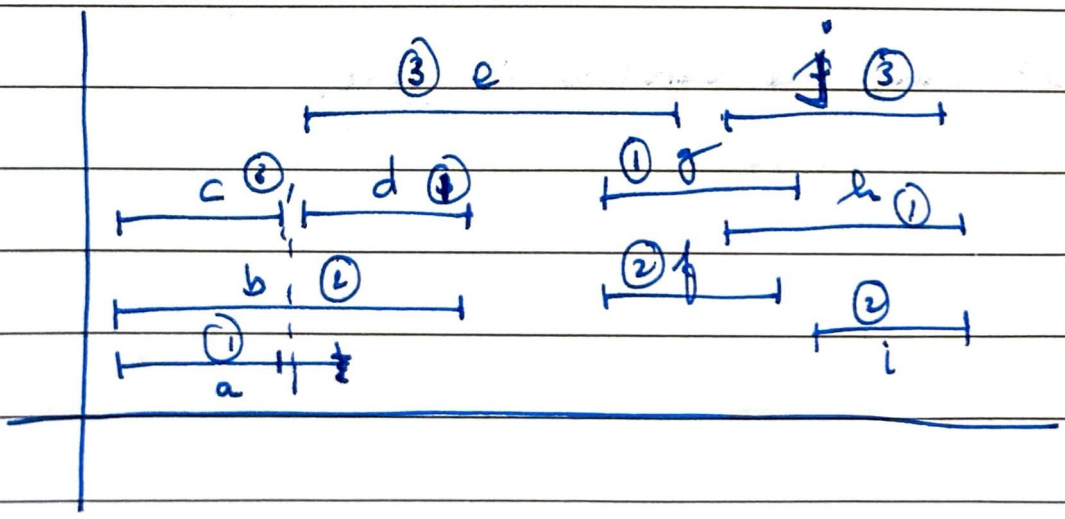
Suppose not, then $k < m$.

$f(I_k) \leq f(J_k)$.



Suppose we have another feasible solution after I_k which is ~~first~~ starting after I_k, but then why didn't we pick it, so by contradiction you will always pick a feasible solution Hence, $k \geq m$ proved by contradiction.

* **Scheduling All Intervals**
(Interval Partitioning / Colouring).



Depth d of the collection of intervals the max no. of intervals the max no. of intervals that share any are time point.

★ **Scheduling to minimize Lateness**

Input: n requests $\{r_1 = (t_1, d_1), r_2 = (t_2, d_2), \ldots r_n = (t_n, d_n)\}$
- t_i, d_i are the duration and deadline of request r_i, respectively

Goal: Find a schedule that satisfies all the requests (on one resource), and minimizes the maximum lateness.

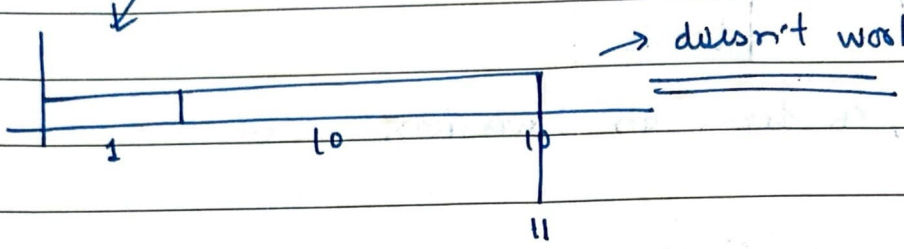
For a given schedule, let $s_1, s_2, \ldots s_n$ be the start times of these n requests
The end times are $f_i = s_i + t_i$

Request r_i is late if $f_i > d_i$
- The lateness of r_i is $f_i - d_i$, if x_i is late.

- If not late, the lateness is zero.

* **Shortest duration first**

{ (1, 100) , (10, 10) }



Claim I: The schedule A generated by the greedy algorithm has no idle time.

Claim II: There is an optimal solution with no idle time.

