BBM402-Lecture 5: Greedy algorithms: tape sorting, scheduling, exchange arguments

Lecturer: Lale Özkahya

Resources for the presentation:
https://courses.engr.illinois.edu/cs374/fa2016/lectures.html
https://courses.engr.illinois.edu/cs374/fa2015/lectures.html

Backtracking

- We have seen Backtracking/DP so far

— Make a simple choice

— Recursively solve everything else

e.g. **Subset Sum** : is a certain element of the set in the subset or not? If only we could know…

Backtracking

- We have seen Backtracking/DP so far

Try all options for

— ~~Make~~ a simple choice

— Recursively solve everything else For each choice!

e.g. **Subset Sum** : is a certain element of the set in the subset or not? If only we could know…

LIS: Do I include an element in the sequence or not?

NFA accept: should I transition to a certain state?

(see nondeterminism)

Backtracking

- We have seen Backtracking/DP so far

Try all options for

— ~~Make~~ a simple choice

— Recursively solve everything else For each choice!

Greedy

Really tempting to

- Choose one option
- Recurse (e.g. Edit Distance: choose two characters that are equal to leave them as such)

Course Policy on Greedy

- When you use greedy algorithm, you need to ALWAYS prove correctness. Otherwise you get a zero, EVEN IF THE ALGORITHM IS CORRECT!

- Greedy is a loaded gun!

Greedy Algorithm Example

- Sorting files on magnetic tape (not RAM)
- Remember music cassettes?
- Blue Water Supercomputer.

Greedy Algorithm Example

- Sorting files on magnetic tape (not RAM)
- Remember music cassettes?
- Blue Water Supercomputer.

The Problem:

- Given an array of lengths of each file: L[1…n]
- I want to sort the files so that if someone asks me for a random file, the expected time it takes to wind the tape to the start of the file and rewind it back is small.

Sorting Files on Tape

The Problem:

- Given an array of lengths of each file: L[1…n]

- I want to sort the files so that if someone asks me for a random file, the expected time it takes to wind the tape to the start of the file and rewind it back is small.

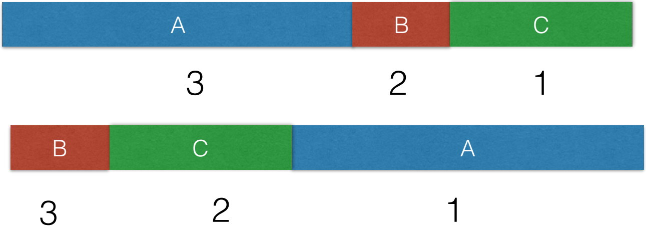
- Formally, I want to find a permutation that minimizes

$$\sum_{k=1}^{n} \sum_{i=1}^{k} L[\pi(i)]$$

Where $\pi(i)$ is the index of the file sorted in position i of the tape

Sorting Files on Tape

What order should I sort them?



Claim:

Sort L, in increasing order of lengths is the best solution

$$L[\pi(i)] \leq L[\pi(i+1)]$$ for all i

Needs proof!!!

Sorting Files on Tape

Proof:

Assume in optimal ordering π

$$L[\pi(i)] > L[\pi(i+1)] \quad \text{for some} \quad i$$

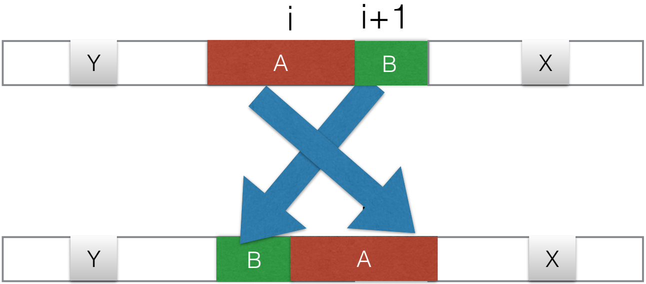
what happens if we switch A and B?

Sorting Files on Tape

Proof:

Assume in optimal ordering π

$$L[\pi(i)] > L[\pi(i+1)] \quad \text{for some} \quad i$$



Cost(A) increases by L[B]
Cost(B) decreases by L[A]

Total cost increases by L[B]-L[A] <0

Exchange Argument

- Consider any non-greedy solution
- Perform an exchange to make the solution look more greedy
- Argue that the new solution after doing the exchange is no worse.
- In out example, the new solution was strictly better, so greedy is the only way.

Sorting Files on Tape

- What if I also had frequencies?

- L[1…n] lengths of files and F[1…n] frequencies.

- Need to minimize: $$\sum_{k=1}^{n}\sum_{i=1}^{k}(F[\pi(k)] \cdot L[\pi(i)])$$

- If all the lengths the same and frequencies different?

Punchline: Swapping adjacent files A,B increases cost by L[B]F[A]-L[A]F[B] <0

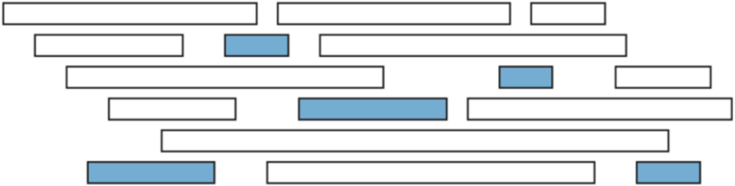
L F

Class Scheduling

- University decides to start a new major, CS+ climbing
- Degree requirements involve taking certain number of classes, certain hours and certain categories.
- Bulk of the degree is determined by taking a certain number of classes. None of these classes require actual work.
- Without the instructors permission, you cannot register for two classes whose times overlap.
- You only need to sign up! Goal: sign up for as many classes as possible, without overlapping classes.

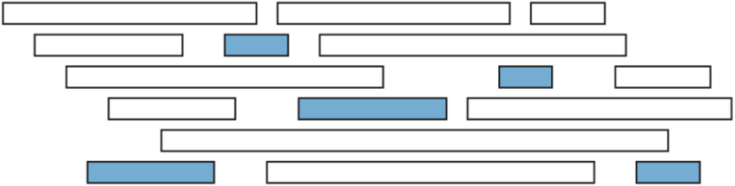
Class Scheduling

- Given a collection of intervals with start and end time, want to choose a subset of those intervals such that no pair overlaps.
- Subset needs to be as large as possible.
- Model it as a graph problem (next time): Independent Set!



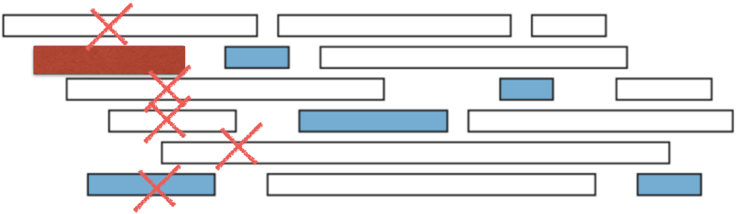
Class Scheduling

- Algorithm? DP? Greedy?
- e.g. find the earliest class, take it and recurse
- find the longest class, throw it away and recurse.
- find the shortest class, take it and recurse.



