Introduction

- Topics and pre-requisites
- Course Policies
- Grading Scheme
  - 5 Assignments (60% for ugrad; 50% for grad)
  - Final exam 40%
  - Scribe notes 10% (for grad)
- References

Stable Matching (or marriage)

- n doctors and n hospitals
- each doctor has an ordered preference list of hospitals
- each hospital has an ordered preference list of doctors
- Goal: Find a perfect matching (each doctor matched to one hospital)

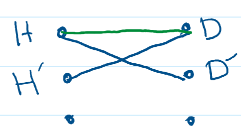
	1st	2nd	3rd	4th	5th
MGH	Bob	Alice	Dorit	Ernie	Clara
BW	Dorit	Bob	Alice	Clara	Ernie
BID	Bob	Ernie	Clara	Dorit	Alice
MTA	Alice	Dorit	Clara	Bob	Ernie
CH	Bob	Dorit	Alice	Ernie	Clara

	1st	2nd	3rd	4th	5th
Alice	CH	MGH	BW	MTA	BID
Bob	BID	BW	MTA	MGH	CH
Clara	BW	BID	MTA	CH	MGH
Dorit	MGH	CH	MTA	BID	BW
Ernie	MTA	BW	CH	BID	MGH

Credit: tables/figures from KW slides

Definition A matching of doctors and hospitals is unstable if there is an "unstable pair"

Suppose (H',D) and (H,D') are two matched pairs; then (H,D) is unstable if H prefers D to D' and D' prefers H to H' (so both H and D prefer to break their current pairing)



so both prefer to break the tie.

Def. A stable matching is a perfect matching with no unstable pairs.

Stable matching problem. Given the preference lists of n hospitals and n doctors, find a stable matching (if one exists)

	1st	2nd	3rd
Atlanta	Xavier	Yolanda	Zeus
Boston	Yolanda	Xavier	Zeus
Chicago	Xavier	Yolanda	Zeus

	1st	2nd	3rd
Xavier	Boston	Atlanta	Chicago
Yolanda	Atlanta	Boston	Chicago
Zeus	Atlanta	Boston	Chicago

A–Y is an unstable pair for matching M = { A–Z, B–Y, C–X }

Question: Do stable matchings always exist?
Not obvious immediately.

We develop an algorithm that always finds one (hence proof of existence too)

Gale-shapely deferred acceptance Algorithm

Input: preference list for hospitals & doctors
Goal: Find a stable matching M
let $M = \phi$
while there is an unmatched hospital h do:
- h offers to the next doctor on its list it has not made an offer before

- if d has no job then add (h,d) to M
- if d has job with h' and h'>h do nothing
- if d has job with h' and h>h' then:

  remove (h',d) from M & add (h,d) to M

return M

	1st	2nd	3rd	4th	5th
MGH	Bob	Alice	Dorit	Ernie	Clara
BW	Dorit	Bob	Alice	Clara	Ernie
BID	Bob	Ernie	Clara	Dorit	Alice
MTA	Alice	Dorit	Clara	Bob	Ernie
CH	Bob	Dorit	Alice	Ernie	Clara

	1st	2nd	3rd	4th	5th
Alice	CH	MGH	BW	MTA	BID
Bob	BID	BW	MTA	MGH	CH
Clara	BW	BID	MTA	CH	MGH
Dorit	MGH	CH	MTA	BID	BW
Ernie	MTA	BW	CH	BID	MGH

Observation: once a doctor gets a job then s/he never becomes jobless

Things to consider:

- The algorithm terminates and outputs a matching
- Hospitals go down" their list; doctors go-up"

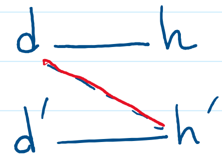
Why a matching at the end?

- No hospital is matched to more than 1 doctor.
- No doctor is matched to more than 1 hospital.
- If a hospital h is not matched at the end--> there is an unmatched doctor d;
- h must have proposed to d; so either it is matched or d was matched.

