L10 5800

feb 18/21 2022

shelat

Greedy is only good for certain problems

	start	end
sy3333	2	3.25
en1612	1	4
ma1231	3	4
Cs5800	3.5	4.75
cs4800	4	5.25
cs6051	4.5	6
sy3100	5	6.5
Cs1234	7	8

How many non-overlapping courses can you take?

problem statement

$(a_1, \ldots, a_n)$ activities

$(s_1, s_2, \ldots, s_n)$ start times

end times

$(f_1, f_2, \ldots, f_n)$ (sorted) $s_i < f_i$

find largest subset of activities $C=\{a_i\}$ such that

(compatible)

For any 2 activities a_i, a_j $i<j$

the start time of a_j is after

the end time of a_i.

problem statement

$(a_1, \ldots, a_n)$

$(s_1, s_2, \ldots, s_n)$

$(f_1, f_2, \ldots, f_n)$ (sorted) $s_i < f_i$

find largest subset of activities C={a_i} such that

(compatible)

For any two activities $a_i, a_j, i < j$ the start time of a_j is after the finish time of a_i.

problem statement

$$(a_1, \ldots, a_n)$$

$$(s_1, s_2, \ldots, s_n)$$

$$(f_1, f_2, \ldots, f_n)$$ (sorted) $s_i < f_i$

find largest subset of activities C={a_i} such that

(compatible)

$$a_i, a_j \in C, i < j$$

$$f_i \leq s_j$$

end time of activity i

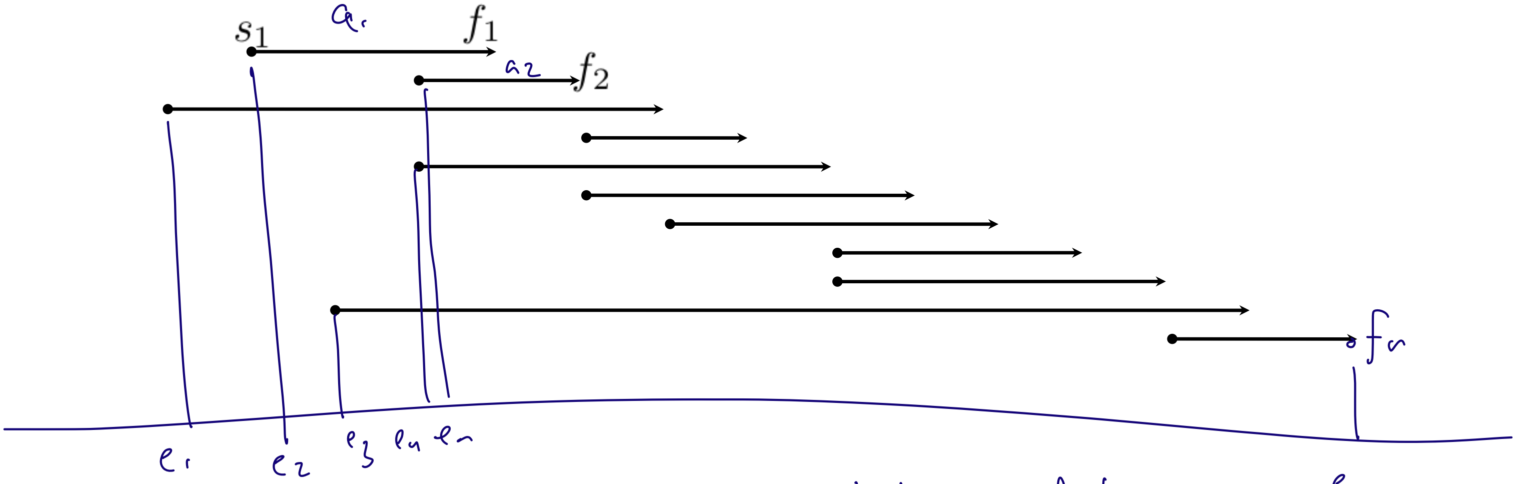
start time of activity j.

problem statement

$(a_1, \ldots, a_n)$

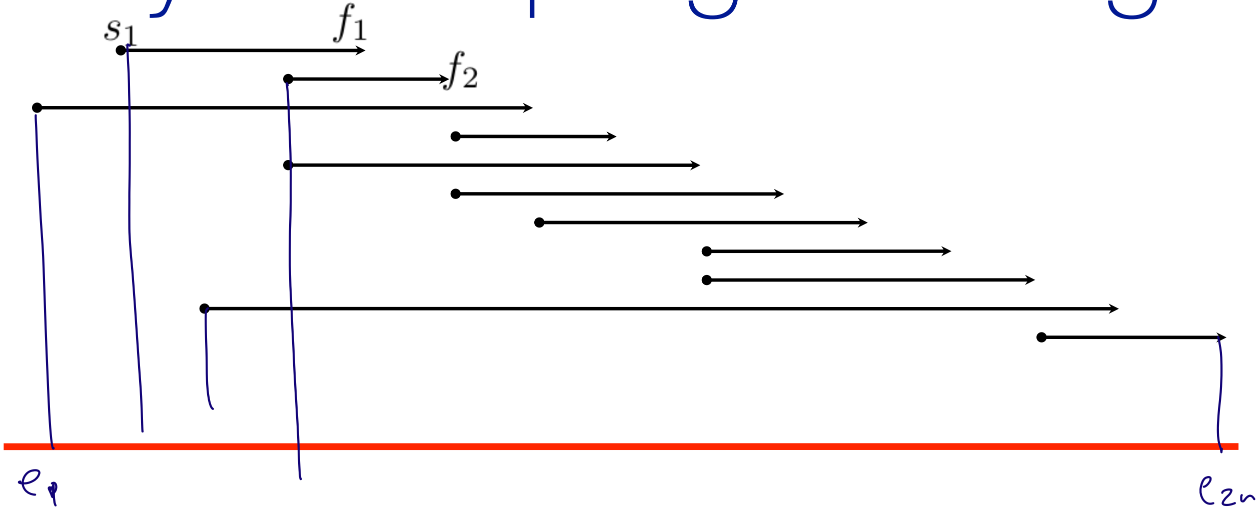
$(s_1, s_2, \ldots, s_n)$

$(f_1, f_2, \ldots, f_n)$ (SORTED) $s_i < f_i$



↑ events are either start or end times.