Class Scheduling

- None of those work!
- Instead: pick the class the ends earliest



Class Scheduling

- None of those work!
- Instead: pick the class the ends earliest

Class Scheduling

- None of those work!
- Instead: pick the class the ends earliest

Class Scheduling

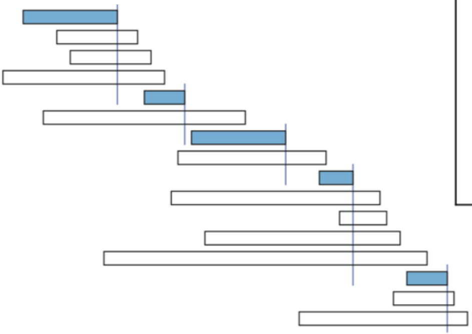
- None of those work!
- Instead: pick the class the ends earliest

Class Scheduling

- None of those work!
- Instead: pick the class the ends earliest

Class Scheduling

- Sort classes according to finish time

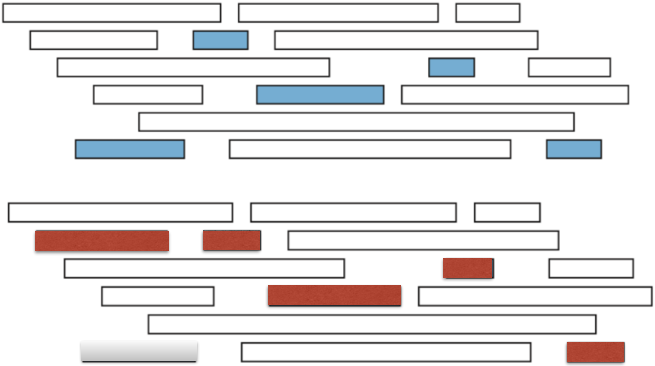


```
GreedySchedule(S[1..n], F[1..n]):
  sort F and permute S to match
  count ← 1
  X[count] ← 1
  for i ← 2 to n
        if S[i] > F[X[count]]
              count ← count + 1
              X[count] ← i
  return X[1..count]
```

Because of sorting, O(nlogn), while DP in O(n^2)

Class Scheduling

- Why is it optimal? Proof!
- Not the only optimal schedule. There are many optimal schedules.



Class Scheduling

- Exchange argument.
- Think of it as a recursive algorithm. Pick the class what finishes first and then recurse.
- Proof by induction!

Lemma:

At least one maximal conflict free schedule includes the class that ends first.

Class Scheduling

Lemma:

At least one maximal conflict free schedule includes the class that ends first.

Proof:

Let f be the class that ends first.

Consider any schedule X that excludes f.

Let g be the first class ending in X.

F[f]<F[g] implies

that f does not overlap any class in X\{g}

Y= X-{g}+{f} is a valid schedule of same size!

What if X is empty?

Huffman Codes

- Binary code assigns a string of 0s and 1s to each character in the alphabet.
- 7-bit ASCII code, Unicode, Morse
- We want the code to be prefix free (Morse code is not).
- Any prefix free code can be visualized as a binary code tree, where the characters are stored at the leafs.
- Codeword for each symbol is given by the path from the root to the corresponding leaf (e.g 1 for right 0 for left).
- Length of codeword for a symbol is the depth of the corresponding leaf.

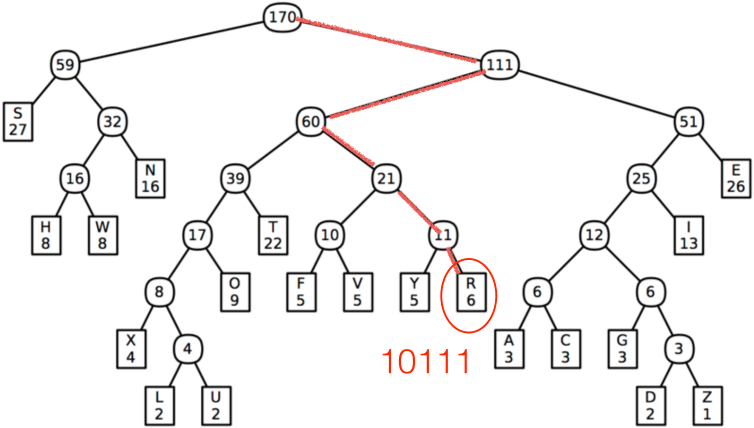
Huffman Codes

- Goal is to encode messages in an n-character alphabet so that the encoded message is as short as possible.

- Given array of frequencies: f[1…n], we want to compute a prefix-free binary code that minimizes the total encoded length of message.

$$\sum_{i=1}^{n} f[i] \cdot \mathrm{depth}(i)$$

Huffman Codes



Huffman Codes

This sentence contains three a's, three c's, two d's, twenty-six e's, five f's, three g's, eight h's, thirteen i's, two l's, sixteen n's, nine o's, six r's, twenty-seven s's, twenty-two t's, two u's, five v's, eight w's, four x's, five y's, and only one z.

A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

Huffman's algorithm: merge two least frequent letters and recurse!

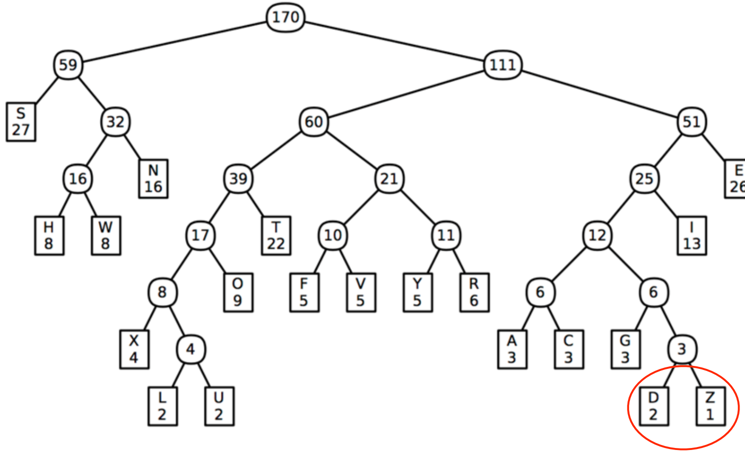
Huffman Codes

A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

A	C	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	DZ
3	3	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	3

Huffman Codes

A	C	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	DZ
3	3	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	3



Huffman Codes

Lemma: Let x and y be the two least frequent characters. There is an optimal code tree in which x and y are siblings, and have the largest depth of any leaf.

