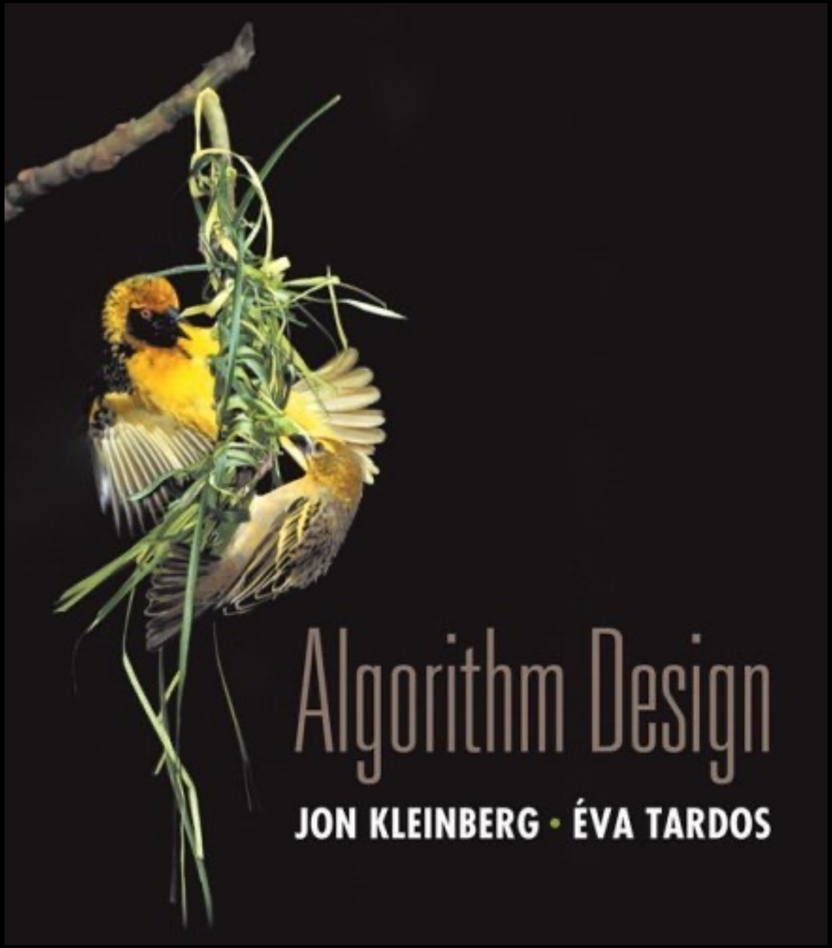


Chapter 4

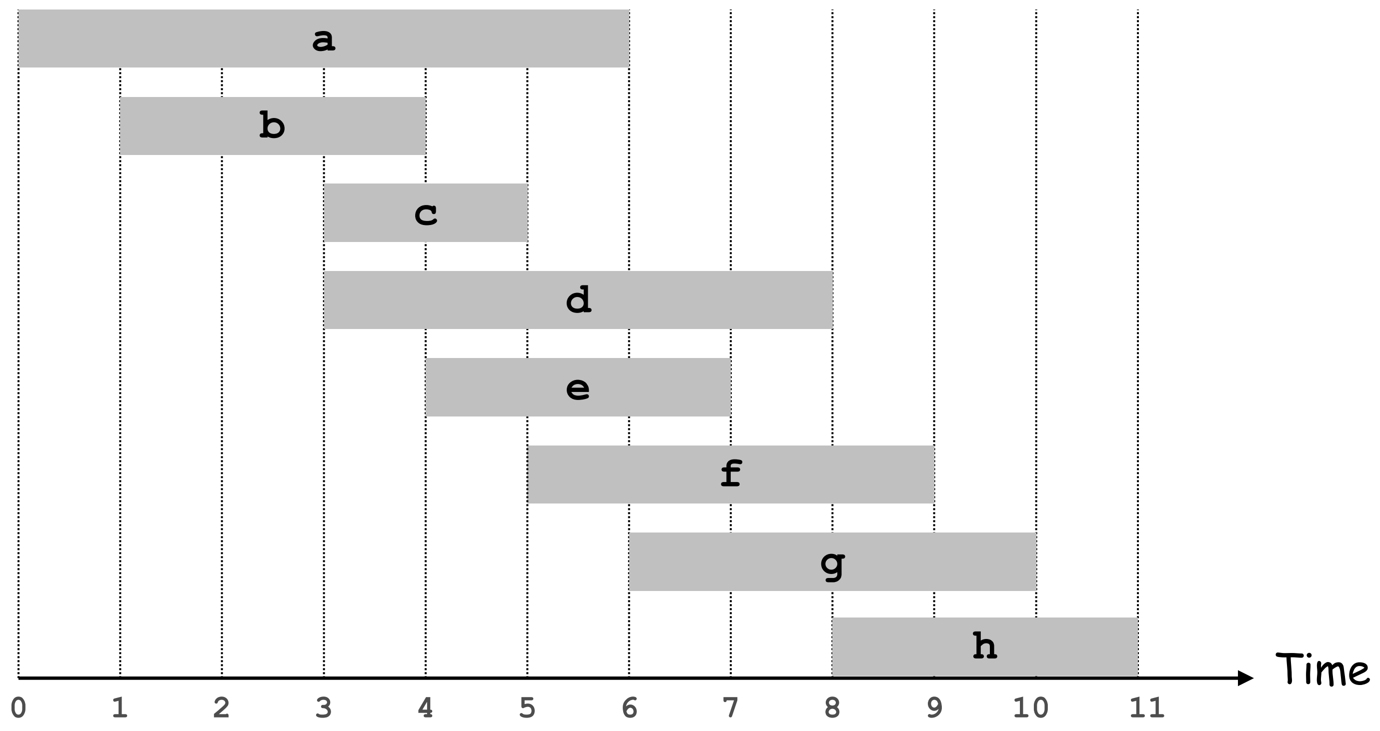
Greedy
Algorithms

4.1 Interval Scheduling

Interval Scheduling

Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Interval Scheduling: Greedy Algorithms

strategy

Greedy template. Consider jobs in some natural order.
Take each job provided it's compatible with the ones already taken.

- [Earliest start time] Consider jobs in ascending order of s_j.

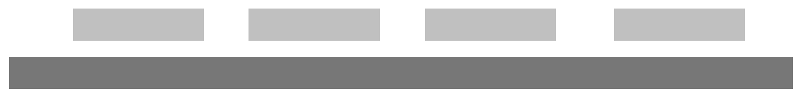
- [Earliest finish time] Consider jobs in ascending order of f_j.

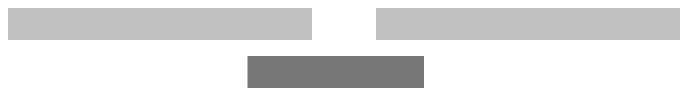
- [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.

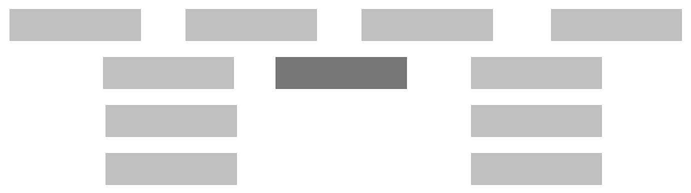
- [Fewest conflicts] For each job j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some natural order.
Take each job provided it's compatible with the ones already taken.


counterexample for earliest start time


counterexample for shortest interval


counterexample for fewest conflicts

Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that $f_1 \le f_2 \le ... \le f_n$.

   set of jobs selected
A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Implementation. O(n log n).

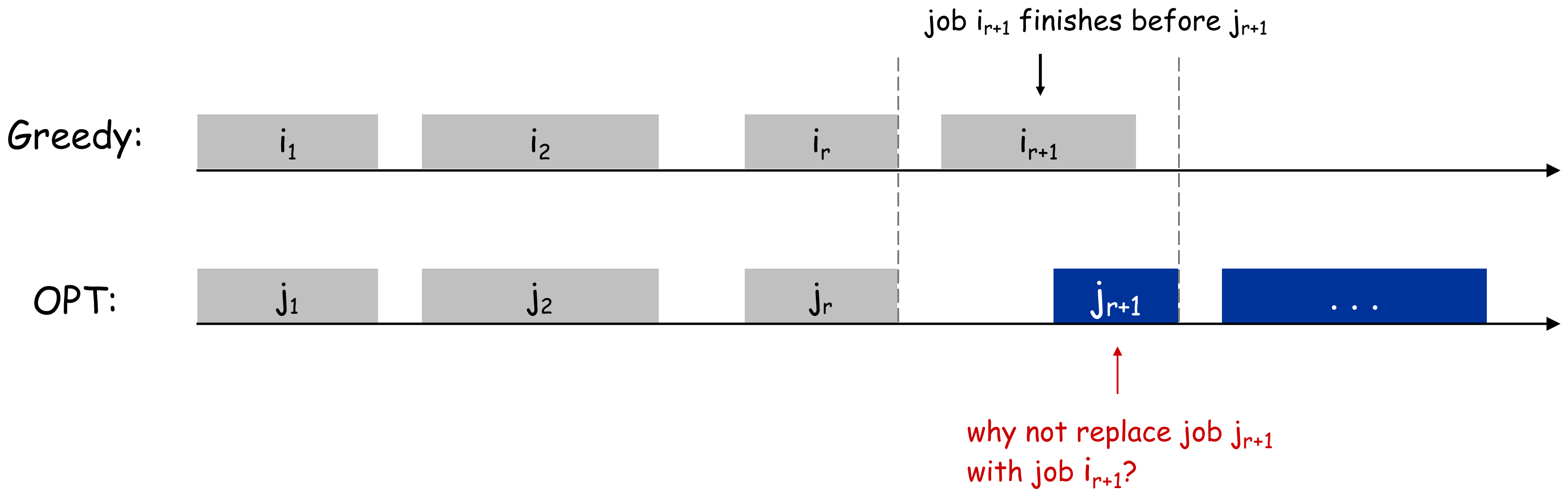
- Remember job j* that was added last to A.
- Job j is compatible with A if $s_j \ge f_{j^*}$.

Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- 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 the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



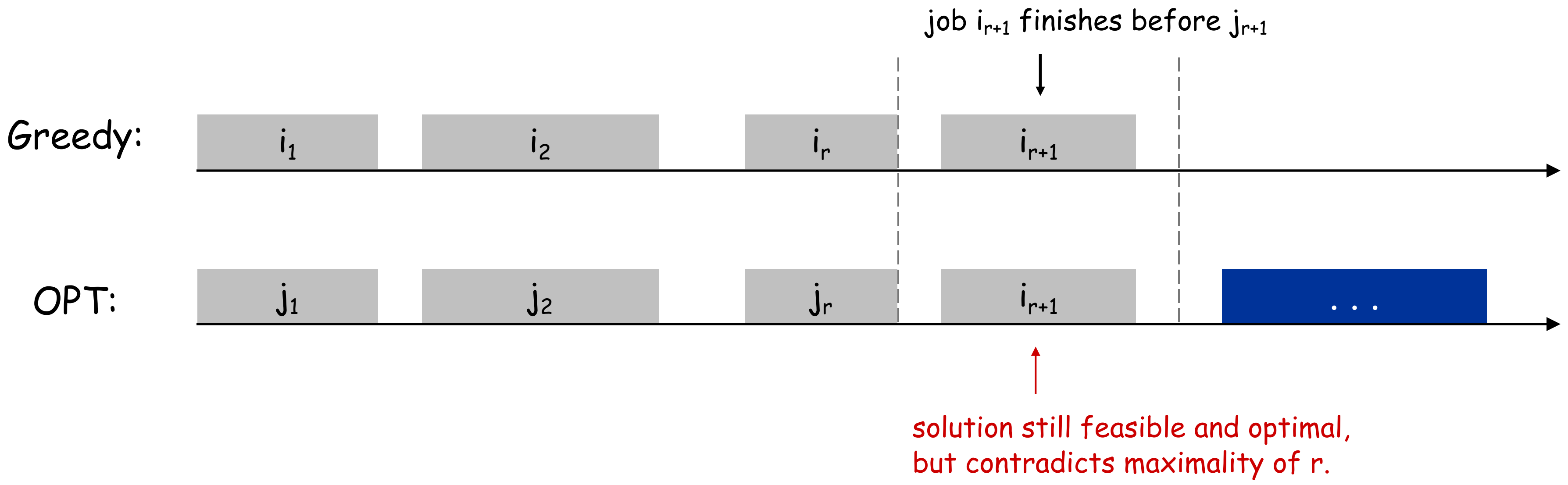
job i_{r+1} finishes before j_{r+1}
Greedy:
i_1
i_2
i_r
i_{r+1}
OPT:
j_1
j_2
j_r
j_{r+1}
. . .
why not replace job j_{r+1} with job i_{r+1}?

Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- 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 the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



job i_{r+1} finishes before j_{r+1}
Greedy:
i_1
i_2
i_r
i_{r+1}
OPT:
j_1
j_2
j_r
i_{r+1}
. . .
solution still feasible and optimal, but contradicts maximality of r.

Interval Scheduling: Analysis

Proof. 4.3 The greedy algorithm returns an optimal set *A*.

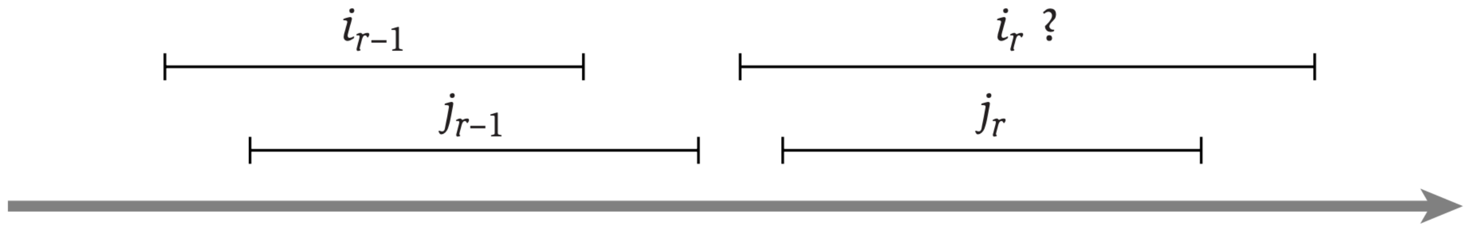
Textbook, page 120-121

Greedy set A = {i_1, i_2, …, i_k}, optimal set O = {j_1, j_2, …, j_m}

Goal: for each k ≥ r ≥ 1, we have $f(i_r) \le f(j_r)$

Induction:

1. Ture for r = 1
2. Let r > 1, assume $f(i_{r-1}) \le f(j_{r-1})$, can we have $f(i_r) \le f(j_r)$?



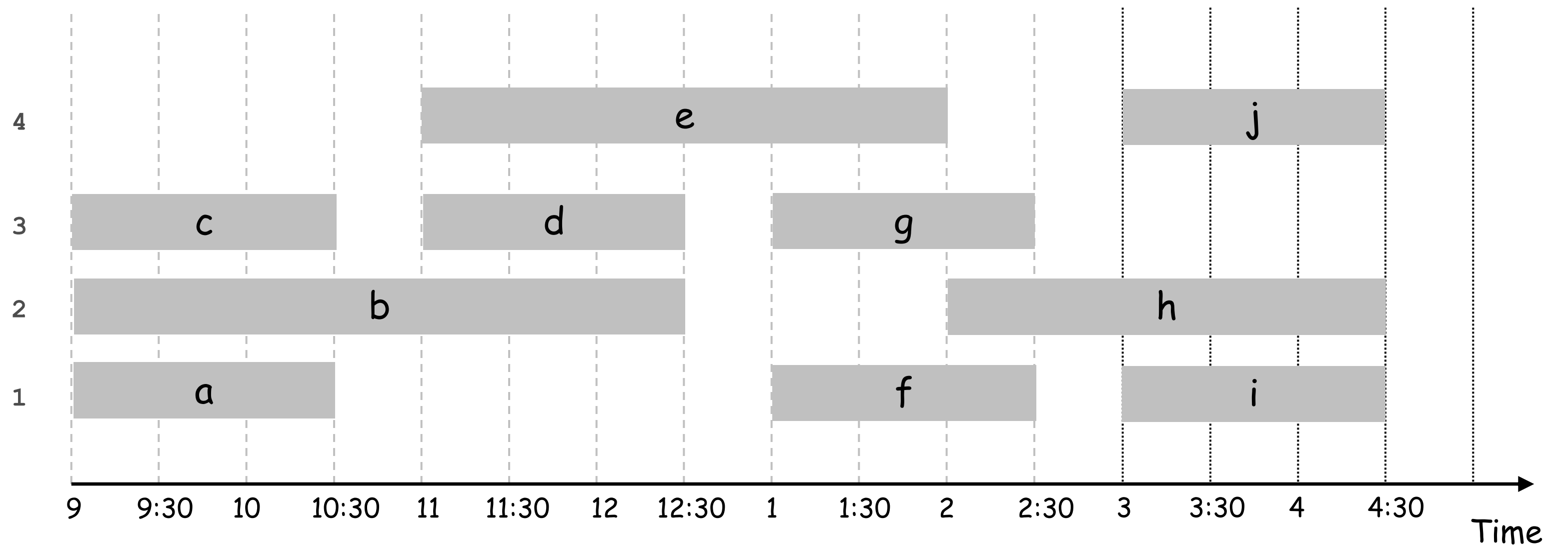
4.2 Interval Partitioning

Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses 4 classrooms to schedule 10 lectures.

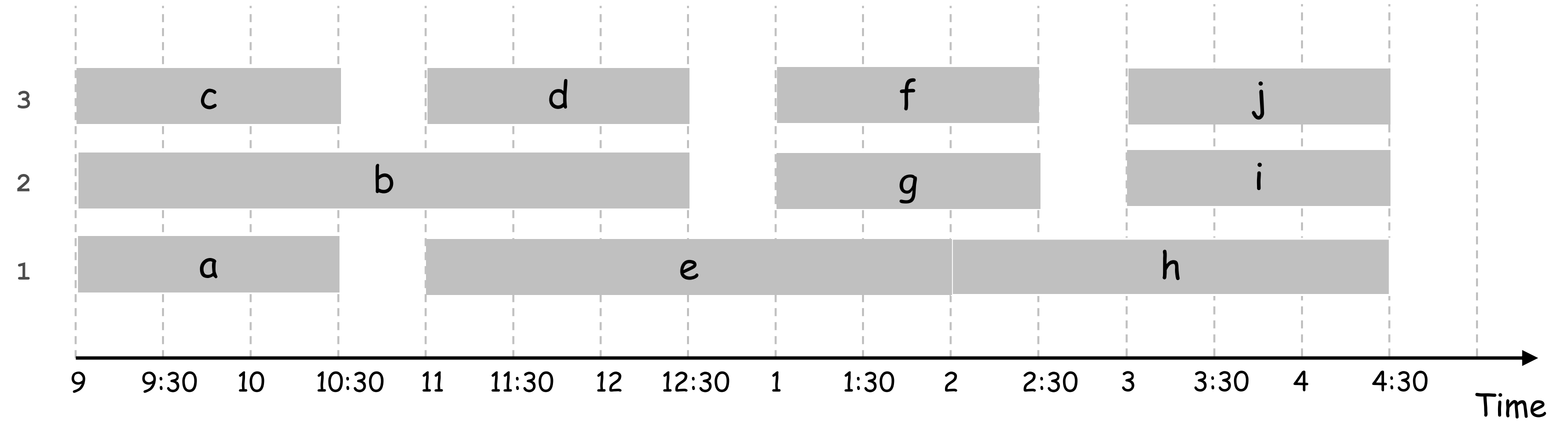


Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses only 3.