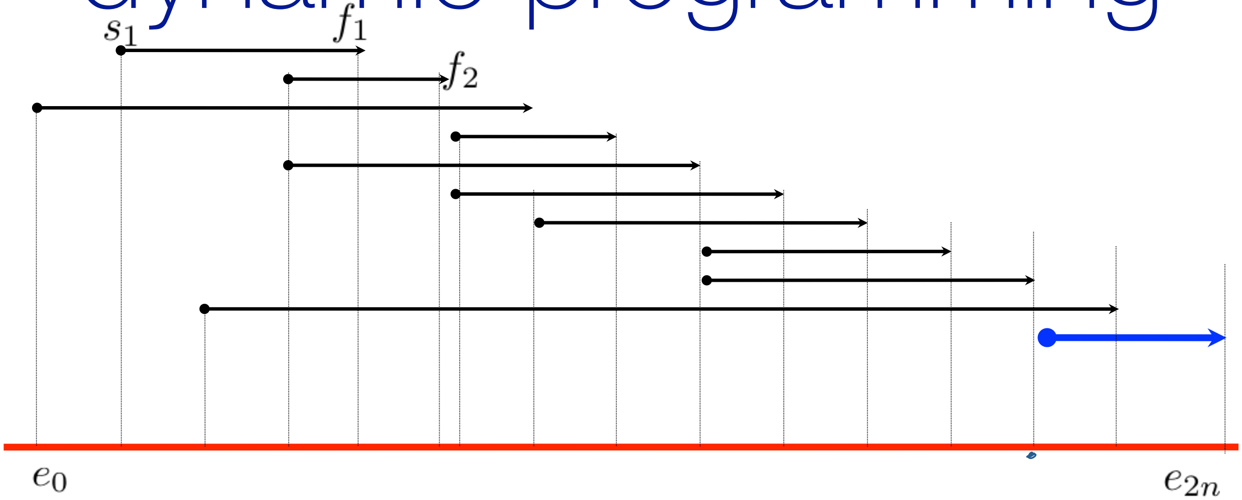
dynamic programming



Best_{2n} = Max # of compatible activities among the first 2n events.

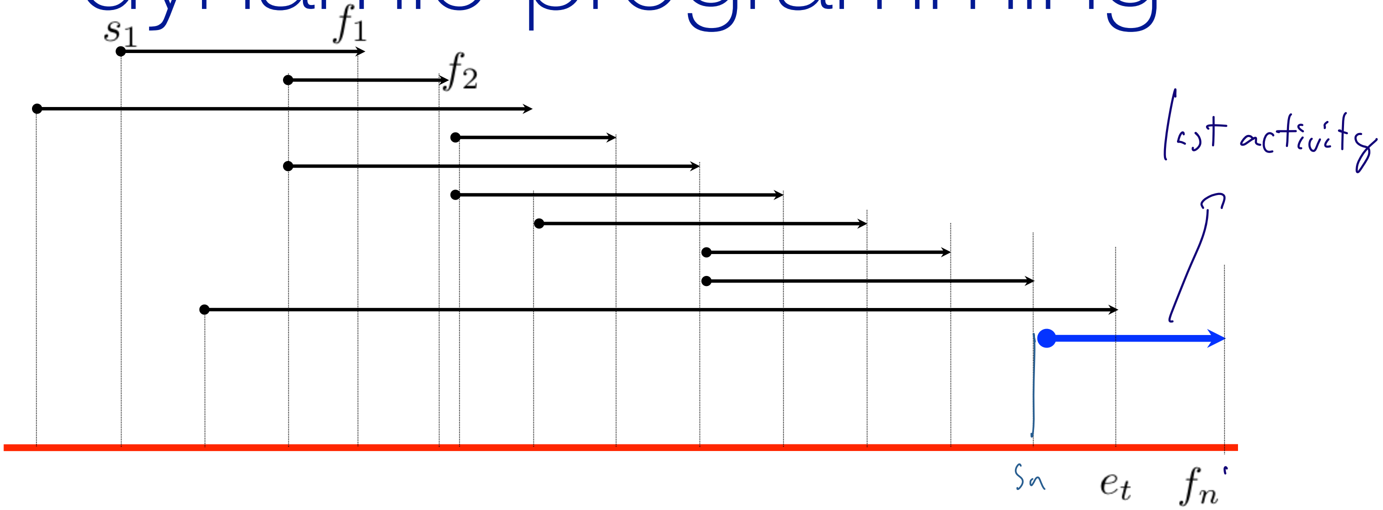
Lets draw all of the events on a timeline.

dynamic programming



$Best_{2n} =$ Maximum number of non-overlapping activities possible among the first 2n events.