Proof: Exchange argument!

Assume, for the optimal schedule that the deepest two leaves are not x and y.

```
BuildHuffman(f[1..n]):
    for i ← 1 to n
        L[i] ← 0; R[i] ← 0
        Insert(i, f[i])

    for i ← n to 2n − 1
        x ← ExtractMin()
        y ← ExtractMin()
        f[i] ← f[x] + f[y]
        L[i] ← x; R[i] ← y
        P[x] ← i; P[y] ← i
        Insert(i, f[i])

    P[2n − 1] ← 0
```

```
HuffmanEncode(A[1..k]):
    m ← 1
    for i ← 1 to k
        HuffmanEncodeOne(A[i])

HuffmanEncodeOne(x):
    if x < 2n − 1
        HuffmanEncodeOne(P[x])
        if x = L[P[x]]
            B[m] ← 0
        else
            B[m] ← 1
        m ← m + 1
```

```
HuffmanDecode(B[1..m]):
    k ← 1
    v ← 2n − 1
    for i ← 1 to m
        if B[i] = 0
            v ← L[v]
        else
            v ← R[v]

        if L[v] = 0
            A[k] ← v
            k ← k + 1
            v ← 2n − 1
```

Part I

Greedy Algorithms: Tools and Techniques

What is a Greedy Algorithm?

What is a Greedy Algorithm?

No real consensus on a universal definition.

What is a Greedy Algorithm?

No real consensus on a universal definition.

Greedy algorithms:

1. make decision incrementally in small steps *without backtracking*
2. decision at each step is based on improving *local or current* state in a myopic fashion without paying attention to the *global* situation
3. decisions often based on some fixed and simple *priority* rules

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to a first-cut heuristic when problem not well understood

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to a first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work. Easy to lull oneself into believing they work
2. Many greedy algorithms possible for a problem and no structured way to find effective ones

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to a first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work. Easy to lull oneself into believing they work
2. Many greedy algorithms possible for a problem and no structured way to find effective ones

CS 374: Every greedy algorithm needs a proof of correctness

Greedy Algorithm Types

Crude classification:

1. Non-adaptive: fix some ordering of decisions a priori and stick with the order
2. Adaptive: make decisions adaptively but greedily/locally at each step

Greedy Algorithm Types

Crude classification:

1. Non-adaptive: fix some ordering of decisions a priori and stick with the order
2. Adaptive: make decisions adaptively but greedily/locally at each step

Plan:

1. See several examples
2. Pick up some proof techniques

Part II

Scheduling Jobs to Minimize Average Waiting Time

The Problem

- $\mathbf{n}$ jobs $\mathbf{J_1, J_2, \ldots, J_n}$. $\mathbf{J_i}$ has non-negative processing time $\mathbf{p_i}$
- One server/machine/person available to process jobs.
- Schedule/order the jobs to minimize total or average *waiting time*
- Waiting time of $\mathbf{J_i}$ in schedule $\boldsymbol{\sigma}$: sum of processing times of all jobs scheduled before $\mathbf{J_i}$

	$\mathbf{J_1}$	$\mathbf{J_2}$	$\mathbf{J_3}$	$\mathbf{J_4}$	$\mathbf{J_5}$	$\mathbf{J_6}$
time	3	4	1	8	2	6

The Problem

- $\mathbf{n}$ jobs $\mathbf{J_1, J_2, \ldots, J_n}$. $\mathbf{J_i}$ has non-negative processing time $\mathbf{p_i}$
- One server/machine/person available to process jobs.
- Schedule/order the jobs to minimize total or average *waiting time*
- Waiting time of $\mathbf{J_i}$ in schedule $\boldsymbol{\sigma}$: sum of processing times of all jobs scheduled before $\mathbf{J_i}$

	$\mathbf{J_1}$	$\mathbf{J_2}$	$\mathbf{J_3}$	$\mathbf{J_4}$	$\mathbf{J_5}$	$\mathbf{J_6}$
time	3	4	1	8	2	6

Example: schedule is $\mathbf{J_1, J_2, J_3, J_4, J_5, J_6}$. Total waiting time is

$$\mathbf{0 + 3 + (3 + 4) + (3 + 4 + 1) + (3 + 4 + 1 + 8) + \ldots =}$$

The Problem

- $\mathbf{n}$ jobs $\mathbf{J_1, J_2, \ldots, J_n}$. $\mathbf{J_i}$ has non-negative processing time $\mathbf{p_i}$
- One server/machine/person available to process jobs.
- Schedule/order the jobs to minimize total or average *waiting time*
- Waiting time of $\mathbf{J_i}$ in schedule $\boldsymbol{\sigma}$: sum of processing times of all jobs scheduled before $\mathbf{J_i}$

	$\mathbf{J_1}$	$\mathbf{J_2}$	$\mathbf{J_3}$	$\mathbf{J_4}$	$\mathbf{J_5}$	$\mathbf{J_6}$
time	3	4	1	8	2	6

Example: schedule is $\mathbf{J_1, J_2, J_3, J_4, J_5, J_6}$. Total waiting time is

$$\mathbf{0 + 3 + (3 + 4) + (3 + 4 + 1) + (3 + 4 + 1 + 8) + \ldots =}$$

Optimal schedule:

The Problem

- $\mathbf{n}$ jobs $J_1, J_2, \ldots, J_n$. J_i has non-negative processing time p_i
- One server/machine/person available to process jobs.
- Schedule/order the jobs to minimize total or average *waiting time*
- Waiting time of J_i in schedule σ: sum of processing times of all jobs scheduled before J_i

	J_1	J_2	J_3	J_4	J_5	J_6
time	3	4	1	8	2	6

Example: schedule is $J_1, J_2, J_3, J_4, J_5, J_6$. Total waiting time is

$$0 + 3 + (3 + 4) + (3 + 4 + 1) + (3 + 4 + 1 + 8) + \ldots =$$

Optimal schedule: Shortest Job First. $J_3, J_5, J_1, J_2, J_6, J_4$.

Optimality of SJF

Theorem

Shortest Job First gives an optimum schedule for the problem of minimizing total waiting time.

Optimality of SJF

Theorem

Shortest Job First gives an optimum schedule for the problem of minimizing total waiting time.

Proof strategy: exchange argument

Optimality of SJF

Theorem

Shortest Job First gives an optimum schedule for the problem of minimizing total waiting time.

Proof strategy: exchange argument

Assume without loss of generality that job sorted in increasing order of processing time and hence $p_1 \leq p_2 \leq \ldots \leq p_n$ and SJF order is $J_1, J_2, \ldots, J_n$.

Inversions

Definition

A schedule $J_{i_1}, J_{i_2}, \ldots, J_{i_n}$ is said to have an inversion if there are jobs J_a and J_b such that S schedules J_a before J_b, but $p_a > p_b$.