Why M is stable?

Why M is stable?

- suppose there is an unstable pair: (d,h) and (d',h')



$d: h' > h$

$h': d > d'$

- case 1: h' never offered to d
- case 2: h' made an offer to d:
    - d accepted but later switched to h
    - d rejected, so it was matched to h''>h'>h

This matching is in favor of hospitals; can do it based on preference lists of doctors

National resident matching program (NRMP).

Centralized to match med-school students to hospitals.
Began in 1952 to fix unraveling of offer dates.
Originally used the "Boston Pool" algorithm.
Algorithm overhauled in 1998.
- med-school student optimal
- deals with various side constraints

(e.g., allow couples to match together)

Greedy Algorithms

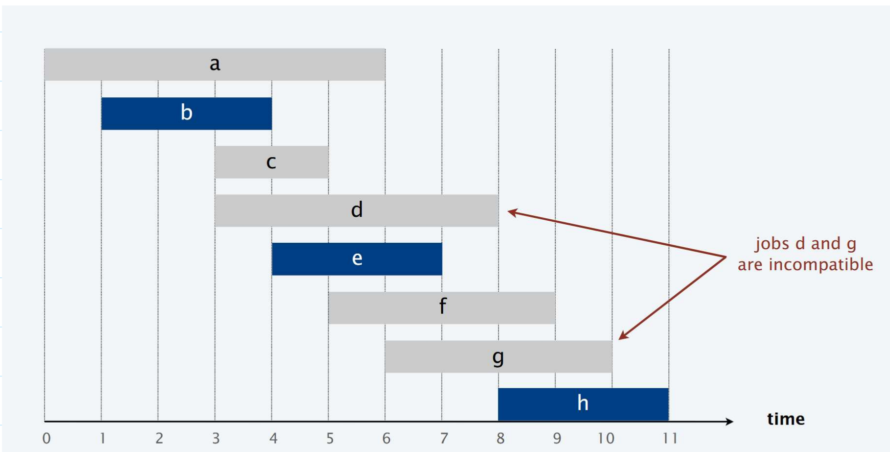
- used for optimization problems (e-g. coin change, shortest paths in weighted graphs, scheduling)

- Decisions made are locally the best and often never changed.

- Algorithms developed are typically efficient.
- Proof often is based on induction and uses an exchange argument.

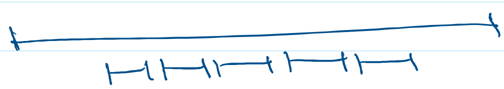
Example 1: Interval scheduling

- Given a set of n jobs J
- each job j has a start time s_j and finish f_j
  - Two jobs are compatible if their intervals don't overlap.
- Goal: Find a largest set of compatible jobs.



Several ways to design by greedy:

1 – sort by s_j



2 – Sort by interval length $f_j - s_j$

3 – Sort by f_j: this might work!

Greedy Interval Scheduling

Sort the jobs based on finish time so $f_1 \leq f_2 \leq \cdots \leq f_n$

let $S = \phi$

For i=1 to n do

if $[s_i, f_i]$ does not conflict with anything in S

add i to S

Return S

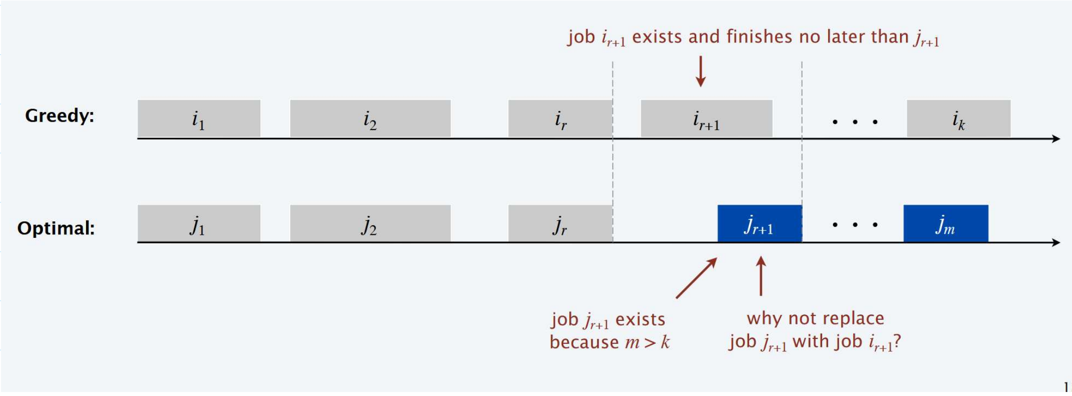
Proposition. Can implement in O(n log n) time.

