Algorithm II

4. Greedy Algorithms I

WU Xiaokun 吴晓堃

xkun.wu [at] gmail

Greedy idea

Greedy decision: builds up a solution in small steps, choosing a decision at each step *myopically* to optimize some underlying criterion.

- different **measure** leads to different solution.

Greedy idea

Greedy decision: builds up a solution in small steps, choosing a decision at each step *myopically* to optimize some underlying criterion.

- different **measure** leads to different solution.

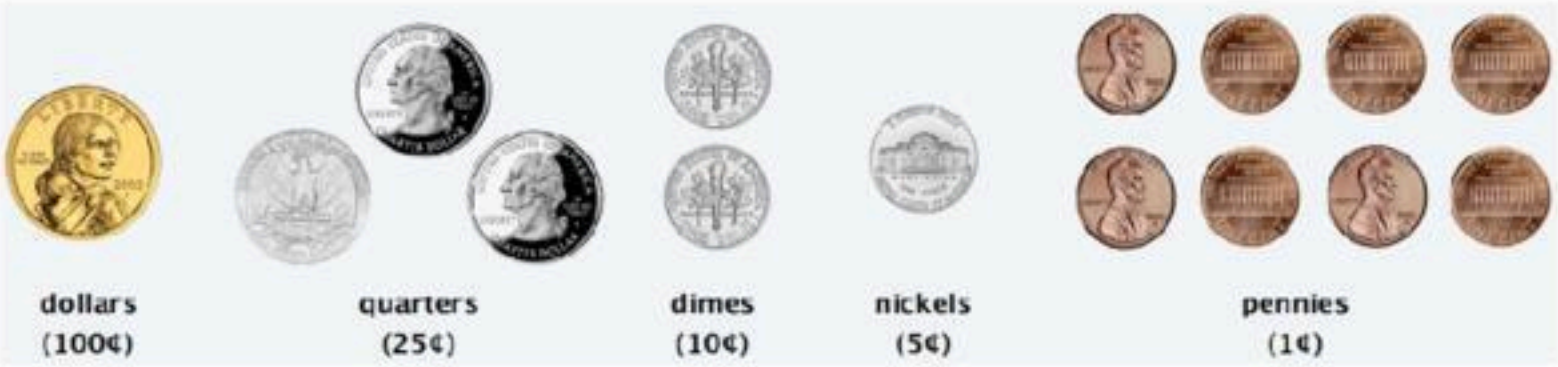
It's easy to invent greedy algorithms for almost *any* problem;

- finding cases in which they work well, and proving that they actually work well, is the interesting challenge.

Coin changing

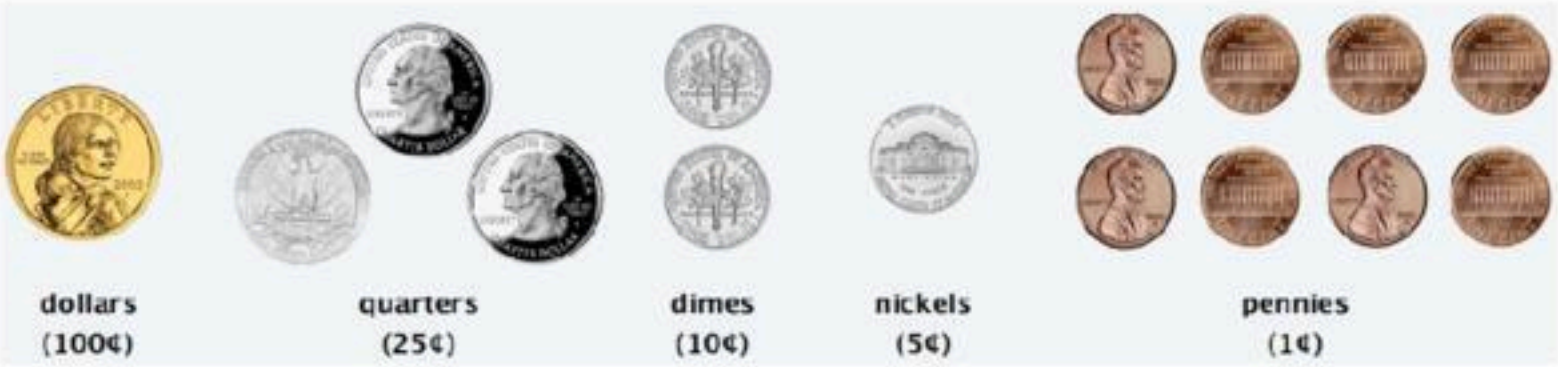
Coin changing problem

Goal. Given U.S. currency denominations $\{1, 5, 10, 25, 100\}$, devise a method to pay amount to customer using *fewest* coins.