Inversions

Definition

A schedule $\mathbf{J_{i_1}, J_{i_2}, \ldots, J_{i_n}}$ is said to have an inversion if there are jobs $\mathbf{J_a}$ and $\mathbf{J_b}$ such that $\mathbf{S}$ schedules $\mathbf{J_a}$ before $\mathbf{J_b}$, but $\mathbf{p_a} > \mathbf{p_b}$.

Claim

If a schedule has an inversion then there is an inversion between two adjacently *scheduled jobs.*

Proof: exercise.

Proof of optimality of SJF

Recall SJF order is $\mathbf{J_1, J_2, \ldots, J_n}$.

- Let $\mathbf{J_{i_1}, J_{i_2}, \ldots, J_{i_n}}$ be an optimum schedule with fewest inversions.
- If schedule has no inversions then it is identical to SJF schedule and we are done.
- Otherwise there is an $\mathbf{1 \le \ell < n}$ such that $\mathbf{i_\ell > i_{\ell+1}}$ since schedule has inversion among two adjacently scheduled jobs

Proof of optimality of SJF

Recall SJF order is $\mathbf{J_1, J_2, \ldots, J_n}$.

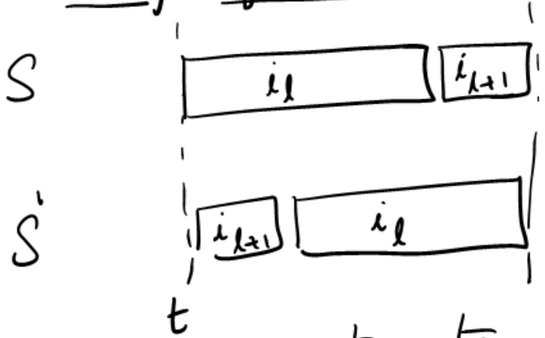
- Let $\mathbf{J_{i_1}, J_{i_2}, \ldots, J_{i_n}}$ be an optimum schedule with fewest inversions.
- If schedule has no inversions then it is identical to SJF schedule and we are done.
- Otherwise there is an $\mathbf{1 \leq \ell < n}$ such that $\mathbf{i_\ell > i_{\ell+1}}$ since schedule has inversion among two adjacently scheduled jobs

Claim

The schedule obtained from $\mathbf{J_{i_1}, J_{i_2}, \ldots, J_{i_n}}$ by exchanging/swapping positions of jobs $\mathbf{J_{i_\ell}}$ and $\mathbf{J_{i_{\ell+1}}}$ is also optimal and has one fewer inversion.

Assuming claim we obtain a contradiction and hence optimum schedule with fewest inversions must be the SJF schedule.

Proof of Claim



Change in waiting time

$$= (t + t + p_{i_{l+1}}) - (t + t + p_{i_l})$$

$$= p_{i_{l+1}} - p_{i_l} \leq 0$$

Part III

Scheduling to Minimize Lateness

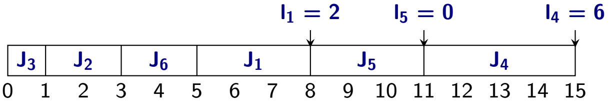
Scheduling to Minimize Lateness

1. Given jobs $J_1, J_2, \ldots, J_n$ with deadlines and processing times to be scheduled on a single resource.
2. If a job i starts at time s_i then it will finish at time $f_i = s_i + t_i$, where t_i is its processing time. d_i: deadline.
3. The lateness of a job is $l_i = \max(0, f_i - d_i)$.
4. Schedule all jobs such that $L = \max l_i$ is minimized.

Scheduling to Minimize Lateness

1. Given jobs $\mathbf{J_1, J_2, \ldots, J_n}$ with deadlines and processing times to be scheduled on a single resource.
2. If a job $\mathbf{i}$ starts at time $\mathbf{s_i}$ then it will finish at time $\mathbf{f_i = s_i + t_i}$, where $\mathbf{t_i}$ is its processing time. $\mathbf{d_i}$: deadline.
3. The lateness of a job is $\mathbf{l_i = \max(0, f_i - d_i)}$.
4. Schedule all jobs such that $\mathbf{L = \max l_i}$ is minimized.

	J_1	J_2	J_3	J_4	J_5	J_6
t_i	3	2	1	4	3	2
d_i	6	8	9	9	14	15



Greedy Template

```
Initially R is the set of all requests
curr_time = 0
max_lateness = 0
while R is not empty do
    choose i ∈ R
    curr_time = curr_time + t_i
    if (curr_time > d_i) then
        max_lateness = max(curr_time − d_i, max_lateness)

return max_lateness
```

Main task: Decide the order in which to process jobs in R

Greedy Template

```
Initially R is the set of all requests
curr_time = 0
max_lateness = 0
while R is not empty do
    choose i ∈ R
    curr_time = curr_time + t_i
    if (curr_time > d_i) then
        max_lateness = max(curr_time − d_i, max_lateness)

return max_lateness
```

Main task: Decide the order in which to process jobs in R

Three Algorithms

1. Shortest job first — sort according to $\mathbf{t_i}$.
2. Shortest slack first — sort according to $\mathbf{d_i} - \mathbf{t_i}$.
3. EDF = Earliest deadline first — sort according to $\mathbf{d_i}$.

$t_1 = 1 \quad d_1 = 3$

$t_2 = 10 \quad d_2 = 12$

Three Algorithms

1. Shortest job first — sort according to $\mathbf{t_i}$.
2. Shortest slack first — sort according to $\mathbf{d_i - t_i}$.
3. EDF = Earliest deadline first — sort according to $\mathbf{d_i}$.

Counter examples for first two: exercise

Earliest Deadline First

Theorem

Greedy with EDF *rule minimizes maximum lateness.*

Earliest Deadline First

Theorem

Greedy with EDF *rule minimizes maximum lateness.*

Proof via an exchange argument.

Earliest Deadline First

Theorem

Greedy with EDF rule minimizes maximum lateness.

Proof via an exchange argument.

Idle time: time during which machine is not working.

Earliest Deadline First

Theorem

Greedy with EDF *rule minimizes maximum lateness.*

Proof via an exchange argument.

Idle time: time during which machine is not working.

Lemma

If there is a feasible schedule then there is one with no idle time before all jobs are finished.

Inversions

Assume jobs are sorted such that $\mathbf{d_1 \leq d_2 \leq \ldots \leq d_n}$. Hence EDF schedules them in this order.

Definition

A schedule $\mathbf{S}$ is said to have an inversion if there are jobs $\mathbf{i}$ and $\mathbf{j}$ such that $\mathbf{S}$ schedules $\mathbf{i}$ before $\mathbf{j}$, but $\mathbf{d_i > d_j}$.