dynamic programming

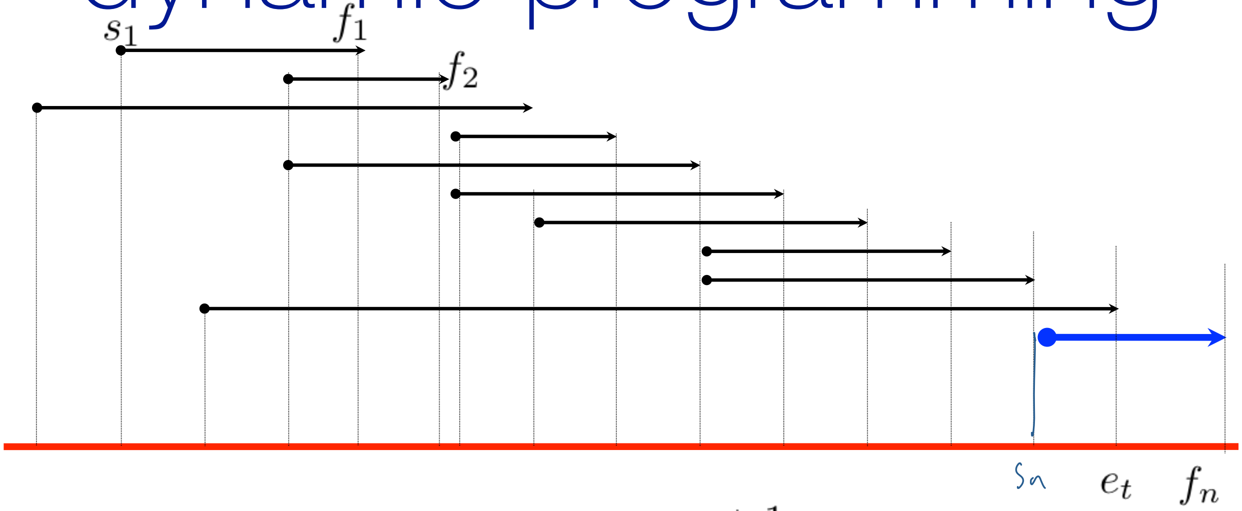


$$\mathrm{BEST}_{f_n} = \mathrm{MAX}\begin{cases} \mathrm{Best}_{e_t} \leftarrow \text{next event} \\ 1 + \mathrm{Best}_{s_n}. \end{cases}$$

we don't include a_n

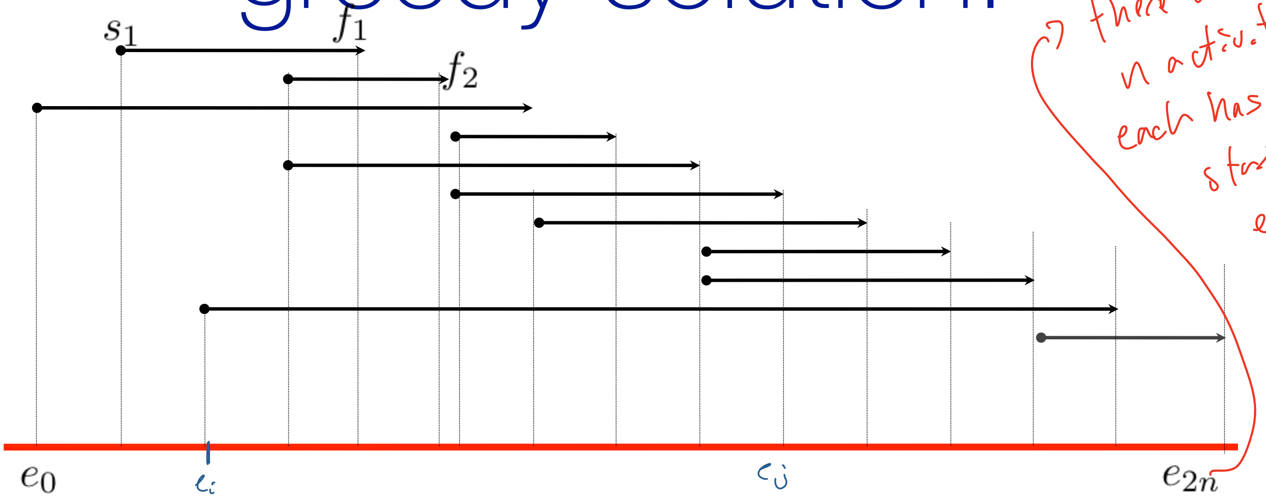
we include a_n in our set.

dynamic programming



$$\text{BEST}_{f_n} = \max \begin{array}{ll} \text{BEST}_{s_n} + 1 & \text{in: } a_n \\ \text{BEST}_{e_t} & \text{out: } a_n \end{array}$$

greedy solution:



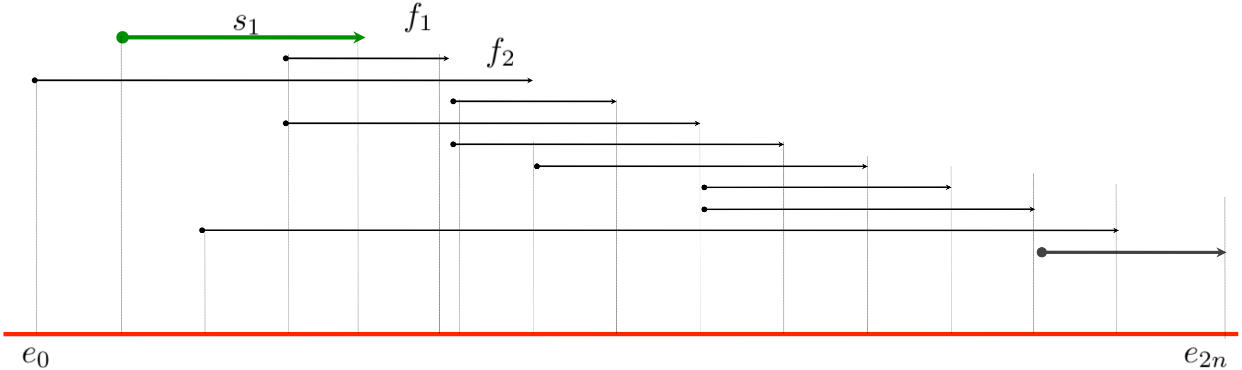
there are n activities. each has a start & end ⇒ 2n events

DEFINITION:

soltn$_{i,j}$ = largest set of compatible activities between event e_i, e_j.

GOAL: SOLTN$_{0,2n}$

greedy solution:



claim: the first action to finish in e[i,j] is always part of some $\textsc{soltn}_{i,j}$

claim: the $\overset{a^*}{\text{first action to finish}}$ in e[i,j] is always part of some $\text{SOLTN}_{i,j}$

(Exchange argument)

PROOF: Consider some optimal solution $\text{SOLTN}_{i,j}$. Let a^* be the first activity to finish between events $[e_i, e_j]$.

Case 1: If $a^* \in \text{SOLTN}_{i,j}$ then the claim is TRUE.

Case 2: If $a^* \notin \text{SOLTN}_{i,j}$, then let a be the first to finish in $\text{SOLTN}_{i,j}$.

Let $\text{SOLTN}^*_{i,j} = \text{SOLTN}_{i,j} - \{a\} \overset{\text{(union)}}{\cup} \{a^*\}$.

$\text{SOLTN}^*_{i,j}$ is non-overlapping because a^* ends before a, and thus cannot overlap with any other activity.