Interval Partitioning: Lower Bound on Optimal Solution

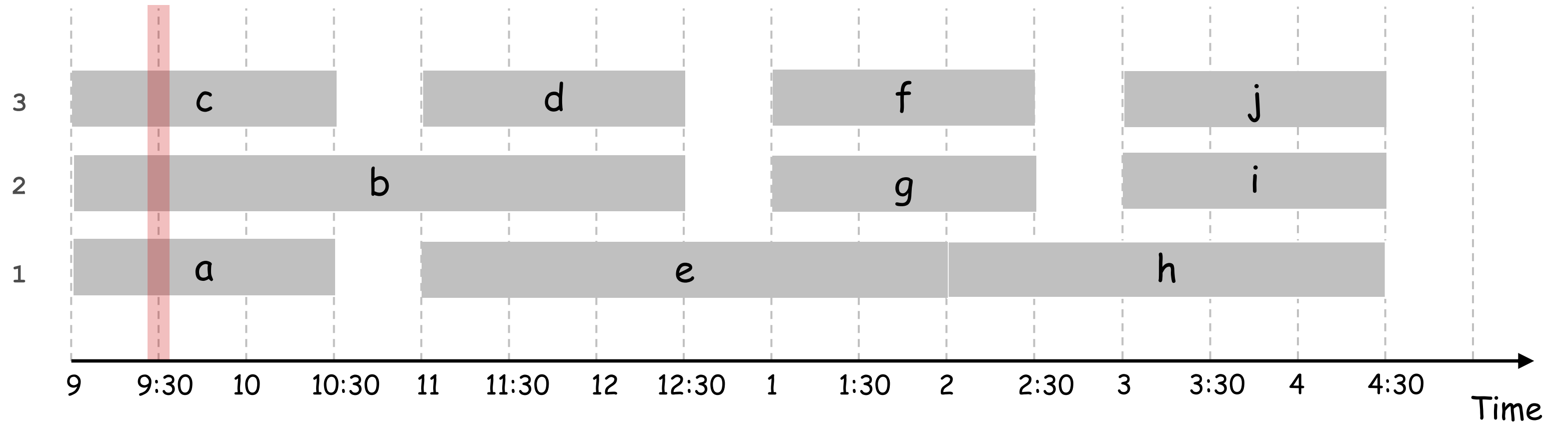
Def. The depth of a set of open intervals is the maximum number that contain any given time.

Key observation. Number of classrooms needed $\geq$ depth.

Ex: Depth of schedule below = 3 $\Rightarrow$ schedule below is optimal.

a, b, c all contain 9:30

Q. Does there always exist a schedule equal to depth of intervals?



Interval Partitioning: Greedy Algorithm

Greedy algorithm. Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that $s_1 \le s_2 \le ... \le s_n$.
d ← 0   ← number of allocated classrooms

for j = 1 to n {
   if (lecture j is compatible with some classroom k)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

Implementation. O(n log n). (why?)

- For each classroom k, maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.

Interval Partitioning: Greedy Analysis

Observation. Greedy algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Greedy algorithm is optimal.
Pf.

- Let d = number of classrooms that the greedy algorithm allocates.
- Classroom d is opened because we needed to schedule a job, say j, that is incompatible with all d-1 other classrooms.
- These d jobs each end after s_j.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j.
- Thus, we have d lectures overlapping at time $s_j + \varepsilon$.
- Key observation $\Rightarrow$ all schedules use $\geq$ d classrooms. ▪
- (d is the depth)

Interval Partitioning: Greedy Algorithm (alternative)

Textbook page 124, alternative

```
Sort the intervals by their start times, breaking ties arbitrarily
Let $I_1, I_2, \ldots, I_n$ denote the intervals in this order
For $j = 1, 2, 3, \ldots, n$
  For each interval $I_i$ that precedes $I_j$ in sorted order and overlaps it
     Exclude the label of $I_i$ from consideration for $I_j$
  Endfor
  If there is any label from $\{1, 2, \ldots, d\}$ that has not been excluded then
    Assign a nonexcluded label to $I_j$
  Else
    Leave $I_j$ unlabeled
  Endif
Endfor
```

Q: Why using d labels is sufficient?

4.3 Scheduling to Minimize Lateness

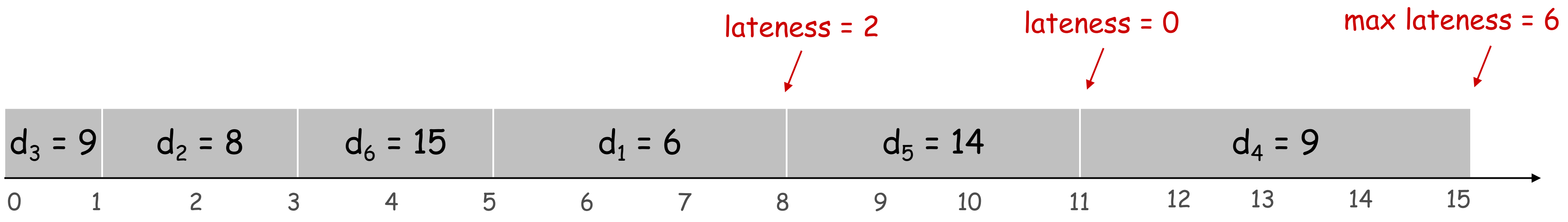
Scheduling to Minimizing Lateness

Minimizing lateness problem.

- Single resource processes one job at a time.
- Job j requires t_j units of processing time and is due at time d_j.
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
- Lateness: $\ell_j = \max\{0, f_j - d_j\}$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

Ex:

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



Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.
- [Earliest deadline first] Consider jobs in ascending order of deadline d_j.
- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

	1	2
t_j	1	10
d_j	100	10

counterexample

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

	1	2
t_j	1	10
d_j	2	10

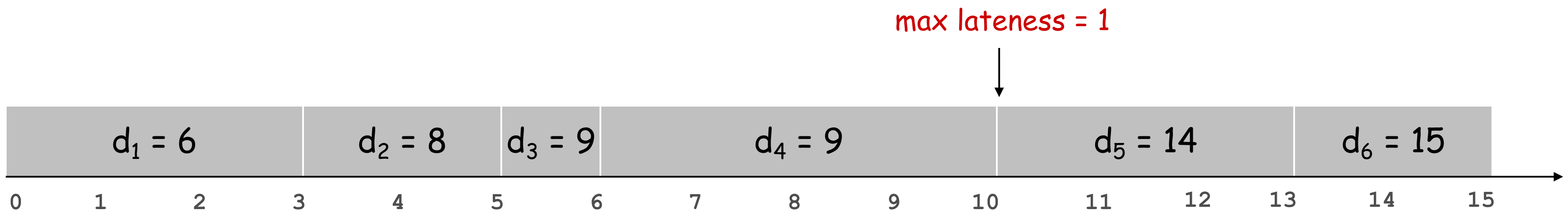
counterexample

Minimizing Lateness: Greedy Algorithm

Greedy algorithm. Earliest deadline first.

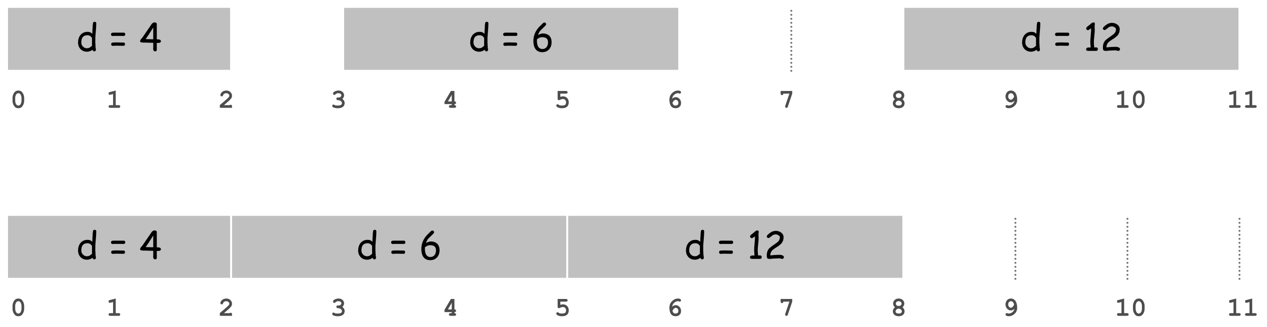
```
Sort n jobs by deadline so that $d_1 \le d_2 \le \ldots \le d_n$

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + $t_j$]
   $s_j$ ← t, $f_j$ ← t + $t_j$
   t ← t + $t_j$
output intervals [$s_j$, $f_j$]
```



Minimizing Lateness: No Idle Time

Observation. There exists an optimal schedule with no idle time.