Inversions

Assume jobs are sorted such that $\mathbf{d_1} \leq \mathbf{d_2} \leq \ldots \leq \mathbf{d_n}$. Hence EDF schedules them in this order.

Definition

A schedule $\mathbf{S}$ is said to have an inversion if there are jobs $\mathbf{i}$ and $\mathbf{j}$ such that $\mathbf{S}$ schedules $\mathbf{i}$ before $\mathbf{j}$, but $\mathbf{d_i} > \mathbf{d_j}$.

Claim

If a schedule $\mathbf{S}$ *has an inversion then there is an inversion between two* adjacently *scheduled jobs.*

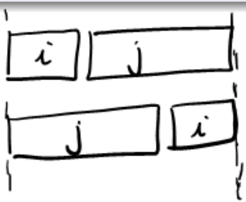
Proof: exercise.

Proof sketch of Optimality of EDP

- Let **S** be an optimum schedule with smallest number of inversions.
- If **S** has no inversions then this is same as EDF and we are done.
- Else **S** has two adjacent jobs **i** and **j** with $\mathbf{d_i} > \mathbf{d_j}$.
- Swap positions of **i** and **j** to obtain a new schedule **S′**

Claim

*Maximum lateness of **S′** is no more than that of **S**. And **S′** has strictly fewer inversions than **S**.*



$d_i > d_j$

Part IV

Maximum Weight Subset of Elements: Cardinality and Beyond

Picking k elements to maximize total weight

1. Given $\mathbf{n}$ items each with non-negative weights/profits and integer $\mathbf{1} \leq \mathbf{k} \leq \mathbf{n}$.
2. Goal: pick $\mathbf{k}$ elements to maximize total weight of items picked.

	$\mathbf{e_1}$	$\mathbf{e_2}$	$\mathbf{e_3}$	$\mathbf{e_4}$	$\mathbf{e_5}$	$\mathbf{e_6}$
weight	3	2	1	4	3	2

$\mathbf{k} = \mathbf{2}$:
$\mathbf{k} = \mathbf{3}$:
$\mathbf{k} = \mathbf{4}$:

Greedy Template

```
N is the set of all elements X ← ∅
(* X will store all the elements that will be picked *)
while |X| < k and N is not empty do
    choose e_j ∈ N of maximum weight
    add e_j to X
    remove e_j from N
return the set X
```

Remark: One can rephrase algorithm simply as sorting elements in decreasing weight order and picking the top $\mathbf{k}$ elements but the above template generalizes to other settings a bit more easily.

Greedy Template

$\mathbf{N}$ is the set of all elements $\mathbf{X} \leftarrow \emptyset$
(* $\mathbf{X}$ will store all the elements that will be picked *)
while $|\mathbf{X}| < \mathbf{k}$ and $\mathbf{N}$ is not empty **do**
    choose $\mathbf{e_j} \in \mathbf{N}$ of maximum weight
    add $\mathbf{e_j}$ to $\mathbf{X}$
    remove $\mathbf{e_j}$ from $\mathbf{N}$
return the set $\mathbf{X}$

Remark: One can rephrase algorithm simply as sorting elements in decreasing weight order and picking the top $\mathbf{k}$ elements but the above template generalizes to other settings a bit more easily.

Theorem

Greedy is optimal for picking $\mathbf{k}$ elements of maximum weight.

A more interesting problem

1. Given $\mathbf{n}$ items $\mathbf{N = \{e_1, e_2, \ldots, e_n\}}$. Each item $\mathbf{e_i}$ has a non-negative weight $\mathbf{w_i}$.
2. Items partitioned into $\mathbf{h}$ sets $\mathbf{N_1, N_2, \ldots, N_h}$. Think of each item having one of $\mathbf{h}$ colors.
3. Given integers $\mathbf{k_1, k_2, \ldots, k_h}$ and another integer $\mathbf{k}$
4. Goal: pick $\mathbf{k}$ elements such that no more than $\mathbf{k_i}$ from $\mathbf{N_i}$ to maximize total weight of items picked.

	$\mathbf{e_1}$	$\mathbf{e_2}$	$\mathbf{e_3}$	$\mathbf{e_4}$	$\mathbf{e_5}$	$\mathbf{e_6}$	$\mathbf{e_7}$
weight	3	2	1	4	3	2	1

$\mathbf{N_1 = \{e_1, e_2, e_3\}}$, $\mathbf{N_2 = \{e_4, e_5\}}$, $\mathbf{N_3 = \{e_6, e_7\}}$
$\mathbf{k = 5}$, $\mathbf{k_1 = 2, k_2 = 1, k_3 = 2}$

Greedy Template

```
N is the set of all elements X ← ∅
(* X will store all the elements that will be picked *)
while N is not empty do
    N' = {e_i ∈ N | X ∪ {e_i} is feasible}
    If N' ← ∅ break
    choose e_j ∈ N' of maximum weight
    add e_j to X
    remove e_j from N
return the set X
```

Greedy Template

```
N is the set of all elements X ← ∅
(* X will store all the elements that will be picked *)
while N is not empty do
    N' = {e_i ∈ N | X ∪ {e_i} is feasible}
    If N' ← ∅ break
    choose e_j ∈ N' of maximum weight
    add e_j to X
    remove e_j from N
return the set X
```

Theorem

Greedy is optimal for the problem on previous slide.

Proof: exercise after class.

Special case of the general phenomenon of Greedy working for maximum weight indepedent set in a matroid. Beyond scope of

Part V

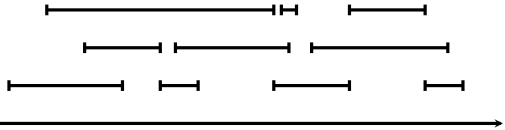
Interval Scheduling

Interval Scheduling

Problem (Interval Scheduling)

Input: *A set of jobs with start and finish times to be scheduled on a resource (example: classes and class rooms).*

Goal: *Schedule as many jobs as possible*



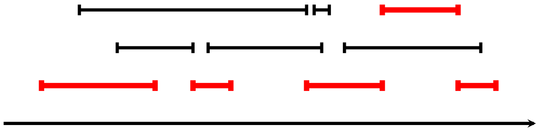
Interval Scheduling

Problem (Interval Scheduling)

Input: *A set of jobs with start and finish times to be scheduled on a resource (example: classes and class rooms).*

Goal: *Schedule as many jobs as possible*

1. *Two jobs with overlapping intervals cannot both be scheduled!*



Greedy Template

```
R is the set of all requests
X ← ∅ (* X will store all the jobs that will be scheduled *)
while R is not empty do
    choose i ∈ R
    add i to X
    remove from R all requests that overlap with i
return the set X
```

Greedy Template

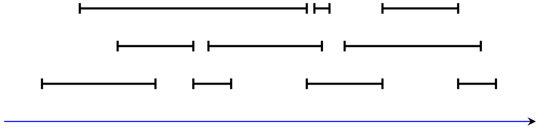
```
R is the set of all requests
X ← ∅ (* X will store all the jobs that will be scheduled *)
while R is not empty do
    choose i ∈ R
    add i to X
    remove from R all requests that overlap with i
return the set X
```

Main task: Decide the order in which to process requests in $\mathbf{R}$

ES SP FC EF

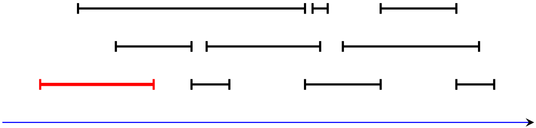
Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.