Moreover $|\text{SOLTN}^*| = |\text{SOLTN}|$ and therefore is also optimal.

claim: the first action to finish in e[i,j] is always part of some $\textsc{soltn}_{i,j}$

PROOF:

Consider optimal soltn$_{i,j}$ and let a^* be the first activity to finish in e[i,j].

→ If $a^* \in$ soltn$_{i,j}$, then the claim follows.

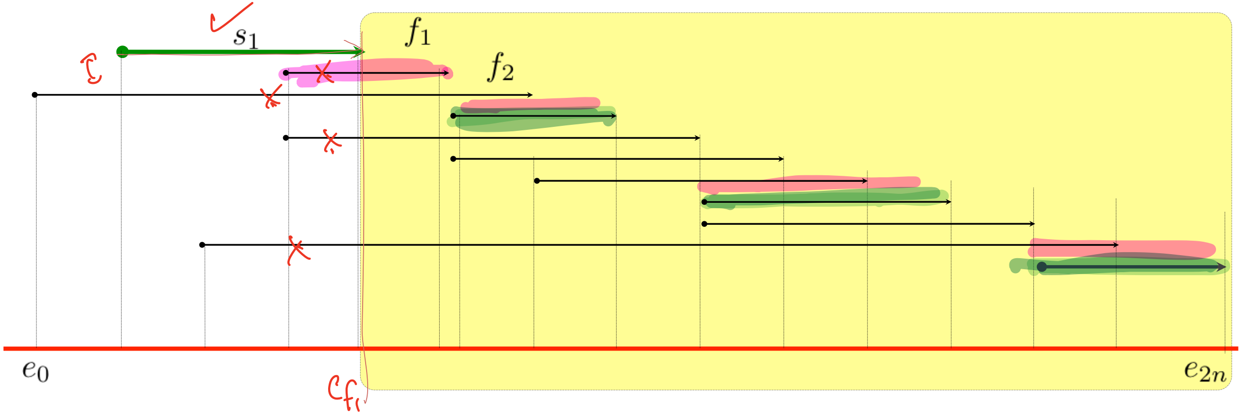
If not, let a be the activity that finishes first in soltn$_{i,j}$.

Consider a new solution that replaces a with a^*.

$$\text{soltn}^*_{i,j} = \text{soltn}_{i,j} - \underline{\{a\}} \cup \underline{\{a^*\}}$$

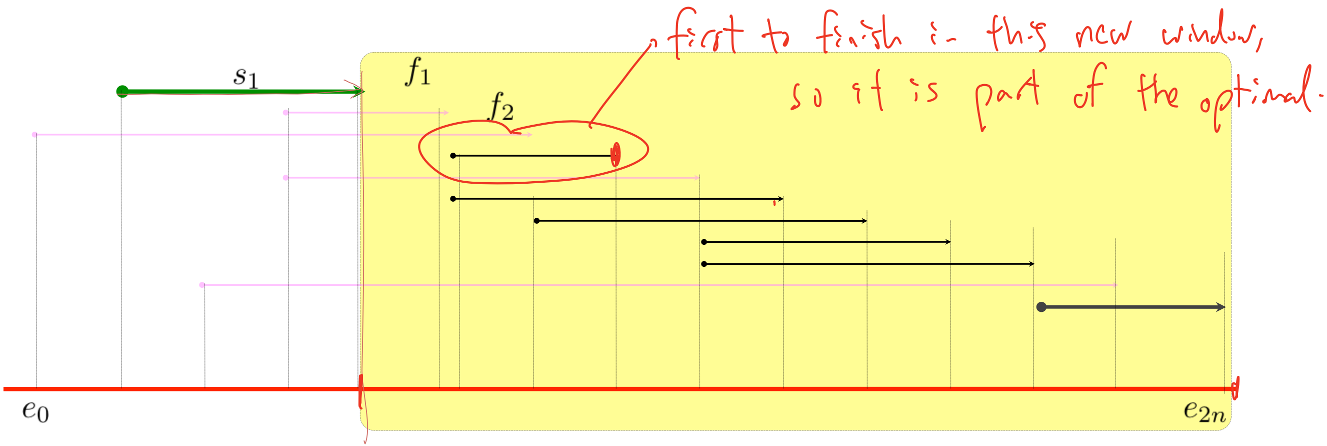
This new set is valid because a^* finishes before a and thus does not overlap with any activities. This new solution also has the same size and is therefore also optimal too.

greedy solution:



algorithm: find first event to finish. add to solution.
remove conflicting events.
continue.

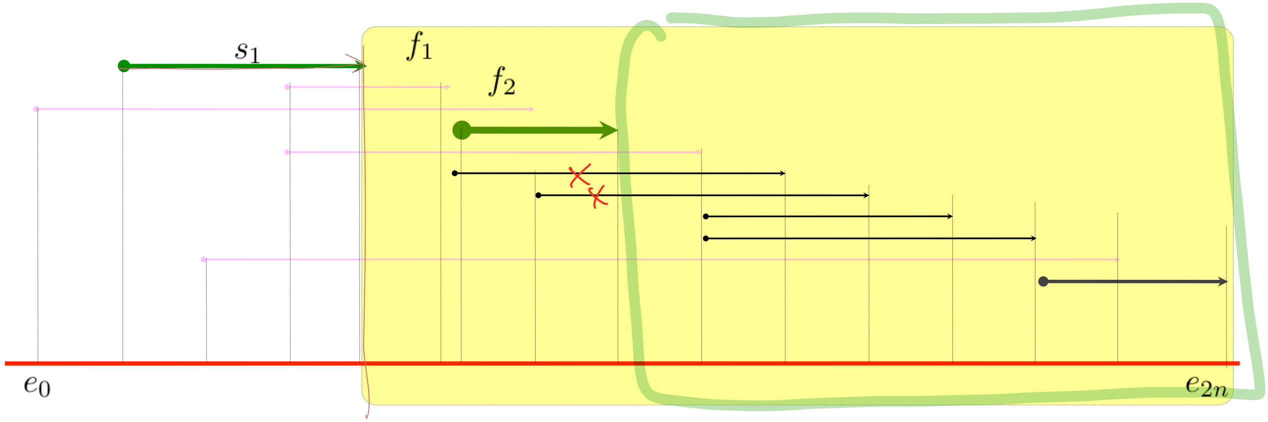
greedy solution:



first to finish in this new window, so it is part of the optimal.