- Keep track of job j* that was added last to S.
- Job j is compatible with S iff $s_j \geq f_{j^*}$.
- Sorting by finish times takes O(n log n) time.

Theorem: This algorithm finds an optimum schedule.

Proof: Assume greedy is not optimal

- Let $i_1, i_2, \ldots, i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots, j_m$ denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.
- Clearly m>k



- we can replace j_{r+1} in opt with i_{r+1}
- We can use this fact to prove the following by induction on m:

Lemma: For any m>=1, after our algorithm completes m intervals, an optimum has completed <=m intervals.

Example 2: What if we have to schedule all the jobs but on minimum number of machines?

Exercie: Think of another Greedy Algorithm for this problem.

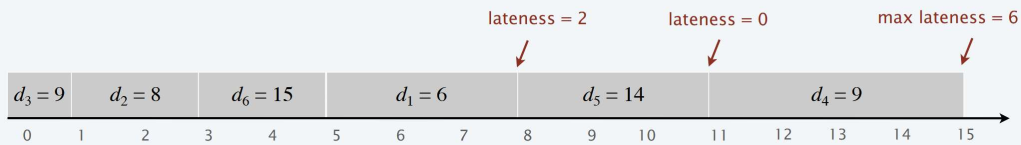
Example 3: Minimum Lateness schedule

Input: n jobs, each has a length t_i and deadline d_i

- if job i starts at time s_i will finish at time s_i+t_i
- and will have lateness $\max\{0, f_i-d_i\}=\ell_i$

Goal: Find an ordering of the jobs (to run on a machine) that minimizes the $\max_i \{\ell_i\}$

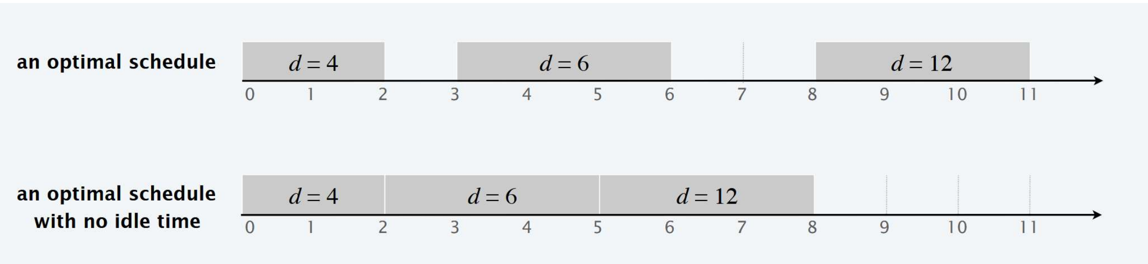
	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



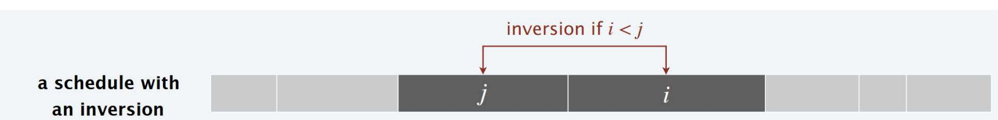
Greedy: What order?

sort by deadline s.t. $d_1 \leq d_2 \leq \cdots \leq d_n$

Observation 1: There is an optimum solution with no idle time.