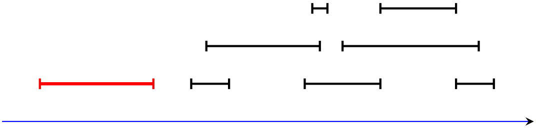
Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.



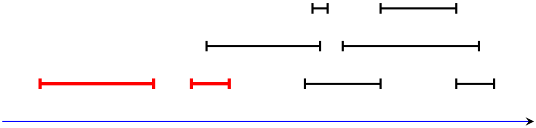
Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.



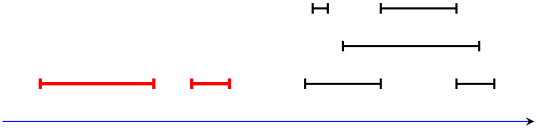
Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.



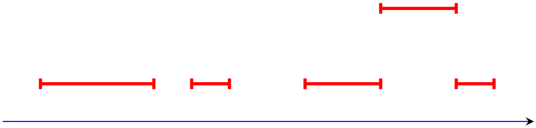
Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.



Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.



Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

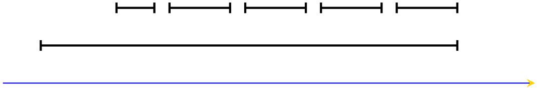


Figure : Counter example for earliest start time

Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

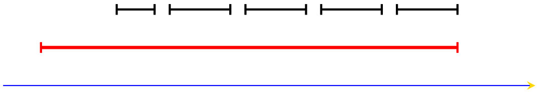


Figure : Counter example for earliest start time

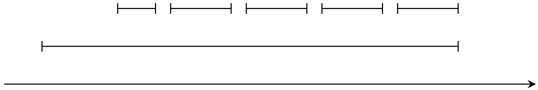
Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

Figure : Counter example for earliest start time

Smallest Processing Time

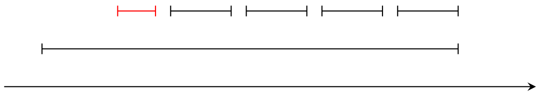
Process jobs in the order of processing time, starting with jobs that require the shortest processing.



Back Counter

Smallest Processing Time

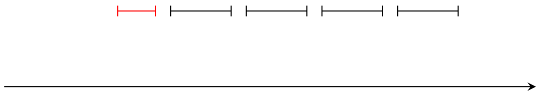
Process jobs in the order of processing time, starting with jobs that require the shortest processing.



Back Counter

Smallest Processing Time

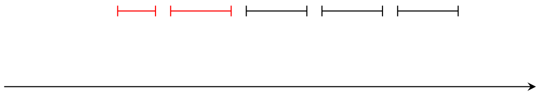
Process jobs in the order of processing time, starting with jobs that require the shortest processing.



Back Counter

Smallest Processing Time

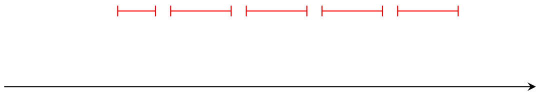
Process jobs in the order of processing time, starting with jobs that require the shortest processing.



Back Counter

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.



Back Counter

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

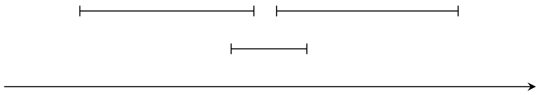


Figure : Counter example for smallest processing time

Back Counter

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

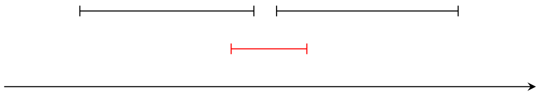


Figure : Counter example for smallest processing time

Back Counter

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

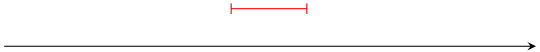
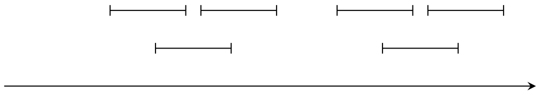


Figure : Counter example for smallest processing time

Back Counter

Fewest Conflicts

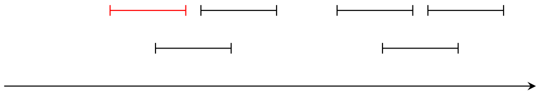
Process jobs in that have the fewest “conflicts” first.



Back Counter

Fewest Conflicts

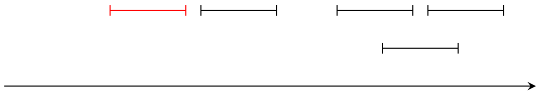
Process jobs in that have the fewest "conflicts" first.



Back Counter

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.



Back Counter

Fewest Conflicts

Process jobs in that have the fewest “conflicts” first.

Back Counter

Fewest Conflicts

Process jobs in that have the fewest “conflicts” first.

Back Counter

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

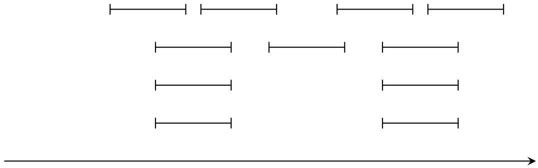


Figure : Counter example for fewest conflicts

Back Counter

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

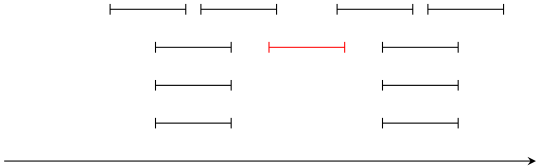


Figure : Counter example for fewest conflicts

Back Counter

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

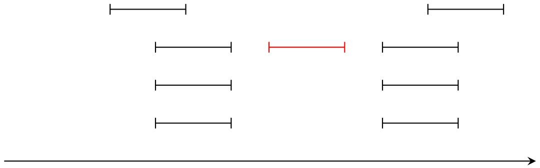


Figure : Counter example for fewest conflicts

Back Counter

Fewest Conflicts

Process jobs in that have the fewest “conflicts” first.