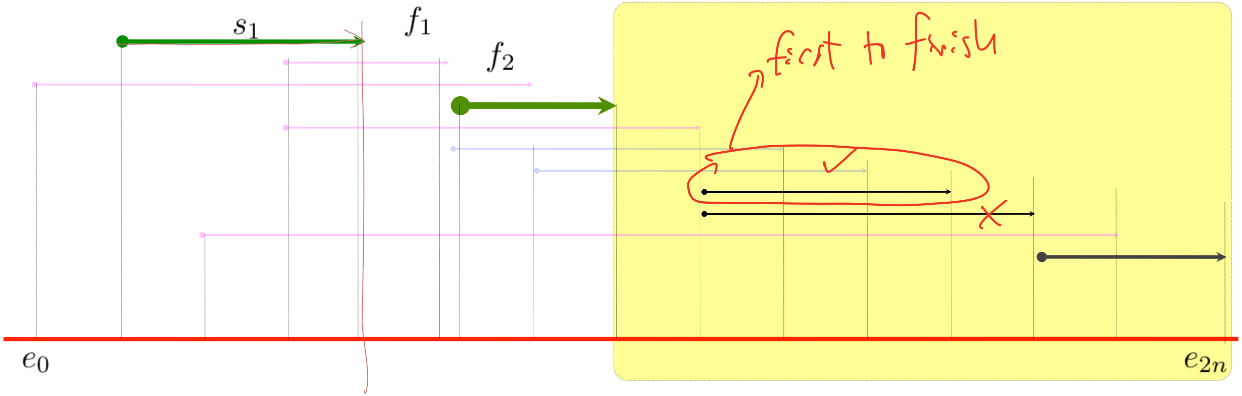
algorithm: find first event to finish. add to solution.
remove conflicting events.
continue.

greedy solution:



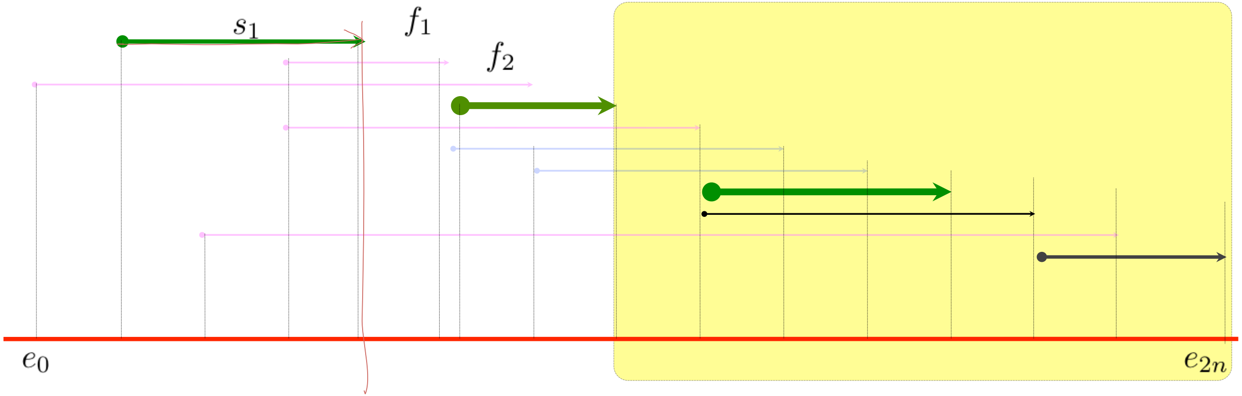
algorithm: find first event to finish. add to solution.
remove conflicting events.
continue.

greedy solution:



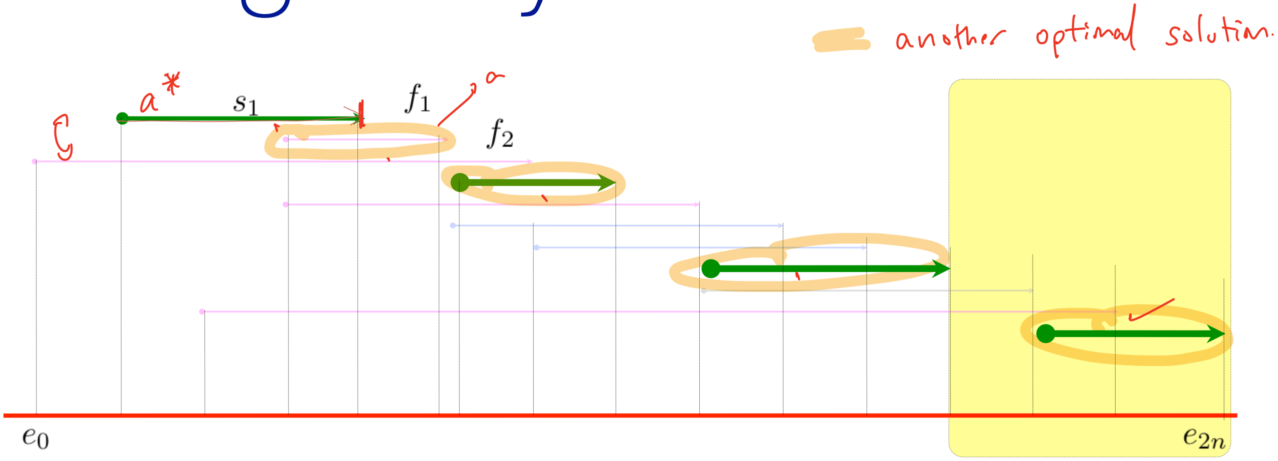
algorithm: find first event to finish. add to solution.
remove conflicting events.
continue.

greedy solution:



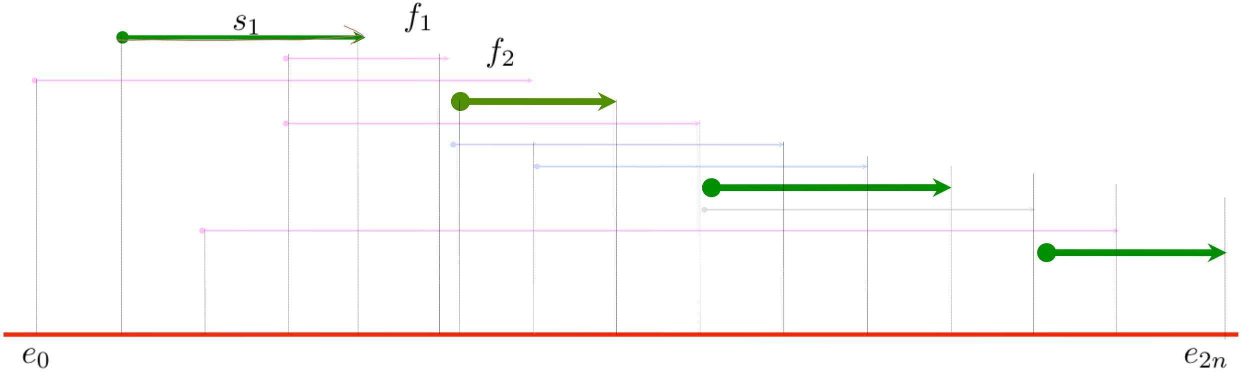
algorithm: find first event to finish. add to solution.
remove conflicting events.
continue.

greedy solution:



algorithm: find first event to finish. add to solution.
remove conflicting events.
continue.

greedy solution:



algorithm: find first event to finish. add to solution.
remove conflicting events.
continue.

running time

algorithm: find first event to finish. add to solution.
remove conflicting events.
continue.

$(f_1, \cancel{f_2}, \cancel{\ldots}, f_n)$ (sorted) $s_i < f_i$

$\Theta(n)$

because we consider each activity only once (either include in solution, or remove).

Recap

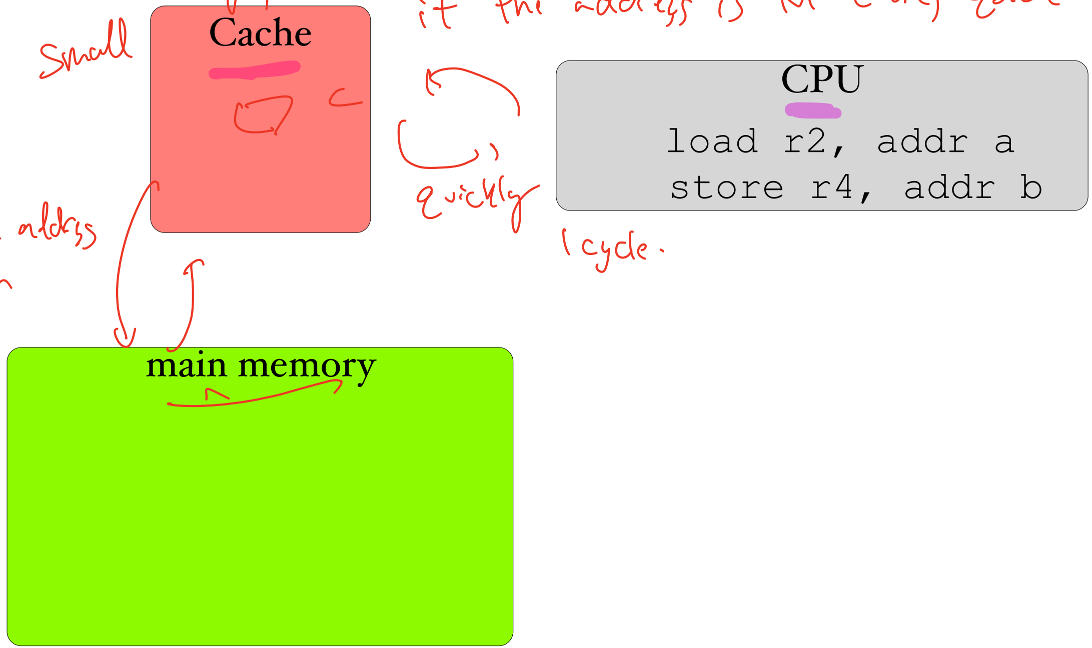
The main idea in this algorithm was the “exchange argument.”

We were able to identify an item (first to finish) that must be part of *some* optimal solution by exchanging this element with one that we can identify in any optimal solution.

Since its easy to identify the item that is first to finish, our algorithm is conversely simple, “greedy.”

caching

cache hit



Cache

CPU

```
load r2, addr a
store r4, addr b
```

main memory

very fast

small

if the address is in cache, quick return

quickly

1 cycle.

if the address is not in cache, it must be retrieved and stored in the cache.

question:

How do we manage the cache ??

(optimally)

question:

How do we manage a fully-associate cache?

When it is full, which element do we replace?

problem statement

input: K cache size, $d_1, d_2, \ldots d_n$: memory access pattern

output: cache schedule that minimizes cache misses.

cache is fully associative, line size I.

problem statement

input: K, the size of the cache
$d_1, d_2, \ldots, d_m$ memory accesses

output: schedule for that cache that minimizes # of cache misses while satisfying requests

cache is fully associative, line size is 1

contrast with reality

contrast with reality

In a real program, we may not know the future memory access patterns.

- Some caches have additional restrictions, like line-size, associativity, etc.

We will consider the easier case described earlier.

Belady eviction rule

Replace the cache entry that is accessed "farthest in the future" (ff)

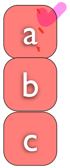
Belady eviction rule

Replace the element in the cache that is accessed "farthest into the future"

example

k=3

cache



a b c d a d e a d b a e c e a ← memory access pattern

d_1 d_2 d_n

Schedule N N N

since d is Not in cache, & cache is full, where do we store d ??

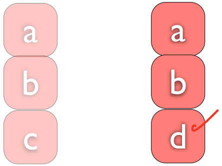
c is the ff address.