Definition: Given a schedule S, an inversion is a pair of jobs i and j such that: $i < j$ but j is scheduled before i.



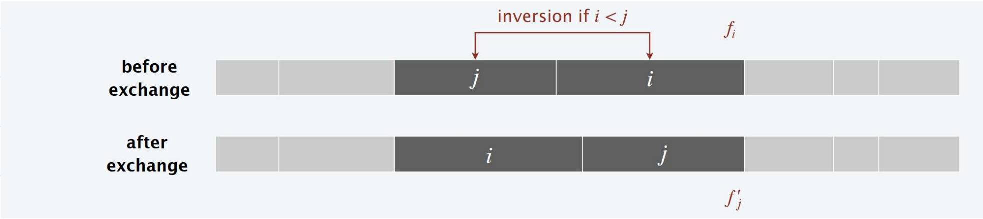
recall: we assume the jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$

Observation 2. The earliest-deadline-first schedule is the unique idle-free schedule with no inversions.

Observation 3. If an idle-free schedule has an inversion, then it has an adjacent inversion

(think of sorting, if it's not sorted two adjacent items are wrong order)

Key Observation: Exchanging two adjacent, inverted jobs i and j reduces the number of inversions by 1 and does not increase the max lateness.



suppose we swap i,j. let ℓ'_i, ℓ'_j be the new lateness of these jobs.
Note: lateness of other jobs don't change and $\ell'_i \leq \ell_i$

Theorem: Earliest deadline-first schedule S is optimum.

Proof: Define S* to be an optimal schedule with the fewest inversions.

- Can assume S* has no idle time.
- Case 1. [S* has no inversions] Then S = S*.
- Case 2. [S* has an inversion]: let i–j be an adjacent inversion

– exchanging jobs i and j decreases the number of inversions by 1 without increasing the max lateness

– contradicts "fewest inversions" part of the definition of S*.

Example 3: Minimum Lateness schedule

Input: n jobs, each has a length t_i and deadline d_i

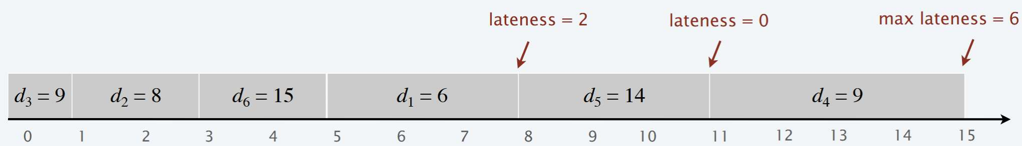
- if job i starts at time s_i will finish at time $s_i + t_i$

Input: n jobs, each has a length t_i and deadline d_i

- if job i starts at time s_i will finish at time $s_i + t_i$
- and will have lateness $\max\{0, f_i - d_i\} = \ell_i$

Goal: Find an ordering of the jobs (to run on a machine) that minimizes the $\max_i \{\ell_i\}$

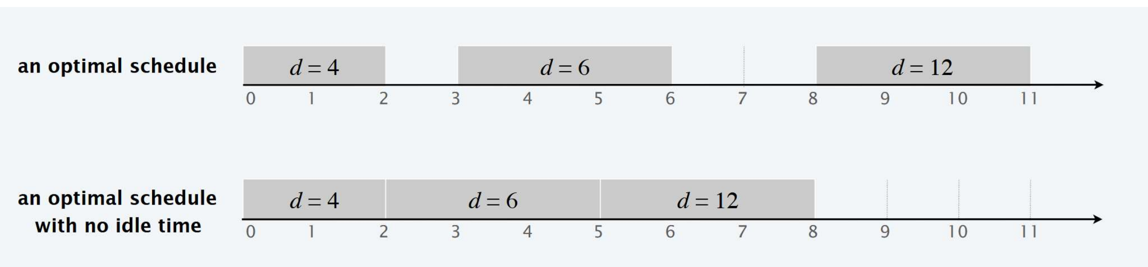
	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