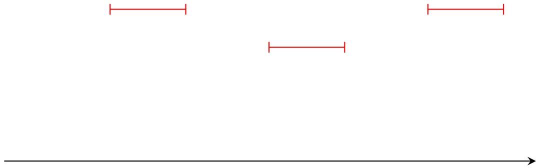
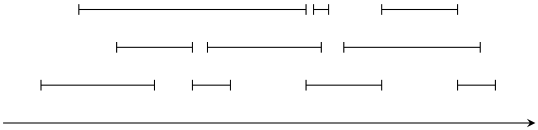


Figure : Counter example for fewest conflicts

Back Counter

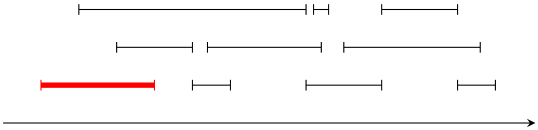
Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.



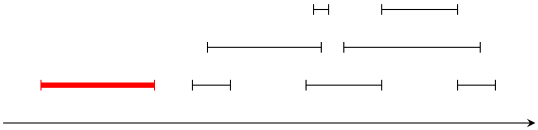
Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.



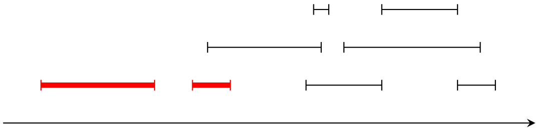
Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.



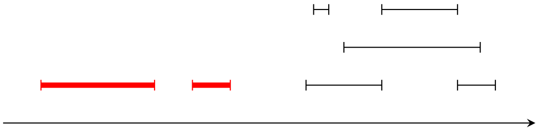
Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.



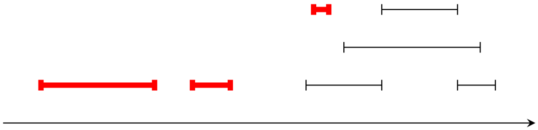
Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.



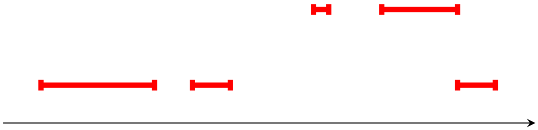
Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.



Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.



Optimal Greedy Algorithm

$\mathbf{R}$ is the set of all requests
$\mathbf{X} \leftarrow \emptyset$ (* $\mathbf{X}$ stores the jobs that will be scheduled *)
while $\mathbf{R}$ is not empty
    choose $\mathbf{i} \in \mathbf{R}$ such that finishing time of $\mathbf{i}$ is smallest
    add $\mathbf{i}$ to $\mathbf{X}$
    remove from $\mathbf{R}$ all requests that overlap with $\mathbf{i}$
return $\mathbf{X}$

Theorem

The greedy algorithm that picks jobs in the order of their finishing times is optimal.

Proving Optimality

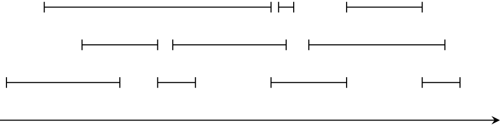
1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts

Proving Optimality

1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests $\mathbf{R}$, let $\mathbf{O}$ be an optimal set and let $\mathbf{X}$ be the set returned by the greedy algorithm. Then $\mathbf{O} = \mathbf{X}$?

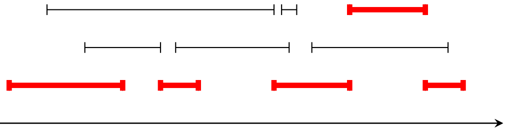
Proving Optimality

1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests $\mathbf{R}$, let $\mathbf{O}$ be an optimal set and let $\mathbf{X}$ be the set returned by the greedy algorithm. Then $\mathbf{O} = \mathbf{X}$?Not likely!



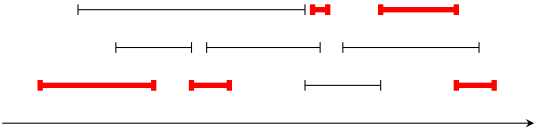
Proving Optimality

1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests **R**, let **O** be an optimal set and let **X** be the set returned by the greedy algorithm. Then **O** = **X**?Not likely!



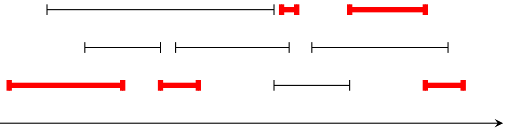
Proving Optimality

1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests $\mathbf{R}$, let $\mathbf{O}$ be an optimal set and let $\mathbf{X}$ be the set returned by the greedy algorithm. Then $\mathbf{O} = \mathbf{X}$?Not likely!



Proving Optimality

1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests $\mathbf{R}$, let $\mathbf{O}$ be an optimal set and let $\mathbf{X}$ be the set returned by the greedy algorithm. Then $\mathbf{O} = \mathbf{X}$?Not likely!



Instead we will show that $|\mathbf{O}| = |\mathbf{X}|$

Proof of Optimality: Key Lemma

Lemma

Let $\mathbf{i_1}$ be first interval picked by Greedy. There exists an optimum solution that contains $\mathbf{i_1}$.

Proof.

Let $\mathbf{O}$ be an *arbitrary* optimum solution. If $\mathbf{i_1} \in \mathbf{O}$ we are done.

Proof of Optimality: Key Lemma

Lemma

Let $\mathbf{i_1}$ *be first interval picked by Greedy. There exists an optimum solution that contains* $\mathbf{i_1}$.

Proof.

Let $\mathbf{O}$ be an *arbitrary* optimum solution. If $\mathbf{i_1} \in \mathbf{O}$ we are done.
Claim: If $\mathbf{i_1} \not\in \mathbf{O}$ then there is exactly one interval $\mathbf{j_1} \in \mathbf{O}$ that conflicts with $\mathbf{i_1}$. (proof later)

Proof of Optimality: Key Lemma

Lemma

Let $\mathbf{i_1}$ *be first interval picked by Greedy. There exists an optimum solution that contains* $\mathbf{i_1}$*.*

Proof.

Let $\mathbf{O}$ be an *arbitrary* optimum solution. If $\mathbf{i_1} \in \mathbf{O}$ we are done.
Claim: If $\mathbf{i_1} \not\in \mathbf{O}$ then there is exactly one interval $\mathbf{j_1} \in \mathbf{O}$ that conflicts with $\mathbf{i_1}$. (proof later)

1. Form a new set $\mathbf{O'}$ by removing $\mathbf{j_1}$ from $\mathbf{O}$ and adding $\mathbf{i_1}$, that is $\mathbf{O' = (O - \{j_1\}) \cup \{i_1\}}$.
2. From claim, $\mathbf{O'}$ is a *feasible* solution (no conflicts).
3. Since $|\mathbf{O'}| = |\mathbf{O}|$, $\mathbf{O'}$ is also an optimum solution and it contains $\mathbf{i_1}$. □

Proof of Claim

Claim

If $\mathbf{i_1} \notin \mathbf{O}$*, there is exactly one interval* $\mathbf{j_1} \in \mathbf{O}$ *that conflicts with* $\mathbf{i_1}$*.*

Proof.