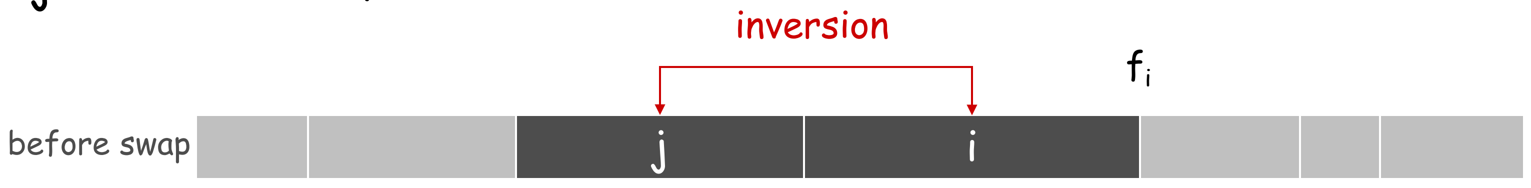


Observation. The greedy schedule has no idle time.

Minimizing Lateness: Inversions

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

i.e., $d_i < d_j$



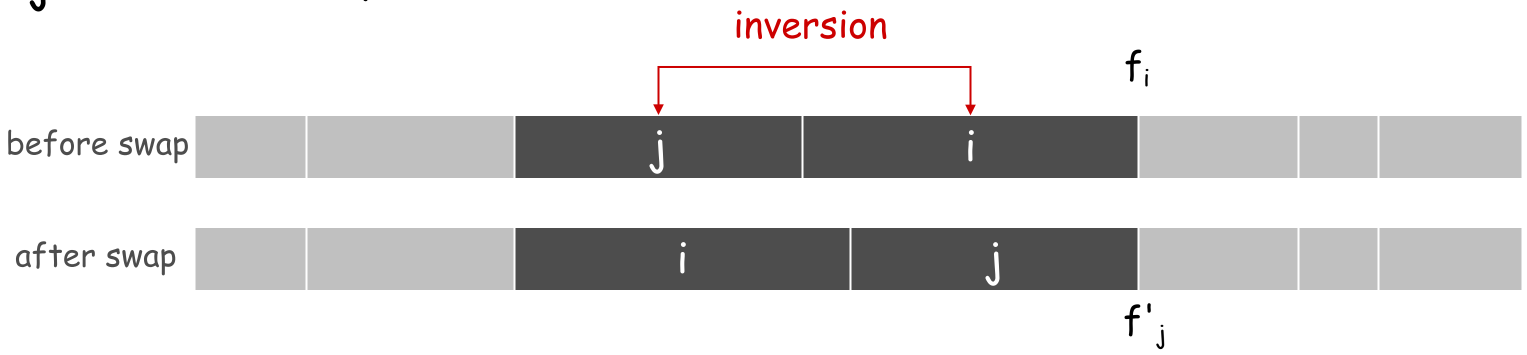
[as before, we assume jobs are numbered so that $d_1 \le d_2 \le \ldots \le d_n$]

Observation. Greedy schedule has no inversions.

Observation. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

Minimizing Lateness: Inversions

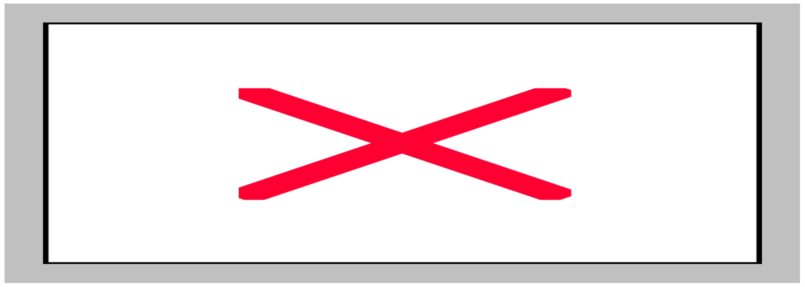
Def. Given a schedule S, an inversion is a pair of jobs i and j such that: i < j but j scheduled before i.



Claim. Swapping two consecutive, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

Pf. Let ℓ be the lateness before the swap, and let ℓ' be it afterwards.

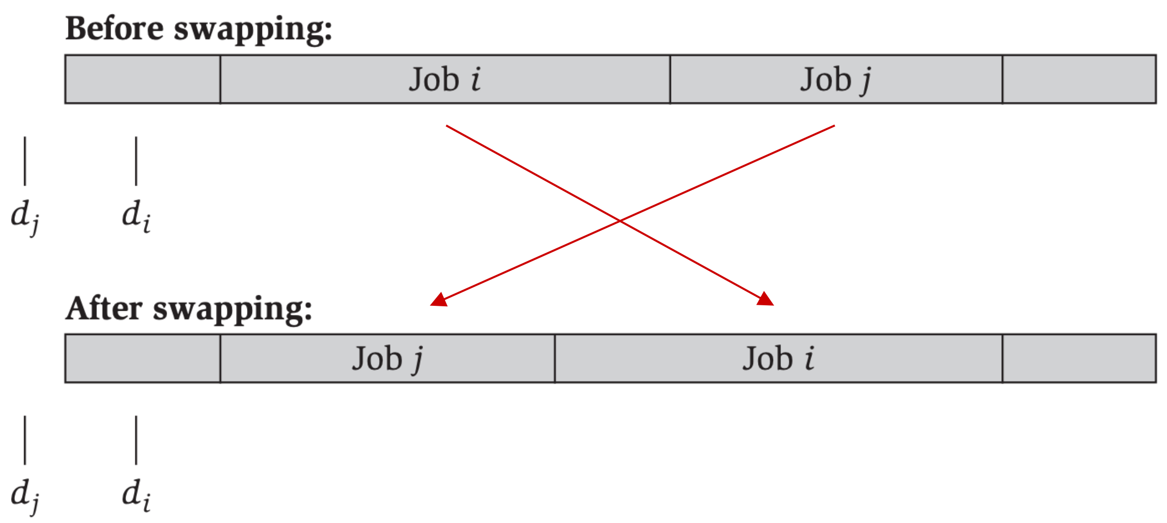
- $\ell'_k = \ell_k$ for all $k \neq i, j$
- $\ell'_i \leq \ell_i$
- If job j is late:
- $\ell'_j < \ell_j$



Minimizing Lateness: Inversions

Proof of 4.9 (c) The new swapped schedule has a maximum lateness no larger than that of the optimal schedule.

Textbook, page 129-130



Original schedule: $\mathcal{O}$

Swapped schedule: $\bar{\mathcal{O}}$

Assumption: $d_i > d_j$

Obviously for j, $L_j = f(j) - d_j$ => $\overset{\downarrow}{\bar{L}_j} = \overset{\downarrow}{\bar{f}}(j) - d_j$

What about i? $\bar{L}_i = \bar{f}(i) - d_i = f(j) - d_i < f(j) - d_j = L$

Minimizing Lateness: Analysis of Greedy Algorithm

Theorem. Greedy schedule S is optimal.

Pf. Define S* to be an optimal schedule that has the fewest number of inversions, and let's see what happens.

- Can assume S* has no idle time.
- If S* has no inversions, then S = S*.
- If S* has an inversion, let i-j be an adjacent inversion.
  - swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions
  - this contradicts definition of S* ■

Greedy Analysis Strategies

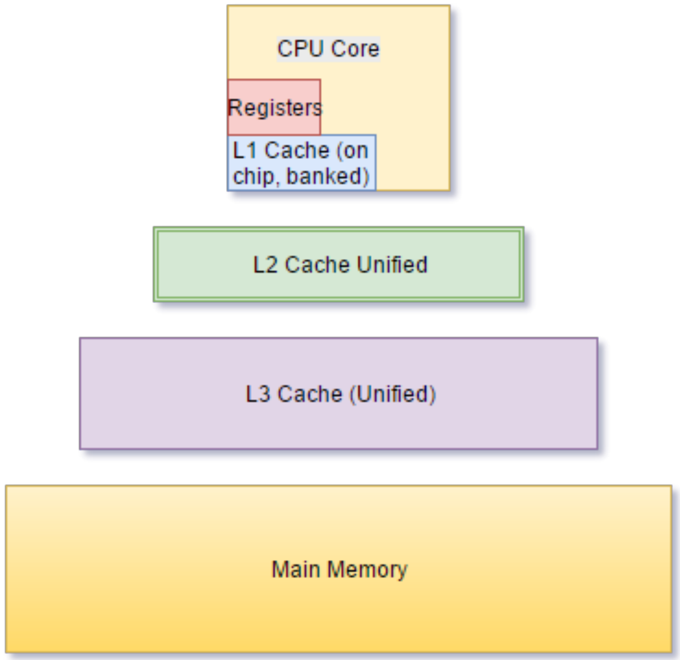
Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

Structural. Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

Other greedy algorithms. Kruskal, Prim, Dijkstra, Huffman, …

4.4 Optimal Caching



Optimal Offline Caching

最优离线缓存

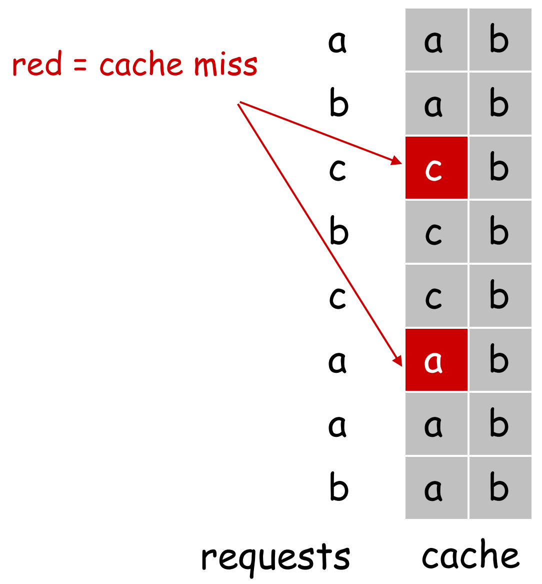
Caching.

