Lec-12 Dynamic Programming (DP)

- The history of its name seems interesting!
- All about Table filling
- Memoization.
- Fibonacci Numbers : 1, 1, 2, 3, 5, 8, ...

$$F_0 = F_1 = 1$$

$$F_i = F_{i-1} + F_{i-2} \quad \forall \; i \geq 2$$

Input: Non-negative integers n

Output: The Fibonacci Number, F_n.

Algorithm :

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

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

$$T(n) = O(1.618^n)$$

n	$T(n)$
35	2.2s
40	24s
45	267s
50	2990s

← In his laptop.

→ Solve the issue using memoization.

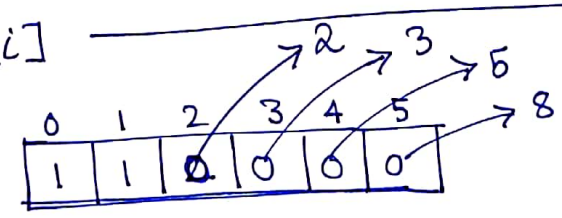
Memoization :

Steps

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

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

E.g., F =




```
SmartFib(5)
FibHelper(5)
  └ FibHelper(4)
      └ FibHelper(3)
          └ FibHelper(2)
              └ FibHelper(1) : rets (1)
              └ FibHelper(0) : rets (0)
          └ FibHelper(1) : rets (1)
      └ FibHelper(2) : rets (2)
  └ FibHelper(3) : rets (3)
```

updated running times:

n	SimpleFib	SmartFib
35	2·2 secs	8.6×10^{-5} secs
40	2·4 secs	9.1×10^{-5} secs
45	267 secs	9.9×10^{-5} secs
50	2990 secs	10^{-4} secs.

$T(n)$: time taken for FibHelper

$T(n) = T(n-1) + 5$

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

$S(n)$: SmartFib

$S(n) = T(n) + 2n + 7$

$\Rightarrow S(n) = O(n^2)$.

SimplerFibDP(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 Bin. Coeff ${}^nC_k$

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

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

Result: ${}^nC_k = {}^{n-1}C_{k-1} + {}^{n-1}C_k$

① 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)
```

Ignoring what k is doing, let $T(n)$ = atleast calls so that

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

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

(2) SmartChoose (n, k): Memoization

```
if k=0 or k=n: return 1
C = a global (n+1) x (k+1) array.
Set all values of 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])
```

Lec-13 Dynamic Programming

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 $0 \leq i \leq 10$, a integer, let r_i be the max revenue that can be obtained by (possibly) cutting up and selling a rod of length, i, $r_0 = 0$

Eg.)

i	r_i	Length of the rods sold (pieces)
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
8	22	6+2
9	25	6+3
10	30	10

$P[0, \ldots, n]$ is the list of prices. $P[0] = 0$ and $P[i] = P_i$ for $i \geq 1$

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

$T(n) =$ times $T(n)$ is called

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

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

p_i is the selling price of a 'piece' of length i
r_j denotes the maximum revenue that can be made from a piece of length j.
we want to find r_n.

Step 1: Choose an attribute of the solution and express the entire solution space S as a union of subsets, where each subset has a different value for this attribute

E.g., Say we choose "the length of any one piece is a solution" as the attribute of that solution for $1 \le i \le n$. Let S_i be the set of all solutions that have attribute i. Then,

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

Let maxRevenue(S_i) be the maximum revenue that can be made from an element of set S_i

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

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

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

$P[0] = 0$

$P[i] = p_i$

```
CutRodDP(n, P)
    R = an array of length (n+1)
    # R[i] will contain r_i
    R[0] = 0
    for j = 1 to n:
        jMaxRevenue = -1
        # Go over all possible lengths of 'first' piece
        for i = 1 to j:
            iRevenue = P[i] + R[j-i]
            if iRevenue > jMaxRevenue:
                jMaxRevenue = iRevenue
        R[j] = jMaxRevenue
    return R[n]
```

Lec 14 : Memoization

```
#SmartFib(n):
    if n ≤ 1 : return 1
    F = 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
```

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

How do we analyze the previous algorithm?

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

Then: FibHelper(n) runs in O(t) time.

FibHelper is called each time the (n-1) elements of f is changed to a non-zero element, so that we change a zero

$t \leq 2(n-1)+1$ ← once in SmartFib(n)

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

SmartChoose(3,2):

1	NIL	NIL
1	1	NIL
1	NIL	1
1	NIL	NIL

```
# CutRod(n, P)   # Recursive
    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.
```