** An **inversion** in a schedule A is a pair of requests r_i, r_j where $d_j < d_i$, and r_j is scheduled before r_j.

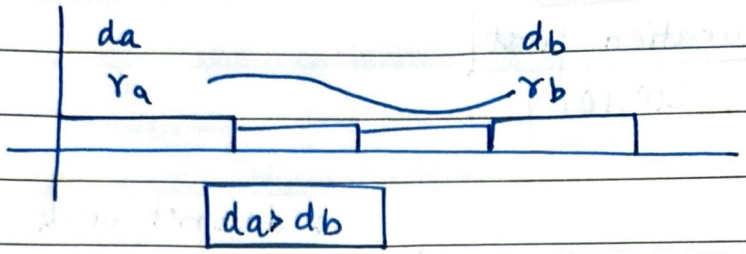
Claim III: Schedule A has no inversions.

Question: Can there be more than one schedule with no inversions and no idle time? ⇒ Yes, but only when 2 requests with the same deadline are flipped.

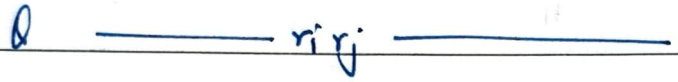
Claim IV: All schedules with no inversions and no idle time, have the same maximum lateness.

Let O be an optimal schedule with no idle time

Claim V: 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 scheduled immediately after r_i, in O.



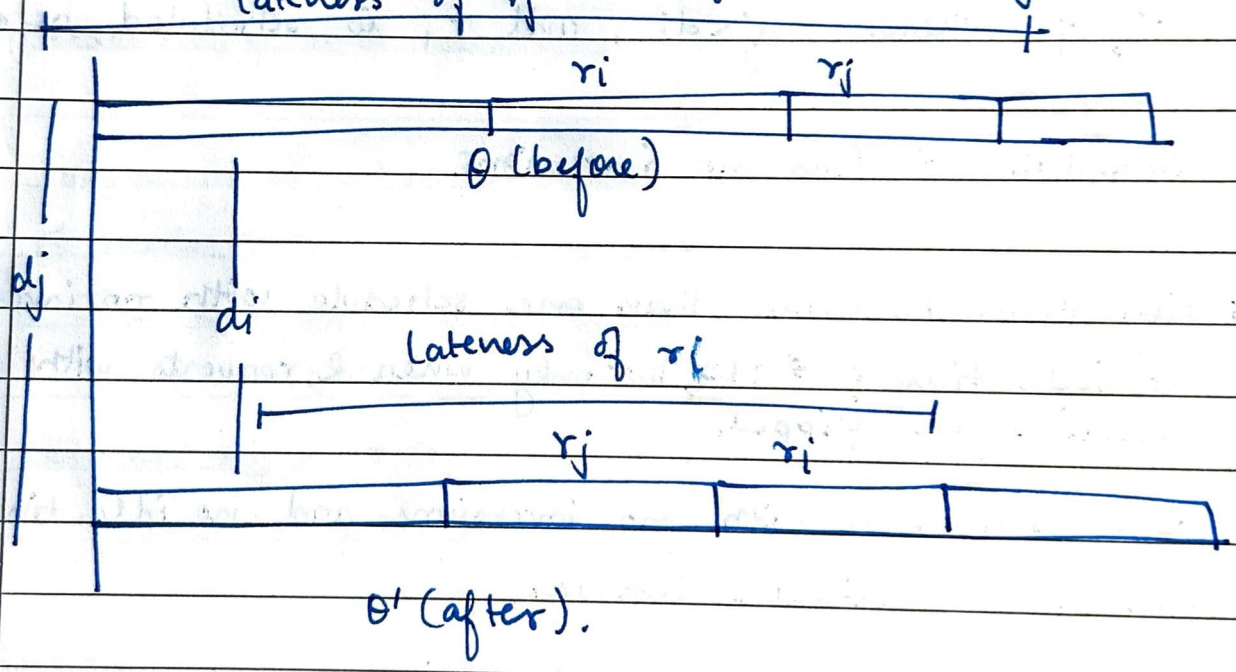
r_a, r_b form an inversion.



Claim VI: Let θ' be the schedule obtained from θ by flipping r_i and r_j. Then

① θ' has one fewer inversion

② The maximum lateness of θ' is not larger.



* **Amortized Analysis**

* **Dynamic Multiset**

- Collection of elements
  - May be atomic or (Key, value) pairs.
- Elements may repeat (Hence, multiset).
- Elements may be added/ removed / modified
  - Hence, dynamic.