- Cache with capacity to store k items.
- Sequence of m item requests $d_1, d_2, \ldots, d_m$.
- Cache **hit**: item already in cache when requested.
- Cache **miss**: item not already in cache when requested: must bring requested item into cache, and *evict* some existing item, if full.

Goal. Eviction schedule that minimizes number of cache misses.

Ex: k = 2, initial cache = ab,
requests: a, b, c, b, c, a, a, b.

Optimal eviction schedule: 2 cache misses.



requests	cache	
a	a	b
b	a	b
c	🟥 c	b
b	c	b
c	c	b
a	🟥 a	b
a	a	b
b	a	b

Optimal Offline Caching: Farthest-In-Future

Farthest-in-future. Evict item in the cache that is not requested until farthest in the future. 每次要弹出一个数据的时候，选择在未来最远的requested的弹出

current cache: a b c d e f

future queries: g a b c e d a b b a c d e a **f** a d e f g h . . .

↑ cache miss (g)

↑ eject this one (f)

Theorem. [Bellady, 1960s] FF is optimal eviction schedule.

Pf. Algorithm and theorem are intuitive; proof is subtle.

Term: online vs. offline caching

Reduced Eviction Schedules

Reduced: 简化的

Def. A reduced schedule is a schedule that only inserts an item into the cache in a step in which that item is requested.

Intuition. Can transform an unreduced schedule into a reduced one with no more cache misses. Unreduced/nonreduced: 繁琐的

a	a	b	c
a	a	x	c
c	a	d	c
d	a	d	b
a	a	c	b
b	a	x	b
c	a	c	b
a	a	b	c
a	a	b	c

an unreduced schedule

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

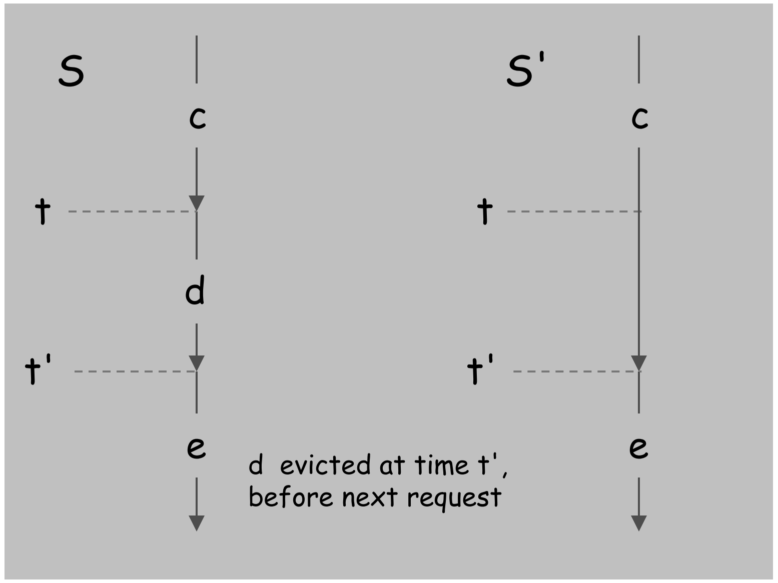
a reduced schedule

Reduced Eviction Schedules

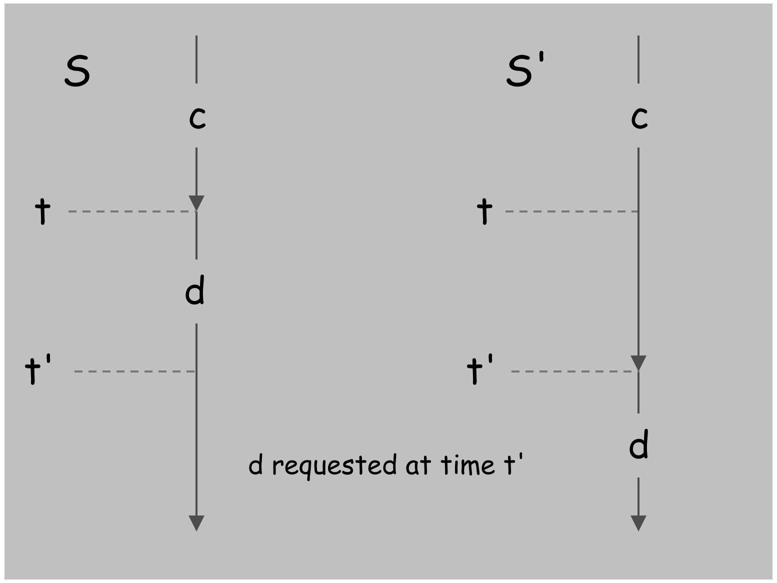
Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more cache misses.

Pf. (by induction on number of unreduced items) ← doesn't enter cache at requested time

- Suppose S brings d into the cache at time t, without a request.
- Let c be the item S evicts when it brings d into the cache.
- Case 1: d evicted at time t', before next request for d.
- Case 2: d requested at time t' before d is evicted. ▪



Case 1



Case 2

Farthest-In-Future: Analysis

Theorem. FF is optimal eviction algorithm.
Pf. (by induction on number of requests j)

Invariant: There exists an optimal reduced schedule S that makes the same eviction schedule as S_{FF} through the first j+1 requests.

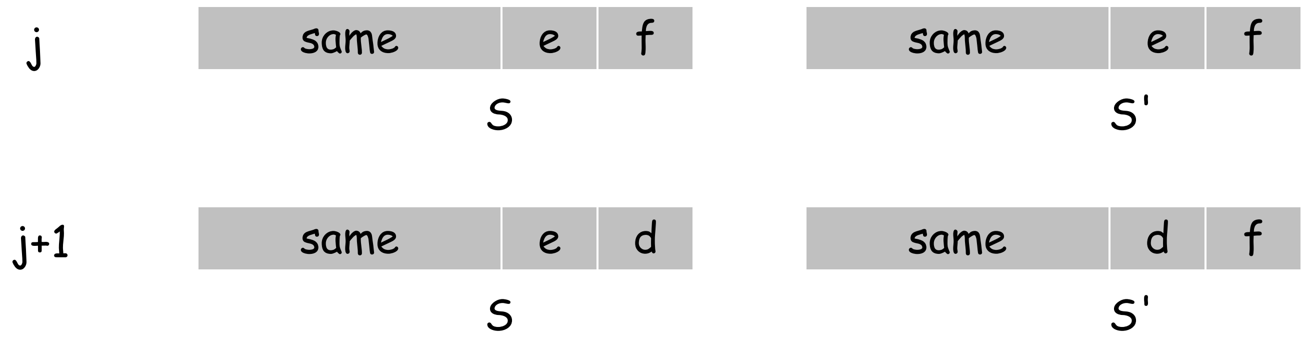
Let S be reduced schedule that satisfies invariant through j requests.
We produce S' that satisfies invariant after j+1 requests.

- Consider (j+1)st request $d = d_{j+1}$.
- Since S and S_{FF} have agreed up until now, they have the same cache contents before request j+1.
- Case 1: (d is already in the cache). S' = S satisfies invariant.
- Case 2: (d is not in the cache and S and S_{FF} evict the same element). S' = S satisfies invariant.

Farthest-In-Future: Analysis

Pf. (continued)

- Case 3: (d is not in the cache; S_{FF} evicts e; S evicts f $\neq$ e).
  - begin construction of S' from S by evicting e instead of f

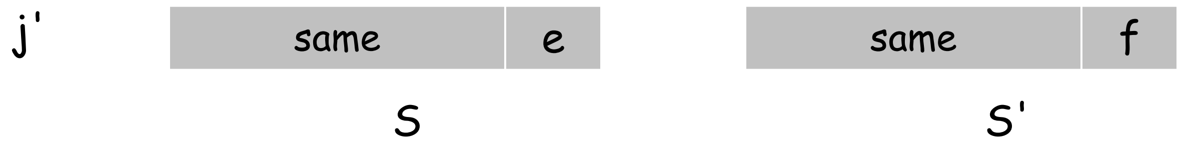


  - now S' agrees with S_{FF} on first j+1 requests; we show that having element f in cache is no worse than having element e

Farthest-In-Future: *Analysis*

Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.

$\uparrow$ must involve e or f (or both)



- Case 3a: g = e. Can't happen with Farthest-In-Future since there must be a request for f before e.

- Case 3b: g = f. Element f can't be in cache of S, so let e' be the element that S evicts.
  - if e' = e, S' accesses f from cache; now S and S' have same cache
  - if e' $\neq$ e, S' evicts e' and brings e into the cache; now S and S' have the same cache

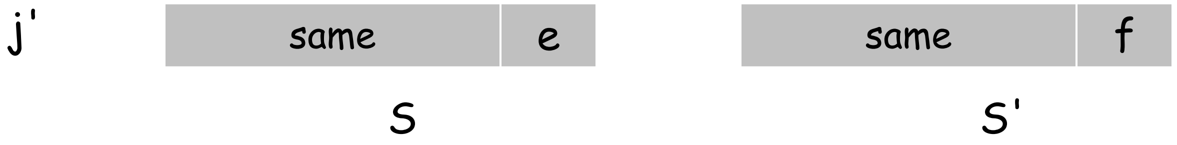
$\uparrow$

Note: S' is no longer reduced, but can be transformed into a reduced schedule that agrees with S_{FF} through step j+1

Farthest-In-Future: *Analysis*

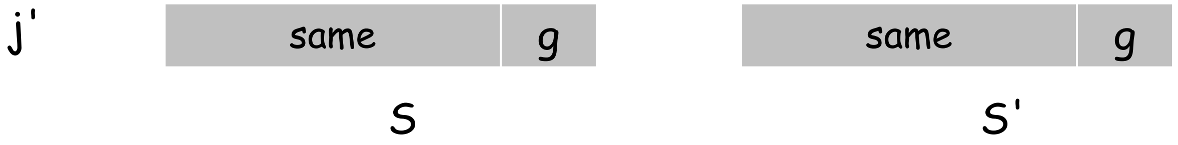
Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.

must involve e or f (or both)



otherwise S' would take the same action

- Case 3c: $g \neq e, f$. S must evict e.
  Make S' evict f; now S and S' have the same cache. ∎



Proof of Farthest-In-Future rephrased

Reference post: https://blog.csdn.net/qq_36665989/article/details/120616786

Goal of proof: considering a reduced schedule S, and a schedule produced by FF algorithm S_{FF}, satisfying that their first j steps are the same. We can always transform S into a new schedule S', such that S' and S_{FF} have the same first j+1 steps, and that S' incurs no more misses than S.