```
# MemCutRod(n, P)   # memoization.
    if n=0 : return P
    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]
```

Q. Analyse

Lec- 15

Matrix Chain Multiplication

#We have A_{pxq}, B_{rxs}
#AB is defined iff $q = r$
#Naive way : $O(pqs)$
$(AB)C = A(BC) = ABC$

Input: A sequence $(A_1, A_2, \ldots, A_n)$ of matrices, the product $A_1 A_2 \cdots A_n$ is defined $\forall\ n > i \geq 1$ (order matters)

Output: Using the highschool matrix multiplication algorithm as a subroutine, compute $(A_1 A_2 \cdots A_n)$ using the least number of arithmetic operations (asymptotically)

$A_1 \cdot A_2 \cdot A_3 \cdot A_4$:

		i		i
1.)	$A_1(A_2(A_3 A_4))$	1	4) $(A_1(A_2 A_3)A_4)$	3.
2.)	$A_1((A_2 A_3) A_4)$	1	5) $((A_1 A_2) A_3)A_4$	...3
3.)	$(A_1 A_2)(A_3 A_4)$	2		

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

steps

1.) $(5 \times 50 \times 10) + (100 \times 5 \times 10) + (10 \times 100 \times 10)$
$= 17500$

2.) $(100 \times 5 \times 50) + (100 \times 50 \times 10) + (10 \times 100 \times 10)$
$= 85000$

Least is 3> 8000!

In order to solve this, we will recursion.
The parenthesization that corresponds to any optimum solution can be thought of as consisting of

1. An index $1 \leq i < n$
2. An optimum cost parenthesization of $\langle A_{i+1}, A_{i+2}, \ldots, A_j \rangle$

3.> An optimum cost parenthesization of $\langle A_{i+1}, A_{i+2} \ldots, A$

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

But why the optimum parenthezation gives optimum global ~~optimum~~ " ? Use exchange argument.

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

E.g., $D = [10, 100, 5, 50, 10]$

$S :=$ Universe of all parenthezations of size n.

For $1 \le b < n$, let S_b be the set of all parenthesizations with break-point at b, then,

$$S = S_1 \cup S_2 \cup S_3 \cdots \cup S_{n-1}$$

Let $\hat{OPT}(S_b)$ be the least cost among all the solutions in S_1. Let

$OPT(1,n) :=$ be the optimum cost from A[1] to A[n]

$$= \min_{b=1}^{(n-1)} \hat{OPT}(S_b) \text{ . Then}$$

$$\hat{OPT}(S_b) = OPT(1,b) + OPT(b+1, n) + (D[0] * D[b] * D[n])$$

Let $OPT[i][j]$ should be the least cost of multiplying $\langle A_i, A_{i+1}, \ldots, A_j \rangle$; $i \le j$

Least Cost MCM (n, D):

```
OPT = A 2D array of dimensions (n+1) x (n+1)
# OPT[0] is a dummy row to make indices match the explanation.
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 = i+l-1
        OPT[i][j] = ∞   # sentinel.
        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]
```

(Non recursive DP algorithm)

Lec - 16

0-1 Knapsack w/o repetition

Burglar robbing a store with a knapsack of capacity W kg. There are 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.

Greedy Strategies fail: e.g.,

W = 55 kg.

Item	I_1	I_2	I_3
Weight	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 repetitions.

Input: The capacity W of the 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

Values: $v_1, v_2, \ldots, v_n$, each a positive integer

Goal: Find the max 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 I_j
2. The number p of different pieces (not types) in a solution.

DP by projections onto item types:

$\mathcal{S}$: set of all feasible solutions. for $1 \le i \le n$;

$\mathcal{S}_i$: set of all solutions that contain atleast one item of type I_i. In general, $S_i \cap S_j \neq \phi$ for $i \neq j$

For $X \subseteq \mathcal{S}$, let $OPT(X)$ be the optimum value of a solution in set X. We want to find $OPT(\mathcal{S})$. Then,

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

$OPT(S_i) = v_i + (\text{The optimum solution for a knapsack of capacity } W - w_i)$

Create an array maxVal of size W+1 :

1. maxVal[W] = the optimum solution for a knapsack of capacity, W.

2. maxVal[0] = 0.

3. use the recurrence to fill maxVal[W] in increasing order of W from 1 to W.

4. return the maxval of the last value.