So ff says "Replace c with d"

example

we just replaced c with d.

cache



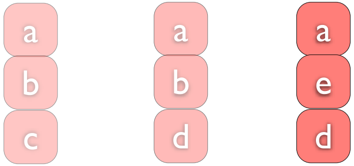
a b c d a d e a d b a e c e a

where do we load e??

ff is b, so we replace b.

example

cache

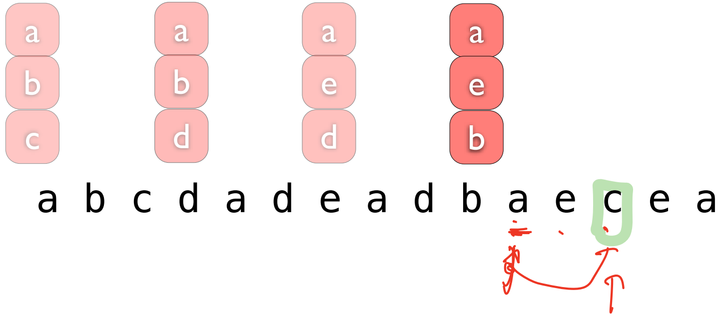


a b c d a d e a d b a e c e a

replace d with b.

example

cache



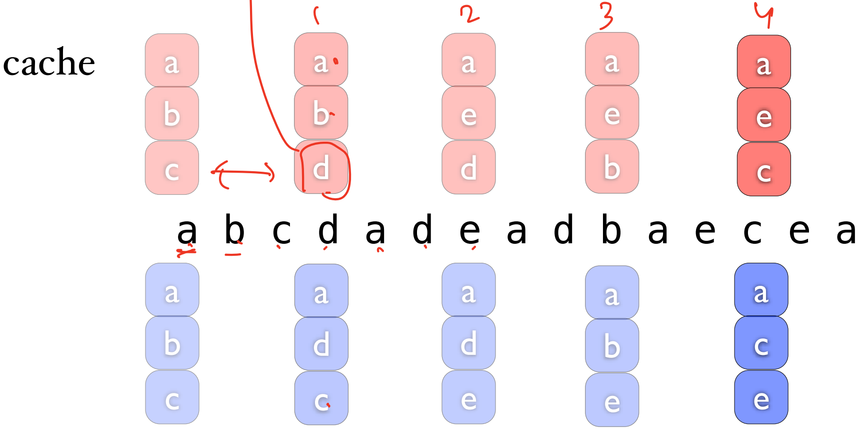
a b c d a d e a d b a e c e a

"evict b for c"

ff is b, so replace b with c.

example

"Replace c with d"



a b c d a d e a d b a e c e a

4 cache misses with this schedule.

this is another optimal 4-miss schedule that does not use FF.

Surprising theorem

Why does ff work??

<u>Theorem</u>: The Belady ff schedule is optimal in terms of minimising the # of cache misses.

Surprising theorem

The schedule S_{ff} produced by the Belady “farthest in the future” eviction rule is optimal.

Lets prove this.

schedule

Schedule for access pattern $d_1, d_2, \ldots, d_n$:

list of instructions, either "NOP" or "replace x with y"

Reduced schedule:

A schedule in which the "Replace x with y" instruction only occurs when y is accessed.

Note: Any schedule can be reduced and incur the same or fewer misses.

schedule

Schedule for access pattern $d_1,d_2,\ldots,d_n$:

A list of instructions for each access that is either “NOP” or “evict x for y”

Reduced schedule:

schedule

Schedule for access pattern $d_1,d_2,\ldots,d_n$:

A list of instructions for each access that is either "NOP" or "evict x for y"

Reduced schedule:

A schedule in which"evict x for y" instruction only occurs when y is accessed.

schedule

Schedule for access pattern $d_1,d_2,\ldots,d_n$:

A list of instructions for each access that is either "NOP" or "evict x for y"

Reduced schedule: (Just in time eviction)

A schedule in which "evict x for y" instruction only occurs when y is accessed.

Note: any schedule can be transformed into a reduced schedule with the same or fewer cache misses.

(Idea: starting at the end, defer "evict...t" until y is read)

Exchange lemma

- Let S be a reduced schedule that agrees with S_{ff} on the first j memory accesses.

There exists a schedule S' that agrees with S_{ff} on $\underline{j+1}$ accesses and incurs the same # of misses a S.

or
fewer

Exchange lemma

Input

Let S be a reduced schedule that agrees with S_{ff} on the first j accesses.

Then there exists a schedule S' that agrees with S_{ff} on the first j+1 accesses and has the same or fewer misses.

This means that schedule S' and S_{ff} perform the same operations on the cache for the first j+1 accesses.

Some optimal schedule.

$\boxed{S^*}$ S_1 S_2 $\underline{S_{ff}}$

0

$\uparrow$ must agree with S_{ff} on 1 access and have same /fewer cache misses as S^*.

$\text{miss}(S^*) \geq \text{miss}(S_1)$

Some optimal
schedule.

S^* S_1

S_{ff}

① Agrees with S_{ff} on the first access.

② $\text{miss}(S) \geq \text{miss}(S_1)$

Some optimal schedule.

S^* S_1 S_2 S_3 S_4 . . . . $S_n = S_{ff}$

Agrees with S_{ff} on the first access.

Agrees with S_{ff} on the first two accesses.

agrees with S_{ff} on all n accesses.

$$\text{miss}(S^*) \geq \text{miss}(S_1) \geq \text{miss}(S_2) \geq \text{miss}(S_3) \ldots \geq \text{miss}(S_n)$$

Because S^* is optimal $\text{miss}(S^*) = \text{miss}(S_n) = \text{miss}(S_{ff})$

Some optimal
schedule.

S^* S_1 S_2 S_3 S_{n-1} S_{ff}

Agrees with S_{ff} on the first access.

Agrees with S_{ff} on the first two accesses.

Agrees with S_{ff} on the first three accesses.

S_{ff} has the same number of cache misses as S^*.

Proof of Lemma

Let S be a reduced sched that agrees with S_{ff} on the first j items. There exists a reduced sched **S'** that agrees with S_{ff} on the first j+1 items and has the same or fewer #misses as S.

Proof of Lemma

Let S be a reduced sched that agrees with S_{ff} on the first j items.
There exists a reduced sched **S'** that agrees with S_{ff} on the first j+1 items and has the same or fewer #misses as S.