Step	S	S_{FF}	S'
j	{e, f, …}	{e, f, …}	{e, f, …}
j + 1	{e, d, …}	{d, f, …}	{d, f, …}
j + 2	{…}	{…}	{…}
…			

Difference at step j+1

S' mimics S in future steps

Proof of Farthest-In-Future rephrased

How do we let S′ mimic S as much as possible?

- Because S′ and S become different since step j+1, we need to first adjust the elements in S′ to make them the same as S as soon as possible
- If S′ and S become identical at some step, then for all the future steps we simply let S′ copy S.

Step	S	S_{FF}	S′
j	{e, f, …}	{e, f, …}	{e, f, …}
j + 1	{e, d, …}	{d, f, …}	{d, f, …}

If the incoming data after step j+1 are not e or f, then S and S′ take the same actions

So we focus on the first time after step j+1 that S and S′ take different actions.

Q: When the first time S and S′ take different actions after step j+1, can the data be e? Why or why not?

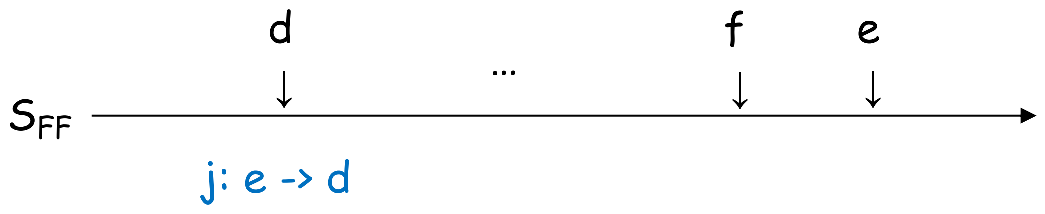
Reference post: https://blog.csdn.net/qq_36665989/article/details/120616786

Proof of Farthest-In-Future rephrased

Q: When the first time S and S′ take different actions after step j+1, can the data be e? Why or why not?

Step	S	S_{FF}	S′
j	{e, f, …}	{e, f, …}	{e, f, …}
j + 1	{e, d, …}	{d, f, …}	{d, f, …}

No! Because S_{FF} replaces e instead of f at step j, so e is farther than f in future events.
In another word, if the input is e after step j+1, then f must come before it.



So, we do not need to consider e for the first time S and S′ take different actions after step j+1.

Proof of Farthest-In-Future rephrased

Analysis: What data can it be for the first time after step j+1 that S and S' take different actions?

Step	S	S_{FF}	S'
j	{e, f, …}	{e, f, …}	{e, f, …}
j + 1	{e, d, others}	{d, f, others}	{d, f, others}

Case 1: input is f.

1 a: S: e -> f.
S becomes {d, f, …}. S' hit f. S' is unchanged
Thus, S and S' become the same

S' is unreduced, but can be converted to reduced. (slide 32)

1 b: S: x -> f, and x is some non-"e" element in "others".
S becomes {e, f, …}. S' hit f. S' is unchanged.
But in order to make S' identical to S, we can replace the same element x in S' with f.
So S' also becomes {e, f, …}, i.e., same as S.

Proof of Farthest-In-Future rephrased

Analysis: What data can it be for the first time after step j+1 that S and S' take different actions?

Step	S	S_{FF}	S'
j	{e, f, …}	{e, f, …}	{e, f, …}
j + 1	{e, d, others}	{d, f, others}	{d, f, others}

Case 2: input is neither e nor f, say g

For S, if it replaces some non-"e" element, then for S' we can do the same, then they take the same actions (not the case under discussion)

So S can only replace e with g. S becomes {g, d, others}
Then S' can replace f with g, and becomes {g, d, others}, too
S and S' are the same now.

Now, we have proved that no matter what input data is at step j+1, S and S' can always become the same after taking one-time different actions.

Proof of Farthest-In-Future rephrased

Recall the goal of proof: considering a reduced schedule S, and a schedule produced by FF algorithm S_{FF}, satisfying that their first j steps are the same. We can always transform S into a new schedule S', such that S' and S_{FF} have the same first j+1 steps, and that S' incurs no more misses than S.

Step	S	S_{FF}	S'
j	{e, f, …}	{e, f, …}	{e, f, …}
j + 1	{e, d, …}	{d, f, …}	{d, f, …}

We have proved: if we can find an optimal schedule S_0 that shares the same first j steps with S_{FF}, then we can always transform it to a new schedule S' that shares the same first j+1 steps with S_{FF}.

It is easy to find S_0 at step 0.

Induction done. S_{FF} is optimal.

Caching Perspective

Online vs. offline algorithms.

- Offline: full sequence of requests is known a priori.
- Online (reality): requests are not known in advance.
- Caching is among most fundamental online problems in CS.

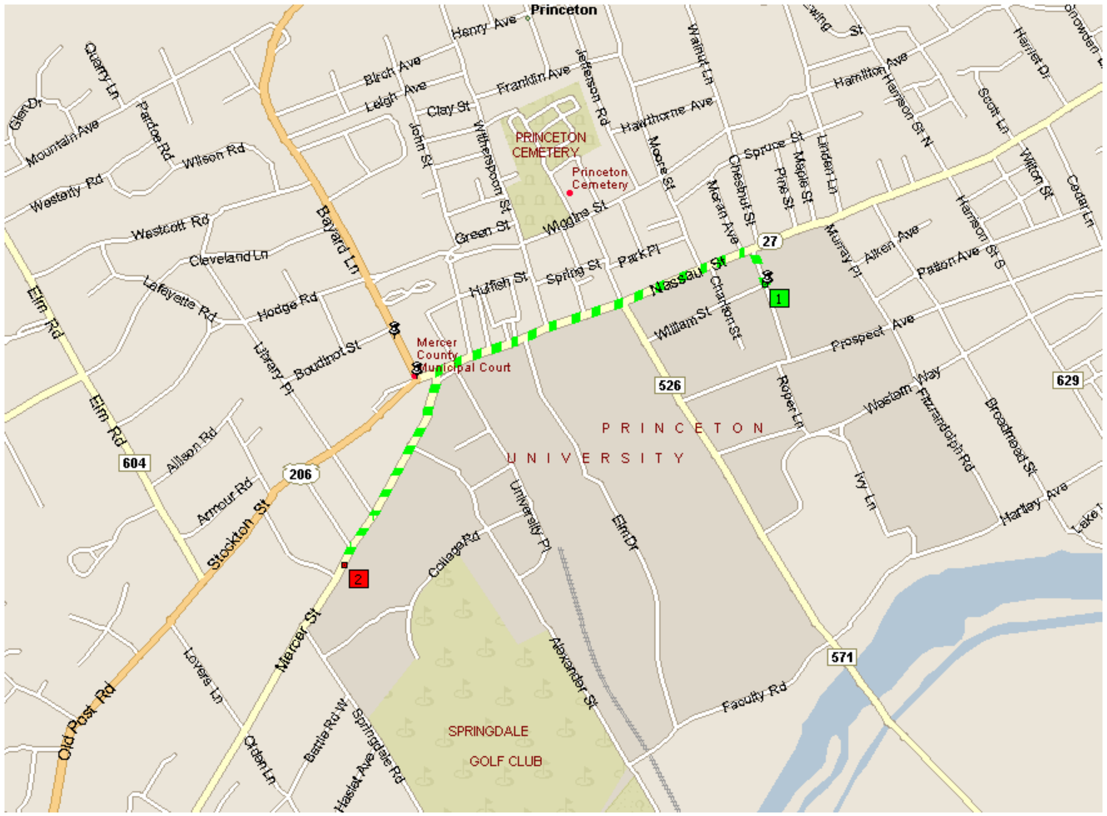
LIFO. Evict item brought in most recently.
LRU. Evict item whose most recent access was earliest.

↑ FF with direction of time reversed!

Theorem. FF is optimal offline eviction algorithm.

- Provides basis for understanding and analyzing online algorithms.
- LRU is k-competitive. [Section 13.8]
- LIFO is arbitrarily bad.