```
RepKnapSackDP(W, values, weights):
    n = ~~len(values)~~ W
    maxVal = an array of
             length n+1, initialized
             to NIL values
    maxVal[0] = 0
    for w in 1 to W:
        wmaxvalue = -1
        for i = 1 to j: if weight[i] ≤ W {
            ivalue = value[i] + maxVal[w - weight[i]]
            if ivalue > wmaxValue : wmaxValue = ivalue.
        maxVal[w] = wMaxValue }
    return maxVal[W].   # Order : O(nW) time
```

DP by projections on to no. of pieces in a solution

S: set of all feasible solutions for $p \in \mathbb{N}$, let S_p,

S_p: set of all solutions that contain exactly P pieces

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

For this case, max value of P, say $N = \left\lfloor \frac{W}{\min_{i=1}^{n} w_i} \right\rfloor$

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

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

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

The pair (capacity, no. of pieces) "dropped". The no of pieces dropped by 1. So we compute the OPT values in increasing order of the no. of pieces, and for each such number p, in increasing order of knapsack capacities.

Given W, values, weights.

Outline: 0. N = len(values)

1. Compute $N = \left\lfloor \frac{W}{\min(\text{weights})} \right\rfloor$

2. Create an array maxVal[0, ..., N][0, ..., W]

3. maxVal[i][j] = Max value that can be obtained with a knapsack of capacity j, containing exactly i pieces.

4. maxVal[0][j] = 0 for $0 \leq j \leq W$

5. maxVal[i][0] = 0 for $0 \leq i \leq N$

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