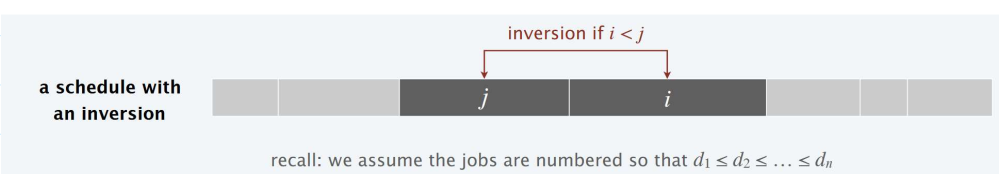
Greedy: What order?

sort by deadline s.t. $d_1 \le d_2 \le \cdots \le d_n$

Observation 1: There is an optimum solution with no idle time.



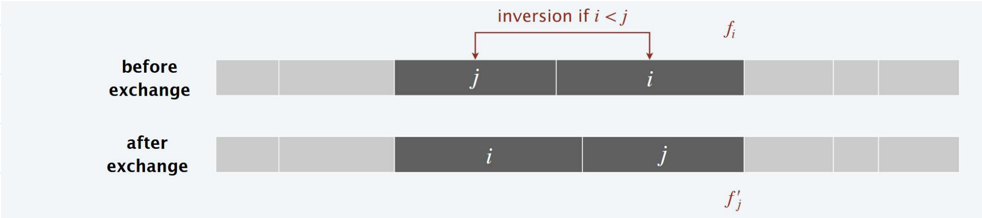
Definition: Given a schedule S, an inversion is a pair of jobs i and j such that: $i < j$ but j is scheduled before i.



Observation 2. The earliest-deadline-first schedule is the unique idle-free schedule with no inversions.

Observation 3. If an idle-free schedule has an inversion, then it has an adjacent inversion
(think of sorting, if it's not sorted two adjacent items are wrong order)

Key Observation: Exchanging two adjacent, inverted jobs i and j reduces the number of inversions by 1 and does not increase the max lateness.



suppose we swap i,j. let ℓ'_i, ℓ'_j be the new lateness of these jobs.
Note: lateness of other jobs don't change and $\ell'_i \leq \ell_i$

Theorem: Earliest deadline-first schedule S is optimum.

Proof: Define S*` to be an optimal schedule with the fewest inversions.
· Can assume S* has no idle time.
· Case 1. [S* has no inversions] Then S = S*.
· Case 2. [S* has an inversion]: let i–j be an adjacent inversion
- exchanging jobs i and j decreases the number of inversions by

1 without increasing the max lateness
- contradicts "fewest inversions" part of the definition of S^*.

Dynamic Programming

- One of the most powerful technique in designing efficient algorithms; often asked about in job interview
- It can be quite involved and solve intricate problems
- The basic principle is simple but coming up with the right approach that works can be quite challenging.

Main idea:

- break the problem to smaller subproblems
- Solve the subproblems and Store the partial solutions into a table
- Use partial solutions recursively to solve the bigger Subproblems.

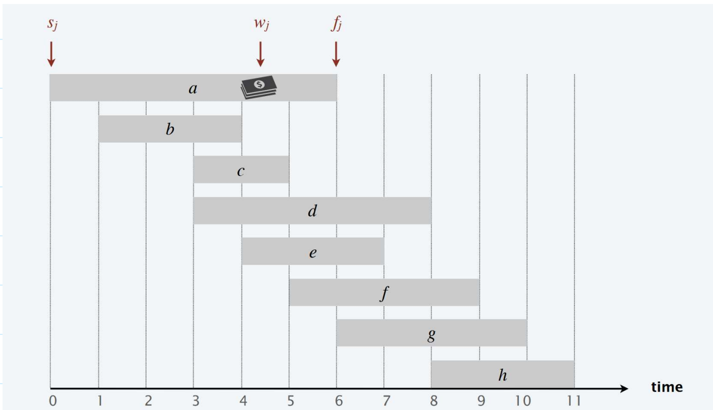
Most important step (of missed by students)

> Define the proper subproblem & a table to store the solution
> (what is it you are storing?)