Coin changing problem

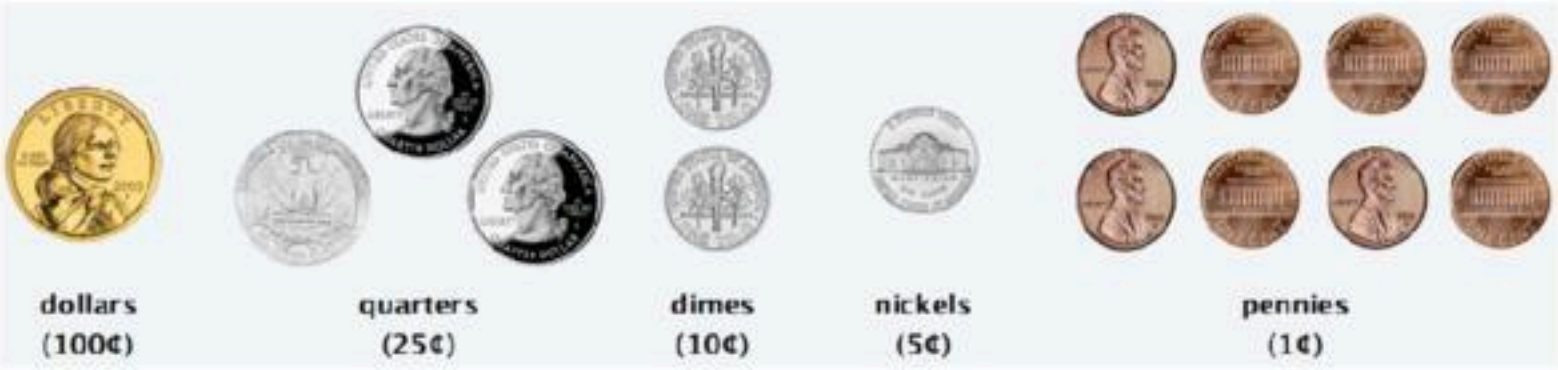
Goal. Given U.S. currency denominations $\{1, 5, 10, 25, 100\}$, devise a method to pay amount to customer using *fewest* coins.



Coin Changing Problem. Given a set $C = \{c_1, .., c_n\}$ of natural numbers and a target value S, find n multipliers $M = \{m_1, .., m_n\}$ such that $\sum_{k=1..n} m_k c_k = S$.

Be a good cashier

Customer usually does not like handsful small changes.

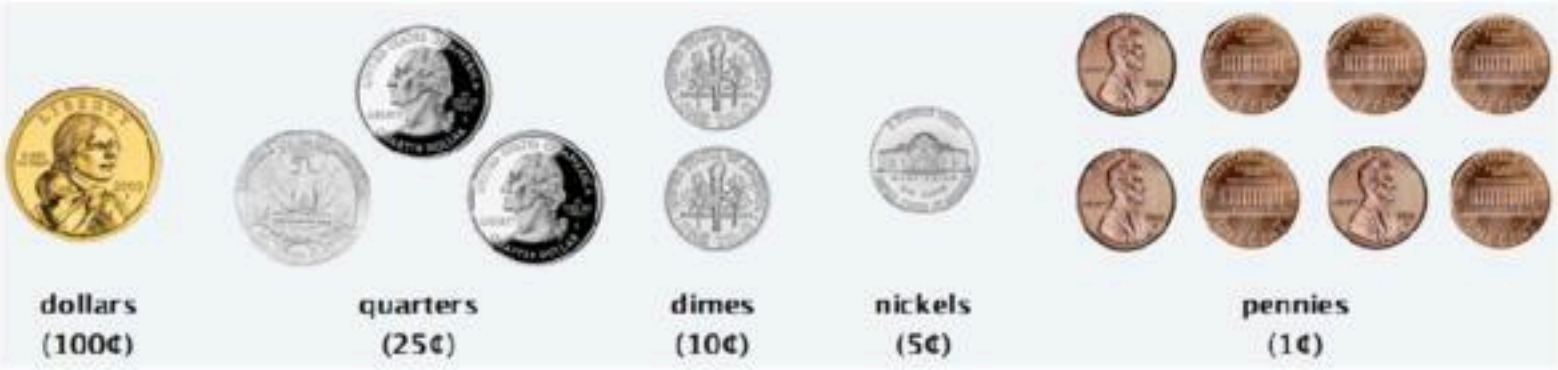


Ex. \$2.89

- $2 \times 100 + 3 \times 25 + 1 \times 10 + 4 \times 1$
- 289×1
  - customer probably complains to the manager.