S: dynamic multiset

k: key, x (pointer to) element.

Search (S, k)

Insert (S, x)

Delete (S, x).

Minimum (S) / Maximum (S)

IsEmpty (S) / IsFull (S)

Others...

Ex: **Stack** **LIFO**

Last In First Out

Insert Push(x)

Delete Pop

IsEmpty, IsFull

Implement a stack with $\leq n$ elements, using an array $S[0 \ldots n-1]$.

Bottom of the stack $S[0]$

Top of the stack $T = [-1, 0, \ldots (n-1)]$

↓ the index of the last element that was pushed on the stack.

Start with $T=-1$.

```
IsEmpty()
    return T == -1
Is Full()
    return T == n-1.
```

Exercise: Complete the implementation of a stack
- MakeStack(n). Create an empty stack of capacity n, return the top of the ~~to~~ stack.
- IsEmpty, Isfull, Push, Pop.

Multipop(k):
- Pop k elements from the stack.
- If $k \leq 0$ do nothing.
- If there are fewer than k elements in the stack pop all of them.

Start with an empty stack
Do n Push(), Pop()
& Multipop() ~~operations~~ operations.
↓
O(n)
n operations
$O(n^2) \Rightarrow$

t_1 pushes $\neq O(t_1)$
t_2 pops $\neq O(t_2)$
t_3 Multipop(k_i).

$$t_2 + \sum_{i=1}^{t_3} k_i \leq O(t_1)$$

~~t₂ ≠ t~~

$t_2 + t_3 \sum_{i=1}^{t_3} k_i \leq O(t_1)$??

I ♡ U (especially when you keep your hair open)
- Your Secret LVR
(Let me be your ChatGPT)

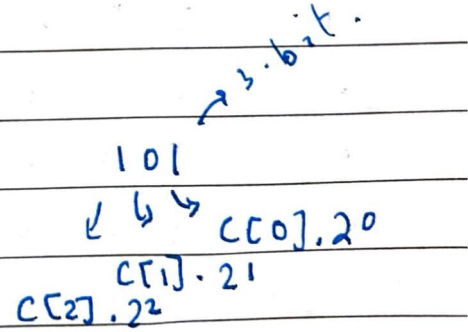
★ Say a total of t calls to PUSH()
The maximum no. of calls to POP(), on a non-empty stack = t.
The time taken by any specific call to MultiPop() is proportional to the no. of calls to Pop() on a non-empty stack. that is a result of this call to Multipop().

The total time taken by all calls to Multipop is proportional to the total no. of calls to Pop() made on non-empty stacks = O(t).

The total time taken by any sequence of n operations = O(n)

★ **k-bit binary counter, starting at 0**

C [0 . . . . (k-1)].
Convert order bit C[0]
Highest order bit C[k-1].
The count, $C = \sum_{i=0}^{k-1} C[i] \cdot 2^i$



e.g. $k=8$, $C = [1,1,1,1,0,0,1,0]$

$c = 79$

Increment by 1

~~$c = [0,0,0,0,1,0,1,0]$~~

$c = [0,0,0,0,1,0,1,0]$.

Increment (c, k) // $c = c[0 \ldots (k-1)]$.

1. $i = 0$
2. while $(i \leq (k-1))$ and $(C[i] == 1)$:
3. $c[i] = 0$.
4. $i = i+1$
5. if $i \leq (k-1)$:
6. $C[i] = 1$.

Starting with $c=0$ do n increment operations.

Worst case $O(nk)$ time.

c	c [0 . . . . 7]
0	0 0 0 0 0 0 0 0
1	1 0 0 0 0 0 0 0
2	0 1 0 0 0 0 0 0
3	1 1 0 0 0 0 0 0
4	0 0 1 0 0 0 0 0
5	1 0 1 0 0 0 0 0
6	0 1 1 0 0 0 0 0
7	1 1 1 0 0 0 0 0

(III.) Potential method → Physics

__/__/____

C[i] flips once every 2^i calls to Increment()

The no. of times that C[i] flips in n calls to Increment.