```
RepKnapSack DP (W, values, Weights):
  n = length(values) - 1 , minWt = min(weights)
  N = floor(W/minWt)
  maxVal = an (N+1) x (W+1) array initiated to 0.
  for p = 1 to N:
      for w = 1 to W:
          pwMaxVal = 0
          for i = 1 to n:
              iValue = values[i], iweight = weights[i]
              if iweight ≤ w:
                  new val = iValue + maxVal[p-1][w - iweight]
                  if new val > pwMaxval:
                      pwMaxval = new val
              maxVal[p][w] = pwMaxVal
  return the maximum among the values maxVal[i][W]
  for 1 ≤ i ≤ N.  # Order O(nW²) time.
```

Projecting onto prefixes of the input

Items: $I_1, I_2, \ldots, I_i, \ldots, I_n$

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

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

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

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

Q. Write the pseudocode.

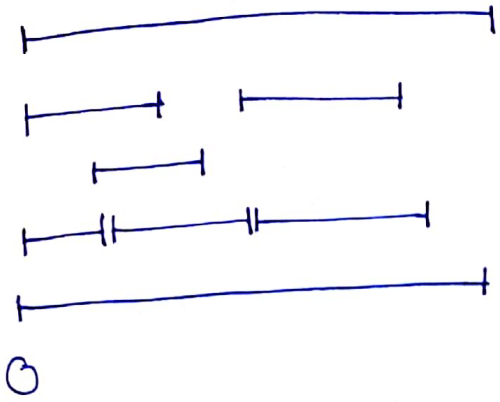
Lec-17 Interval Scheduling

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 all natural numbers and $s(i) < f(i)$ holds for all i

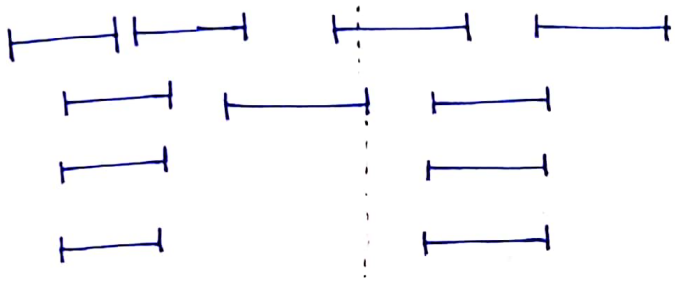
- $s(i), f(i)$ are the starting and finishing times for request
- Request i can be satisfied by alloting the resource from $s(i)$ to $f(i)$
- A subset of requests is compatible if they all can be scheduled together

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

Eg:



#



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.

Soln: Let $A = \{i_1, i_2, \ldots, i_K\}$ in the order in which the algorithm added the requests to A.

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

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

Proof by induction on p.

Base case : $p=1$: True by 'inspection'

Inductive step: To show $f(i_p) \leq f(j_p)$ from the inductive assumption we set $f(i_{p-1}) \leq f(j_{p-1})$

Proof : Discussed in class.

Lec-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 ~~paid~~ pick at least one element of X_a.

Suppose Y is an optimal collection and $Y \cap X = \phi$

Is it true that any solution can pick atmost one interval from X_a? Yes!

Proof: Discussed in the class.

Scheduling to Minimize Lateness.

Input: Non-negative integers, S, a set of n requests $R = \{ r_1 = (t_1, d_1), r_2 = (t_2, d_2), \ldots, r_n = (t_n, d_n) \}$. The t_i, d_i are natural numbers and $t_i \leq d_i$ holds true for all i.

— There is a single resource, available for use starting at time, S

— (t_i, d_i) are the duration and deadline for request r_i

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

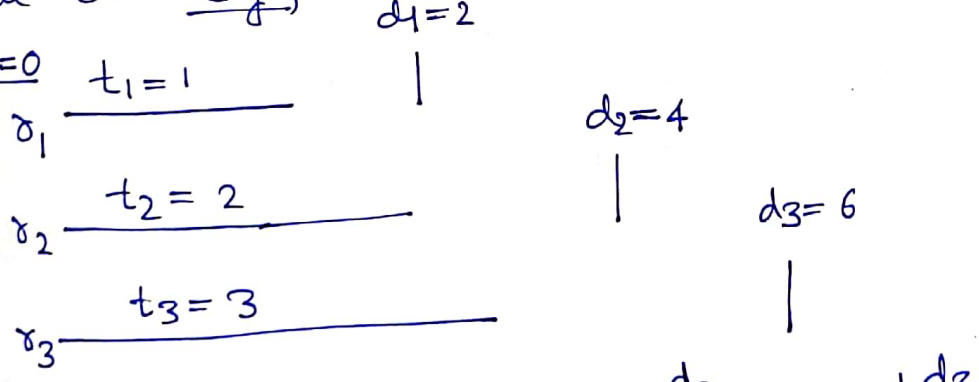
At most one request can be assigned to the resource at any point in time. Each requests once started has to end before another request can start using the resource.

Let $\{S_1, S_2, \ldots, S_n\}$ be the starting times for an assignment

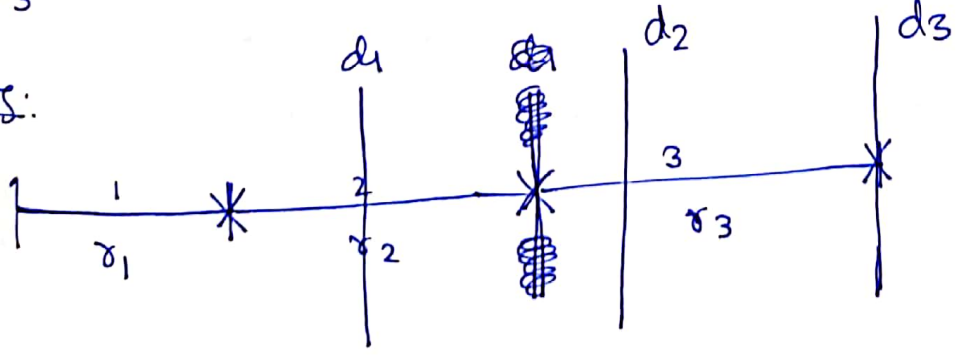
Then the finishing times are $f_1 = s_1 + t_1$, $f_2 = s_2 + t_2$, ...

$f_n = s_n + t_n$.

Request r_i is late if $f_i > d_i$. The lateness of such an r_i is $f_i - d_i$. The lateness of non-late requests is defined to be 0. E.g.:



Ans:



① Picking the shortest request first doesn't work.

e.g.; $R = \{(1,100), (10,100)\}$

② Picking the request with the least slack time $(d_i - t_i)$ doesn't work

e.g.; $R = \{(1,2), (10,100)\}$

Assume w.l.o.g. that $\gamma_1, \gamma_2, \ldots, \gamma_n$ is in non-decreasing order of deadlines that is $d_1 \leq d_2 \leq \cdots \leq d_n$.

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.

$s_1 = s$

$f_1 = s_1 + t_1$

$s_2 = f_1$

$f_2 = s_2 + t_2$

Claim: There is an optimum assignment with no idle time. Because if there were any gaps, we can shift all to the left and they wouldn't be worse than they were before.

An inversion in a schedule A' is a pair of requests γ_i, γ_j where $d_j < d_i$ and γ_i is scheduled before γ_j.

Claim: Schedule A computed by the greedy algorithm has no inversion. by construction.

Claim: All schedules with no gaps and no inversions, have the same maximum lateness.

Note: interchange among requests is possible iff they have the same deadline.

O: An optimum schedule with no gaps.

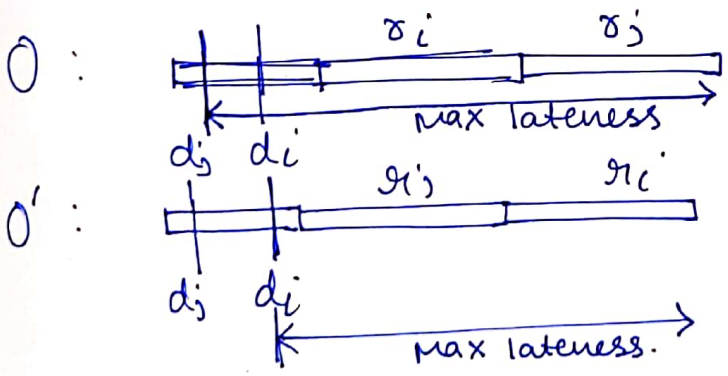
Claim: If O has an inversion, then there is a pair of requests r_i, r_j s.t. $d_j < d_i$ and r_j is immediately after r_i in O

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 where the deadline drops.

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

(i) O' has one fewer inversion than O.

(ii) The maximum lateness of O' is the same as that of O.



The max lateness is reduced.