Be a good cashier

Customer usually does not like handsful small changes.



Ex. $2.89

- $2 \times 100 + 3 \times 25 + 1 \times 10 + 4 \times 1$
- 289×1
  - customer probably complains to the manager.

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

Cashier's algorithm

1. SORT n denominations: $0 < c_1 < c_2 < \ldots < c_n$;
2. SET $m_1 = m_2 = \ldots = m_n = 0$;
3. WHILE $(0 < S)$:
    1. k = largest denomination c_k such that $c_k \leq S$;
    2. IF (no such k):
        1. RETURN "no solution";
    3. ELSE:
        1. S -= c_k;
        2. ++m_k;
4. RETURN M

Quiz: Cashier's algorithm

Is cashier's algorithm optimal?

- Yes, greedy algorithms are always optimal.
- Yes, for any set of coin denominations $c_1 < c_2 < \ldots < c_n$ provided $c_1 = 1$.
- Yes, because of special properties of U.S. coin denominations.
- No.

Quiz: Cashier's algorithm

Is cashier's algorithm optimal?

- Yes, greedy algorithms are always optimal.
- Yes, for any set of coin denominations $c_1 < c_2 < \ldots < c_n$ provided $c_1 = 1$.
- Yes, because of special properties of U.S. coin denominations.
- No.

No. See below.

Arbitrary coin denominations

Q. Is cashier's algorithm optimal for any set of denominations?

A. No. Consider U.S. postage: $1, 10, 21, 34, 70, 100, 350, 1225, 1500$.

- Cashier's algorithm: $140 = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1$.
- Optimal: $140 = 70 + 70$.

Arbitrary coin denominations

Q. Is cashier's algorithm optimal for any set of denominations?

A. No. Consider U.S. postage: $1, 10, 21, 34, 70, 100, 350, 1225, 1500$.

- Cashier's algorithm: $140 = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1$.
- Optimal: $140 = 70 + 70$.

A. No. It may not even lead to a feasible solution if $c_1 > 1$: $7, 8, 9$.

- Cashier's algorithm: $15 = 9 + ?$.
- Optimal: $15 = 7 + 8$.

Properties of optimal solution

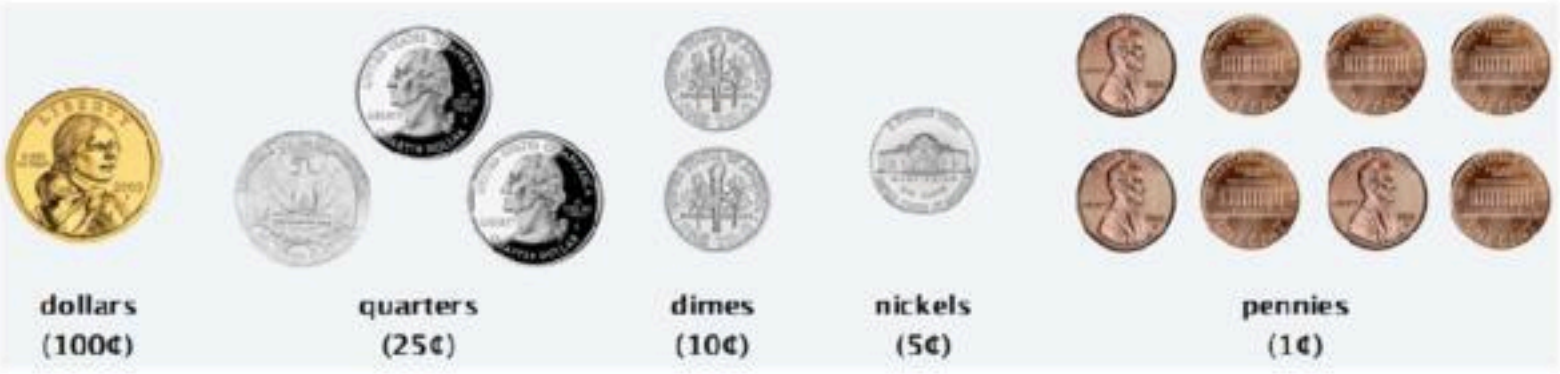
Property. Number of pennies ≤ 4.
Pf. Replace 5 pennies with 1 nickel.