Example 1: weighted interval scheduling

Given a set J of n jobs: s_i start time
f_i finish time
w_i value/weight

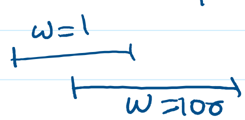
Goal: find a subset of compatible jobs (no two overlap)
with maximum total value.



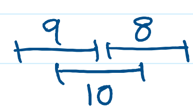
How does greedy do?

We saw greedy works well when all $w_i = 1$.

- Earliest finish time first:



- Decide based on largest w_i to smaller:

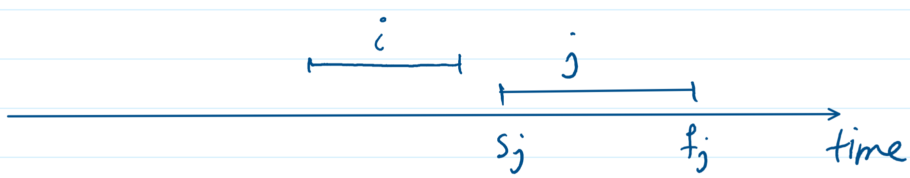


What is a good subproblem?

- lets consider the jobs in increasing order of finish time; so $f_1 \le f_2 \le \cdots \le f_n$
- For each job j let $p(j)$ be the largest index $i < j$ s.t. job i does not overlap with j. ($p(j) = 0$ if no such job exists)
- Let $Opt[j]$ denote the max weight of a .

schedule that uses only jobs (a subset) of jobs in $\{1, 2, \dots, j\}$ Def of subproblem

- Clearly our goal is computing $Opt[n]$ and $opt[1]$ is trivial $(= \omega_1)$.
- When considering job j:
  - case 1: optimum of the firt j jobs does <u>NOT</u> inclued j; So $opt[j] = opt[j-1]$
  - case 2: j belongs to the best schedule of the first j, So it must be the last in that schedule and we have to find the best schedule of jobs $1 \dots p(j)$



$$opt[j] = \omega_j + opt[p(j)]$$

- we don't know which of the two cases so

$$\underset{j \geq 1}{opt[j]} = \max \begin{cases} opt[j-1] \\ opt[p(j)] + \omega_j \end{cases}$$

$$opt[0] = 0$$

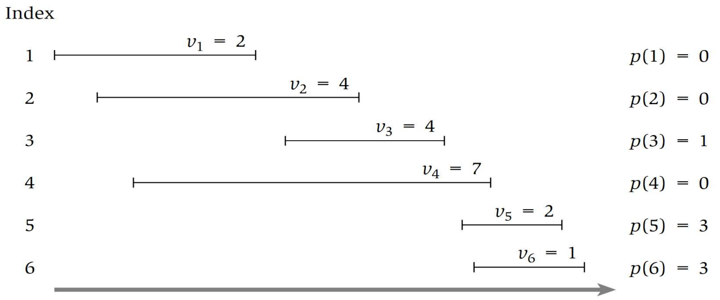
i	1	2	3	4	5	6
opt	2	4	6			

6: 4 (w.o.3), $2+4$ (w.3)

Index

$v_{\_} = 2$

w.o.3 w.3



Weighted_interval_DP

- Sort the jobs s.t. $f_1 \le f_2 \le \dots \le f_n$
- Comput $p(j)$ for each j
- $O[0] = 0$
- for $i \leftarrow 1$ to n do

$$O[i] = \max\{O[i-1],\ w_i + O[P(i)]\}$$

- return $O[n]$

- This computes the value of optimum in $O(n\log n)$
- To find the actual schedule: trace back the selection