4.5 Shortest Paths in a Graph



shortest path from Princeton CS department to Einstein's house

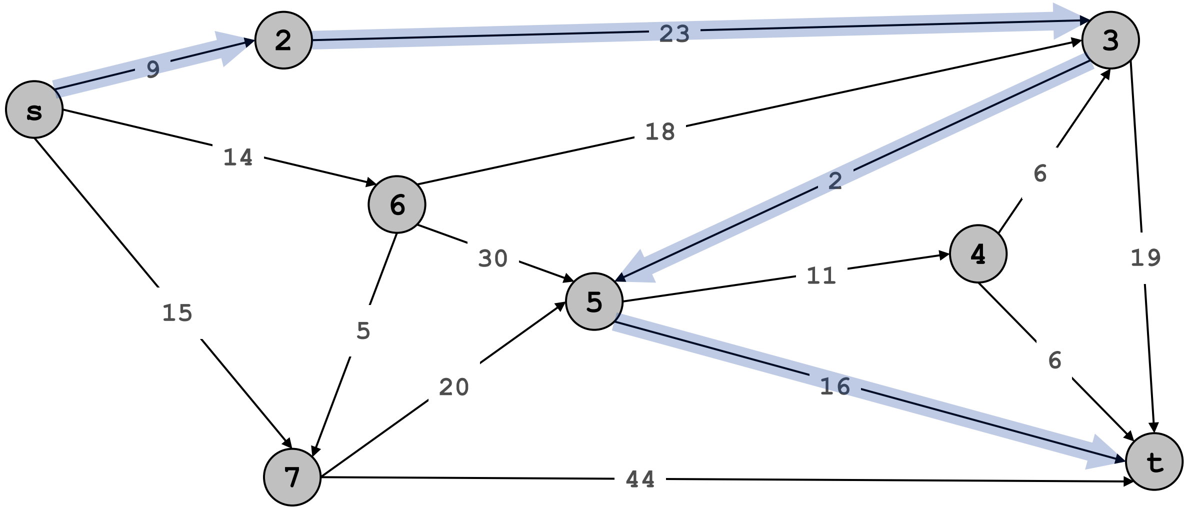
Shortest Path Problem

Shortest path network.

- Directed graph $G = (V, E)$.
- Source s, destination t.
- Length ℓ_e = length of edge e.

Shortest path problem: find shortest directed path from s to t.

cost of path = sum of edge costs in path



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 50.

Dijkstra's Algorithm

Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

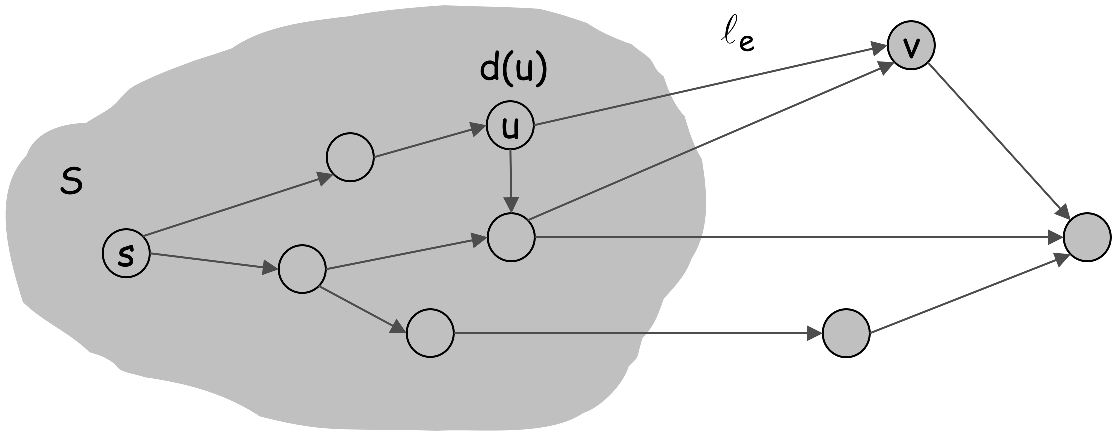
d(u) is a value to be determined

$$\pi(v) = \min_{e=(u,v):u\in S} d(u) + \ell_e$$

-

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).



Dijkstra's Algorithm

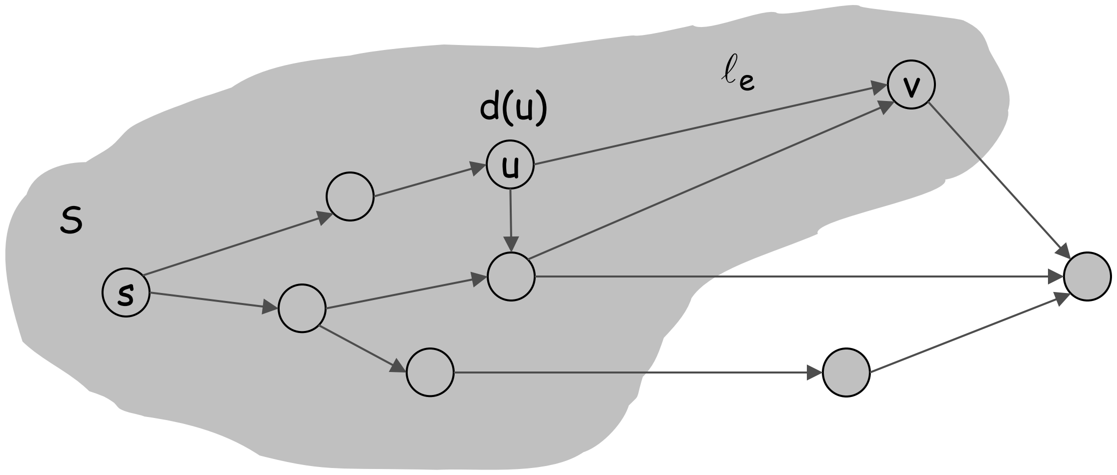
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v):u\in S} d(u) + \ell_e$$

shortest path to some u in explored part, followed by a single edge (u, v)

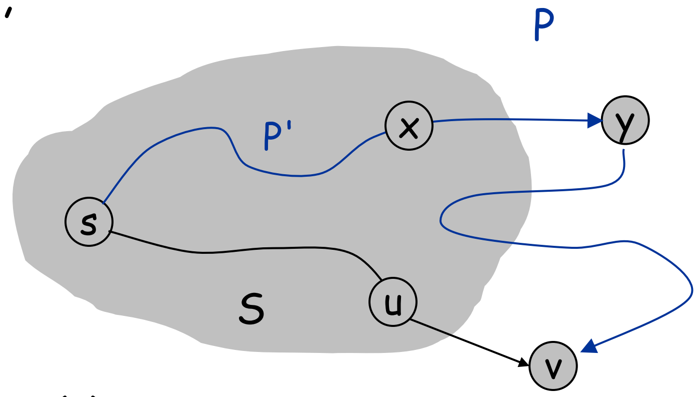
- add v to S, and set d(v) = π(v).



Dijkstra's Algorithm: Proof of Correctness

Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.
Pf. (by induction on $|S|$)
Base case: $|S| = 1$ is trivial.
Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S, and let u-v be the chosen edge.
- The shortest s-u path plus (u, v) is an s-v path of length $\pi(v)$.
- Consider any s-v path P. We'll see that it's no shorter than $\pi(v)$.
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S.



$$\ell(P) \underset{\substack{\uparrow \\ \text{nonnegative} \\ \text{weights}}}{\geq} \ell(P') + \ell(x,y) \underset{\substack{\uparrow \\ \text{inductive} \\ \text{hypothesis}}}{\geq} d(x) + \ell(x, y) \underset{\substack{\uparrow \\ \text{defn of } \pi(y)}}{\geq} \pi(y) \underset{\substack{\uparrow \\ \text{Dijkstra chose } v \\ \text{instead of } y}}{\geq} \pi(v)$$

Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain $\pi(v) = \min_{e=(u,v):u\in S} d(u) + \ell_e$

- Next node to explore = node with minimum $\pi(v)$.
- When exploring v, for each incident edge e = (v, w), update

$$\pi(v) = \min_{e=(u,v):u\in S} d(u) + \ell_e$$

Q: How to update?

Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.

PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap †
Insert	n	n	log n	$d \log_d n$	1
ExtractMin	n	n	log n	$d \log_d n$	log n
ChangeKey	m	1	log n	$\log_d n$	1
IsEmpty	n	1	1	1	1
Total		n^2	m log n	$m \log_{m/n} n$	m + n log n

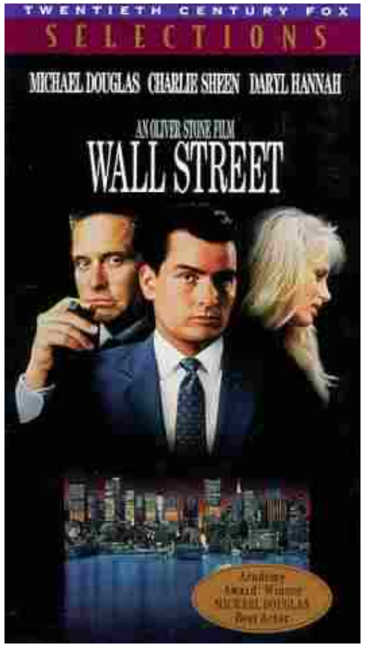
† Individual ops are amortized bounds

Extra Slides

Coin Changing

Greed is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit.

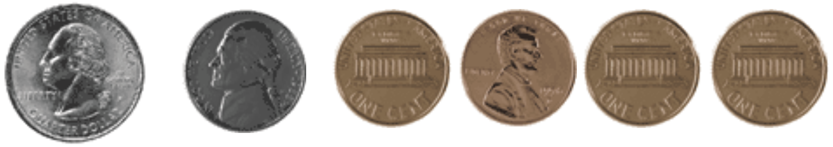
- *Gordon Gecko (Michael Douglas)*



Coin Changing

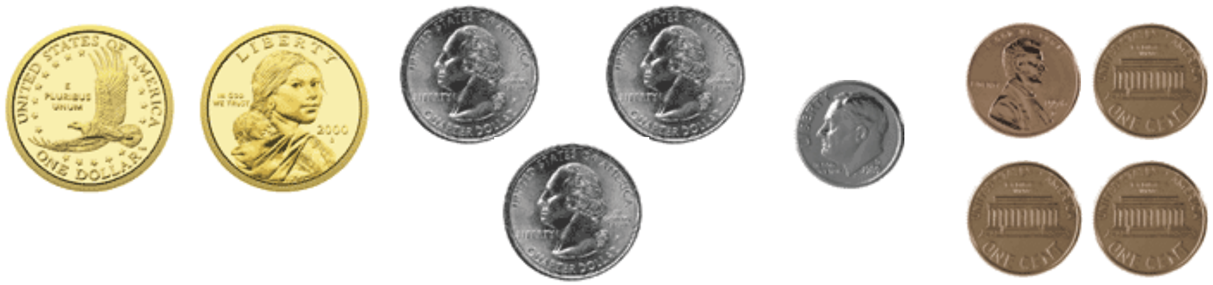
Goal. Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

Ex: 34¢.



Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Ex: $2.89.



Coin-Changing: Greedy Algorithm

Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

```
Sort coins denominations by value: $c_1 < c_2 < … < c_n$.

  coins selected
S ← $\phi$
while (x ≠ 0) {
   let k be largest integer such that $c_k \le x$
   if (k = 0)
      return "no solution found"
   x ← x - $c_k$
   S ← S ∪ {k}
}
return S
```

Q. Is cashier's algorithm optimal?

Coin-Changing: Analysis of Greedy Algorithm

Theorem. Greed is optimal for U.S. coinage: 1, 5, 10, 25, 100.

Pf. (by induction on x)

- Consider optimal way to change $c_k \leq x < c_{k+1}$: greedy takes coin k.
- We claim that any optimal solution must also take coin k.
  - if not, it needs enough coins of type $c_1, \ldots, c_{k-1}$ to add up to x
  - table below indicates no optimal solution can do this
- Problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by greedy algorithm. ▪

k	c_k	All optimal solutions must satisfy	Max value of coins 1, 2, …, k-1 in any OPT
1	1	$P \leq 4$	-
2	5	$N \leq 1$	4
3	10	$N + D \leq 2$	4 + 5 = 9
4	25	$Q \leq 3$	20 + 4 = 24
5	100	no limit	75 + 24 = 99

Coin-Changing: Analysis of Greedy Algorithm

Observation. Greedy algorithm is sub-optimal for US postal denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

Counterexample. 140¢.

- Greedy: 100, 34, 1, 1, 1, 1, 1, 1.
- Optimal: 70, 70.

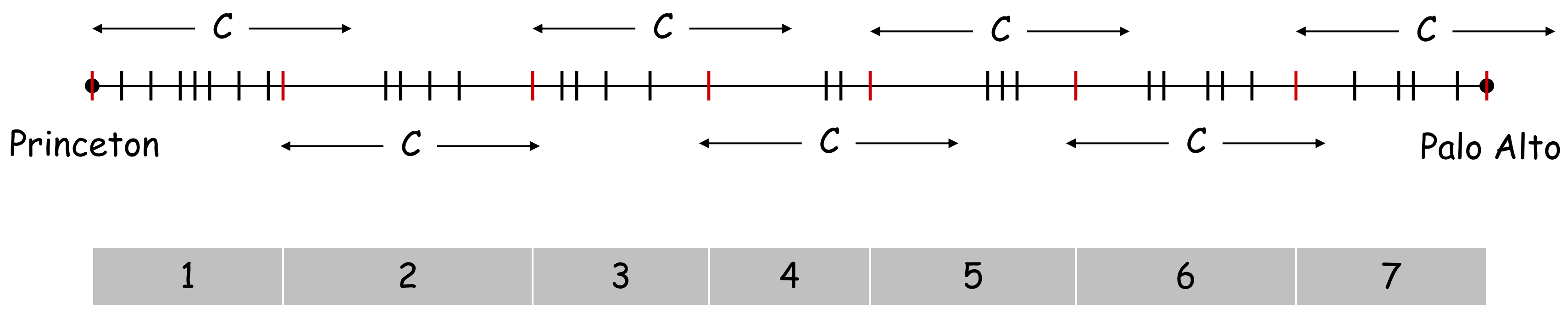
Selecting Breakpoints

Selecting Breakpoints

Selecting breakpoints.

- Road trip from Princeton to Palo Alto along fixed route.
- Refueling stations at certain points along the way.
- Fuel capacity = C.
- Goal: makes as few refueling stops as possible.

Greedy algorithm. Go as far as you can before refueling.



Selecting Breakpoints: Greedy Algorithm

Truck driver's algorithm.

```
Sort breakpoints so that: $0 = b_0 < b_1 < b_2 < ... < b_n = L$

S ← {0}  ←— breakpoints selected
x ← 0    ←— current location

while ($x \neq b_n$)
   let p be largest integer such that $b_p \leq x + C$
   if ($b_p = x$)
      return "no solution"
   x ← $b_p$
   S ← S ∪ {p}
return S
```

Implementation. O(n log n)

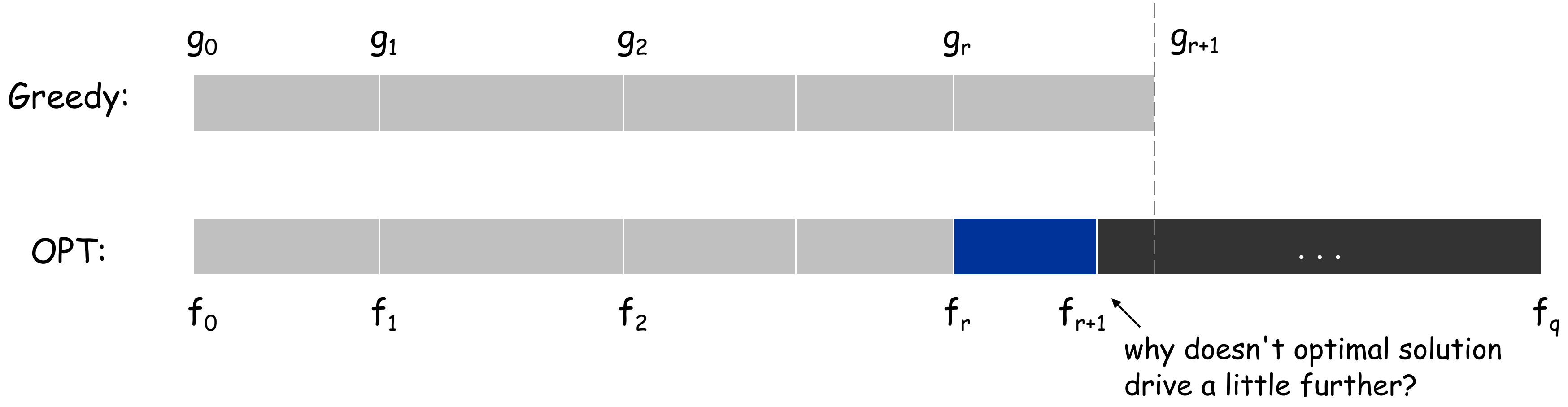
- Use binary search to select each breakpoint p.

Selecting Breakpoints: Correctness

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy.
- Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in an optimal solution with $f_0 = g_0$, $f_1 = g_1, \ldots, f_r = g_r$ for largest possible value of r.
- Note: $g_{r+1} > f_{r+1}$ by greedy choice of algorithm.

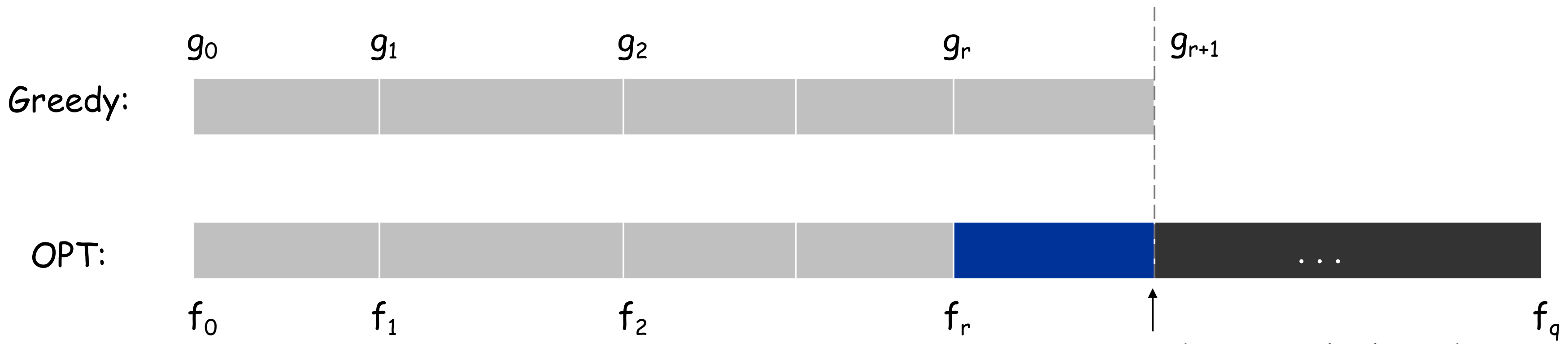


Selecting Breakpoints: Correctness

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy.
- Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in an optimal solution with $f_0 = g_0$, $f_1 = g_1, \ldots, f_r = g_r$ for largest possible value of r.
- Note: $g_{r+1} > f_{r+1}$ by greedy choice of algorithm.



another optimal solution has one more breakpoint in common $\Rightarrow$ contradiction

Edsger W. Dijkstra

The question of whether computers can think is like the question of whether submarines can swim.

Do only what only you can do.

*n.*涟漪；

In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.

史无前例

*v.*使残疾

The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.

犯罪行为

APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.