1. If no $\mathbf{j} \in \mathbf{O}$ conflicts with $\mathbf{i_1}$ then $\mathbf{O}$ is not optimal!
2. Suppose $\mathbf{j_1}, \mathbf{j_2} \in \mathbf{O}$ such that $\mathbf{j_1} \neq \mathbf{j_2}$ and both $\mathbf{j_1}$ and $\mathbf{j_2}$ conflict with $\mathbf{i_1}$.
3. Since $\mathbf{i_1}$ has earliest finish time, $\mathbf{j_1}$ and $\mathbf{i_1}$ overlap at $\mathbf{f(i_1)}$.
4. For same reason $\mathbf{j_2}$ also overlaps with $\mathbf{i_1}$ at $\mathbf{f(i_1)}$.
5. Implies that $\mathbf{j_1}, \mathbf{j_2}$ overlap at $\mathbf{f(i_1)}$ but intervals in $\mathbf{O}$ cannot overlap.

See figure in next slide. □

Figure for proof of Claim

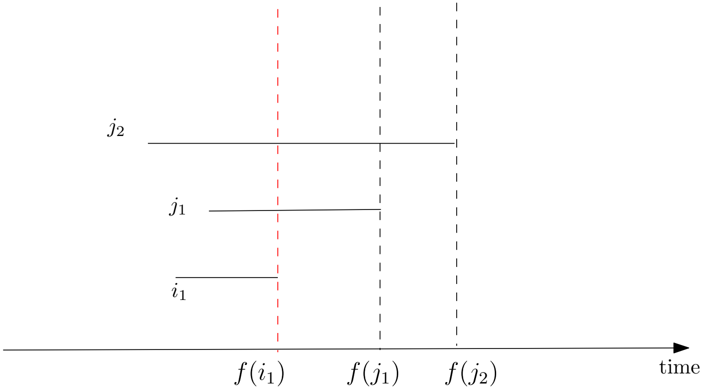


Figure : Since $\mathbf{i_1}$ has the earliest finish time, any interval that conflicts with it does so at $\mathbf{f(i_1)}$. This implies $\mathbf{j_1}$ and $\mathbf{j_2}$ conflict.

Proof of Optimality of Earliest Finish Time First

Proof by Induction on number of intervals.

Base Case: $\mathbf{n = 1}$. Trivial since Greedy picks one interval.
Induction Step: Assume theorem holds for $\mathbf{i < n}$.
Let $\mathbf{I}$ be an instance with $\mathbf{n}$ intervals
$\mathbf{I'}$: $\mathbf{I}$ with $\mathbf{i_1}$ and all intervals that overlap with $\mathbf{i_1}$ removed
$\mathbf{G(I), G(I')}$: Solution produced by Greedy on $\mathbf{I}$ and $\mathbf{I'}$
From Lemma, there is an optimum solution $\mathbf{O}$ to $\mathbf{I}$ and $\mathbf{i_1 \in O}$.
Let $\mathbf{O' = O - \{i_1\}}$. $\mathbf{O'}$ is a solution to $\mathbf{I'}$.

$$\begin{aligned}
|\mathbf{G(I)}| &= 1 + |\mathbf{G(I')}| \quad \text{(from Greedy description)} \\
&\geq 1 + |\mathbf{O'}| \quad \text{(By induction, } \mathbf{G(I')} \text{ is optimum for } \mathbf{I'}\text{)} \\
&= |\mathbf{O}|
\end{aligned}$$

$\square$

Implementation and Running Time

```
Initially R is the set of all requests
X ← ∅ (* X stores the jobs that will be scheduled *)
while R is not empty
    choose i ∈ R such that finishing time of i is least
    if i does not overlap with requests in X
        add i to X
    remove i from R
return the set X
```

- Presort all requests based on finishing time. $O(n \log n)$ time
- Now choosing least finishing time is $O(1)$
- Keep track of the finishing time of the last request added to A. Then check if starting time of i later than that
- Thus, checking non-overlapping is $O(1)$
- Total time $O(n \log n + n) = O(n \log n)$

Comments

1. Interesting Exercise: smallest interval first picks at least half the optimum number of intervals.
2. All requests need not be known at the beginning. Such *online* algorithms are a subject of research

Weighted Interval Scheduling

Suppose we are given $\mathbf{n}$ jobs. Each job $\mathbf{i}$ has a start time $\mathbf{s_i}$, a finish time $\mathbf{f_i}$, and a weight $\mathbf{w_i}$. We would like to find a set $\mathbf{S}$ of compatible jobs whose total weight is maximized. Which of the following greedy algorithms finds the optimum schedule?

- **(A)** Earliest start time first.
- **(B)** Earliest finish time fist.
- **(C)** Highest weight first.
- **(D)** None of the above.
- **(E)** IDK.

Weighted Interval Scheduling

Suppose we are given $\mathbf{n}$ jobs. Each job $\mathbf{i}$ has a start time $\mathbf{s_i}$, a finish time $\mathbf{f_i}$, and a weight $\mathbf{w_i}$. We would like to find a set $\mathbf{S}$ of compatible jobs whose total weight is maximized. Which of the following greedy algorithms finds the optimum schedule?

(A) Earliest start time first.

(B) Earliest finish time fist.

(C) Highest weight first.

(D) None of the above.

(E) IDK.

Weighted problem can be solved via dynamic prog. See notes.

Greedy Analysis: Overview

1. Greedy's first step leads to an optimum solution. Show that there is an optimum solution leading from the first step of Greedy and then use induction. Example, Interval Scheduling.
2. Greedy algorithm stays ahead. Show that after each step the solution of the greedy algorithm is at least as good as the solution of any other algorithm. Example, Interval scheduling.
3. Structural property of solution. Observe some structural bound of every solution to the problem, and show that greedy algorithm achieves this bound. Example, Interval Partitioning (see Kleinberg-Tardos book).
4. Exchange argument. Gradually transform any optimal solution to the one produced by the greedy algorithm, without hurting its optimality. Example, Minimizing lateness.

Takeaway Points

1. Greedy algorithms come naturally but often are incorrect. A proof of correctness is an absolute necessity.
2. *Exchange* arguments are often the key proof ingredient. Focus on why the first step of the algorithm is correct: need to show that there is an optimum/correct solution with the first step of the algorithm.
3. Thinking about correctness is also a good way to figure out which of the many greedy strategies is likely to work.