At time j, both S and S_{ff} have the same state.

Let d be the element accessed at time j+1.

Proof of lemma

State of the cache after J operations under the two schedules.

easy case 1

easy case 2

Proof of lemma

State of the cache after J operations under the two schedules.

easy case 1 d is in the cache.

easy case 2

Proof of lemma

State of the cache after J operations under the two schedules.

easy case 1

d is in the cache.

Both S and S_{ff} agree since both do NOPs at j+1.

easy case 2

Proof of lemma

State of the cache after J operations under the two schedules.

easy case 1 d is in the cache.

Both S and S_{ff} agree since both do NOPs at j+1.

easy case 2 d is not in the cache, but both “evict e for d.”

Proof of lemma

State of the cache after J operations under the two schedules.

easy case 1 d is in the cache.

Both S and S_{ff} agree since both do NOPs at j+1.

easy case 2 d is not in the cache, but both "evict e for d."

Both S and S_{ff} agree at j+1.

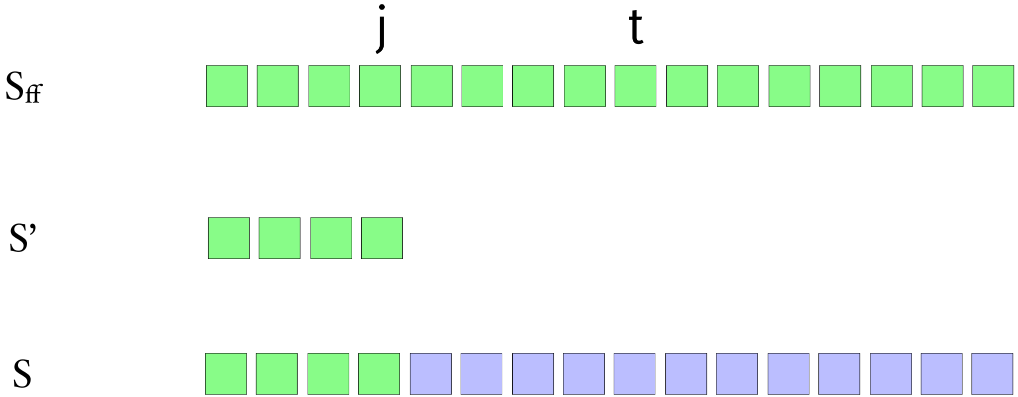
Proof of lemma

case 3

Proof of lemma

case 3 S evicts “e for d”, and S_{ff} evicts “e for f”

Timeline



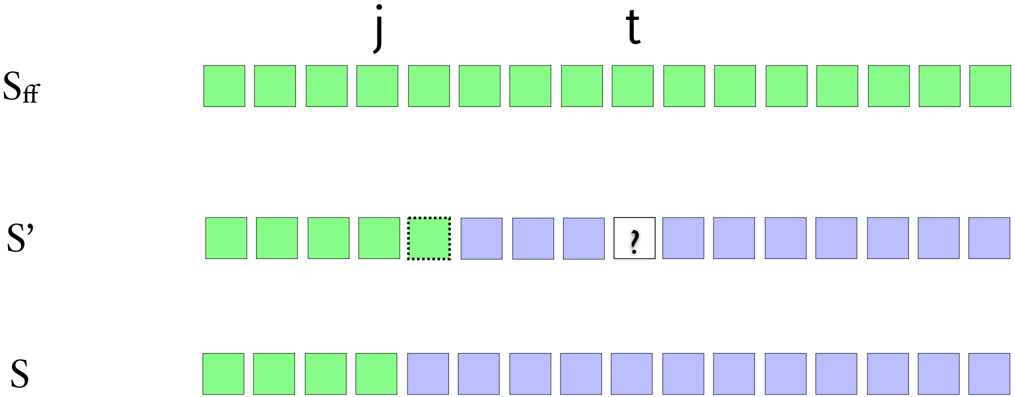
j t

S_{ff}

S’

S

Timeline



Copy j+1 from S_{ff}. Then copy from S until t (the first time that either e or f are accessed). Then copy from S until the end.

Proof of lemma

Let t be the first access that either e or f are accessed.

What if t=e:

Proof of lemma

what if t=e ?

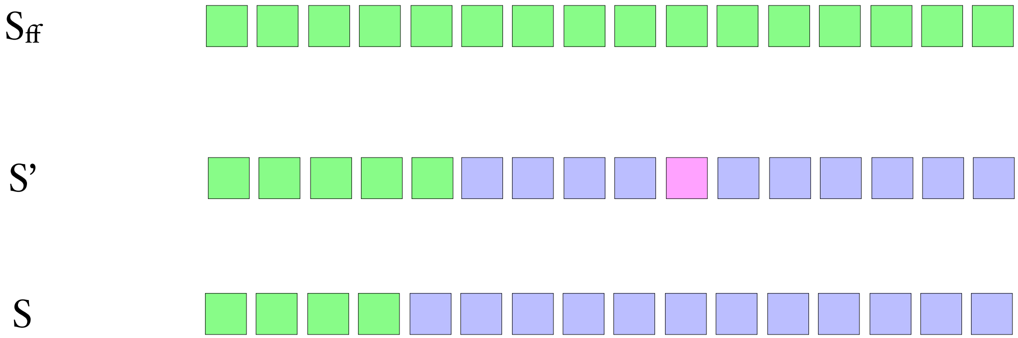
Proof of lemma

what if t=f ?

Proof of lemma

what if t is neither e nor f ?

What have we shown



Let S be a reduced sched that agrees with S_{ff} on the first j items.
There exists a reduced sched **S’** that agrees with S_{ff} on the first j+1 items and has the same or fewer #misses as S.

Let S be a reduced sched that agrees with S_{ff} on the first j items.
There exists a reduced sched **S’** that agrees with S_{ff} on the first j+1 items and has the same or fewer #misses as S.

S^*

S_{ff}

Recap

The greedy algorithm is quite simple.

But the analysis for why the solution works is more subtle and complicated.

In this case, we had to apply the exchange lemma multiple times to prove optimality.