Property. Number of nickels ≤ 1.

Property. Number of quarters ≤ 3.

Property. Number of nickels + number of dimes ≤ 2.
Pf.

- Recall: ≤ 1 nickel.
- Replace 3 dimes and 0 nickels with 1 quarter and 1 nickel;
- Replace 2 dimes and 1 nickel with 1 quarter.


dollars
(100¢)
quarters
(25¢)
dimes
(10¢)
nickels
(5¢)
pennies
(1¢)

Optimality of cashier's algorithm

Theorem. Cashier's algorithm is optimal for U.S. coins $\{1, 5, 10, 25, 100\}$.

Pf. [by induction on amount to be paid S]

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

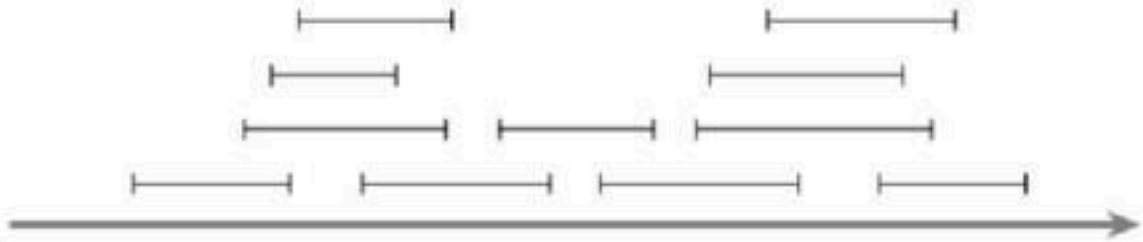
k	c_k	all optimal solutions must satisfy	max value of coin denominations $c_1, c_2, \ldots, c_{k-1}$ in any optimal solution
1	1	$P \leq 4$	-
2	5	$N < 1$	4
3	10	$N + D \leq 2$	4 + 5 = 9
4	25	$Q < 3$	20 + 4 = 24
5	100	*no limit*	75 + 24 = 99

Interval scheduling

Interval scheduling problem

Consider n subjects are sharing a *single* resource.

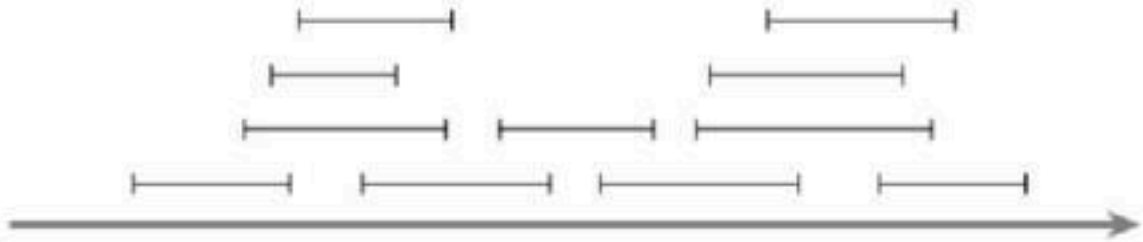
- lecture room, supercomputer, electron microscope, etc.



Interval scheduling problem

Consider n subjects are sharing a *single* resource.

- lecture room, supercomputer, electron microscope, etc.



A set of n **Jobs**:

- job j: start at s_j and finish at f_j
- two jobs are **compatible** if they do not overlap

Interval Scheduling Problem. Find maximum subset of mutually compatible jobs.