$\leq \left\lfloor \frac{n}{2^i} \right\rfloor$

The total cost of n calls to Increment() is proportional to T, where T is the ~~total~~ total no. of bit flips over these n calls.

$$T \leq \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor$$

$$\leq \sum_{i=0}^{(k-1)} \frac{n}{2^i} = n \sum_{i=0}^{(k-1)} \frac{1}{2^i} < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

Accounting method

Budget for Stack with Multipop().

	Actuals
Push → Rs. 2.	Rs. 1
Pop → Rs. 0.	Rs. 1
Multipop → Rs. 0.	Rs. (# things popped)

The total amortized cost of any sequence of operations must be atleast the actual total cost of these operations.

$$\left(\sum_{i=1}^{n} \hat{C}_i\right) \geq \left(\sum_{i=1}^{n} C_i\right)$$

C_i : actual cost of i^{th} operation

$\hat{C}_i$: amortized cost of i^{th} operation.

n-bits → 0 ① n=3

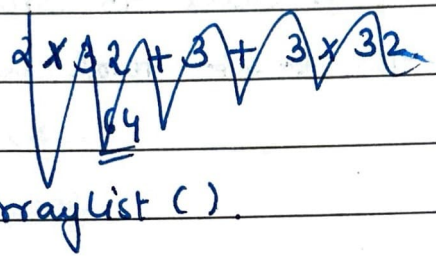
0 0 0 2+2+1 = 2n

__/__/____

For the bit counter:

To flip a bit from
0 to 1 : Rs. 2.

To flip a bit from
1 to 0 : Rs. 0.



A = Create ArrayList ().

Potential Method

Assign a number, called the "potential" to the entire data structure.

Each operation may change the potential

The amortized cost of an operation = the actual cost of the operation + the change in potential.

For any valid sequence of operations, the total amortized cost must cover the total actual cost.

D_0: the initial state of data structure.

$\Phi(D_0)$: the potential of the initial state.

We do a sequence of n operations.

for $i \in \{1, \ldots n\}$

- D_i is the state after operation i.
- $\Phi(D_i)$: potential of D_i
- c_i : the actual cost of operation i.

The amortized cost of operation i (changing D_{i-1} to D_i).

$$\hat{c}_i = c_i + (\phi(D_i) - \phi(D_{i-1}))$$

So

$$\sum_{i=1}^{n} \hat{c}_i = \left(\sum_{i=1}^{n} c_i\right) + (\phi(D_n) - \phi(D_0))$$

Choose $\phi()$ such that

* $\phi(D_0) = 0$
* $\phi(D_i) \geq 0$ for $\forall i$.

* **Stack with multipop**

$\phi(S_i)$ = height of the stack S_i

Initial stack S_0.

Let S_{i-1} be a stack with s elements $\phi(S_{i-1}) = s$.

- Amortized cost of Push when applied to S_{i-1}:
  - Change in potential = $\phi(S_i) - \phi(S_{i-1}) = 1$
  - Actual cost of Push() = 1
  - Amortized cost of Push() = 1+1 = 2.

Amortized cost of Pop().

- Change in potential after one call to Pop() = -1
- Actual cost of Pop(): 1

Amortized cost of Pop(): 0

Amortized cost of Multipop(k)

- Let $k' = \min(s, k)$
- Actual cost of Multipop(k) = k' change in potential = $-k'$.
- Amortized cost of Multipop = 0

__/__/____

* **Bit counter**

k- bit counter $\Phi(C_i)$ = number of 1's.

Initial state = C_0

Let C_i be the state of the counter just after the ith call to Increment().

Let b_i be the no. of 1's in C_i

Suppose the ith call to Increment() sets exactly t_i bits to zero.

The actual cost of this call is $\leq t_i + 1$.

If $b_i = 0$, then $b_{i-1} = t_i = k$.

If $b_i > 0$ then $b_i = b_{i-1} - t_i + 1$.

In any case $b_i \leq b_{(i-1)} - t_i + 1$.

$0 \leq t_i - t_{i+1}$.

The potential difference $\Phi(C_i) - \Phi(C_{i-1})$

$= (b_i - b_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = (1 - t_i)$.

The amortized cost of this call to Increment =

actual cost + change in potential.

$\leq t_i + 1 + 1 - t_i$

$= 2$.