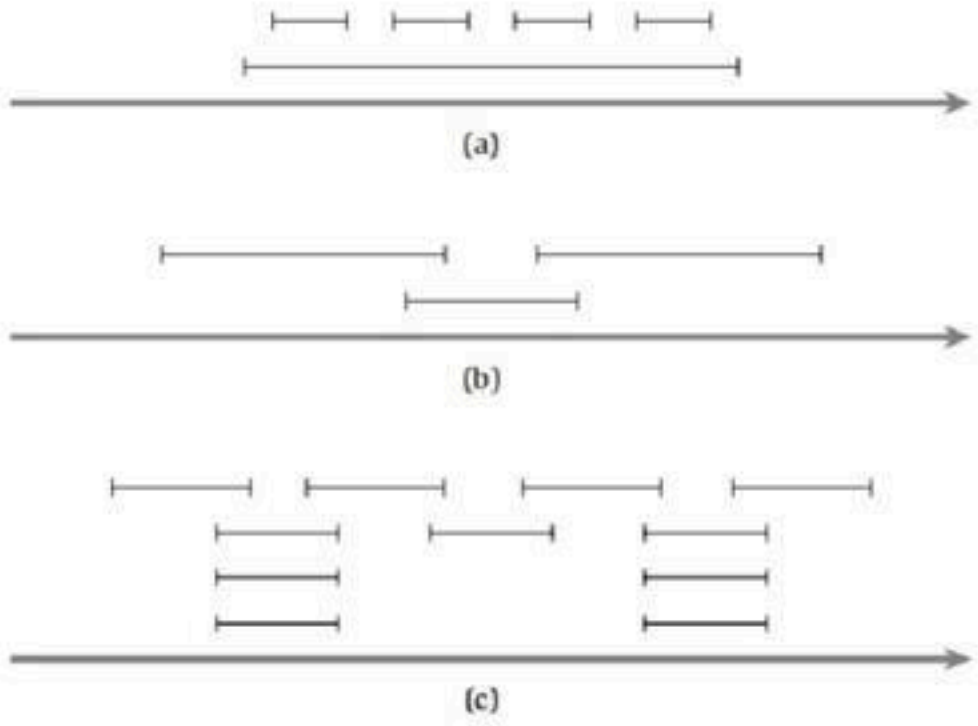
Quiz: Interval scheduling

Which rule is optimal?

- [Earliest start time] Consider jobs in ascending order of s_j.
- [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.
- [Earliest finish time] Consider jobs in ascending order of f_j.
- [Fewest incompatible] Pick the one has fewest number of incompatible requests.
- None of the above.

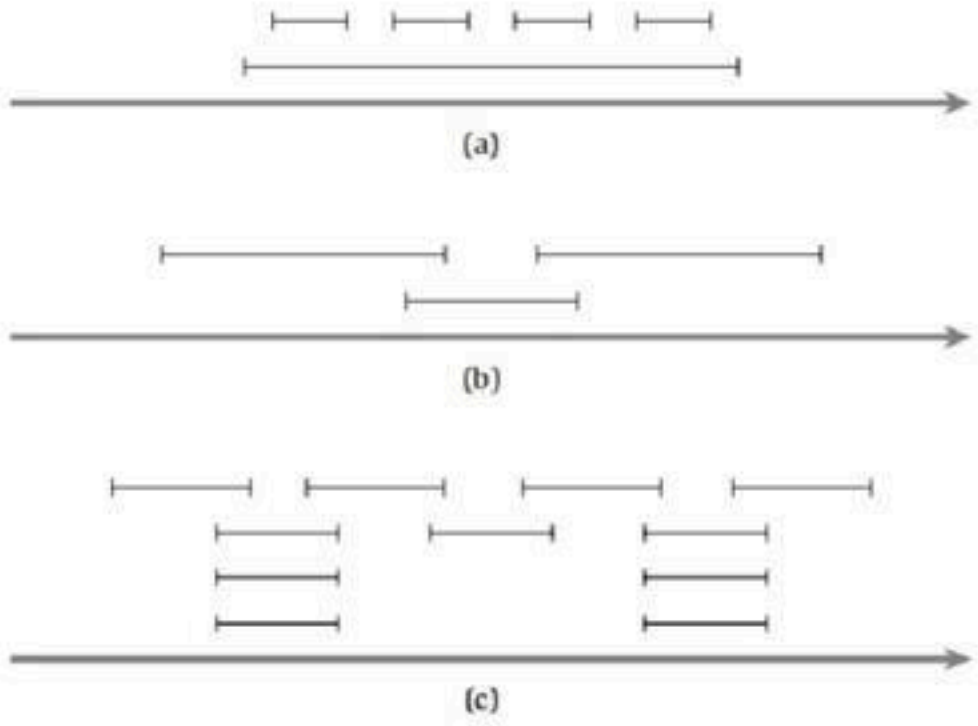
Quiz: Interval scheduling (cont.)

Different rule leads to different greedy algorithm.



Quiz: Interval scheduling (cont.)

Different rule leads to different greedy algorithm.



Claim. The earliest-finish-time-first algorithm is optimal.

Earliest-finish-time-first algorithm

1. SORT jobs by finish times and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$;
2. $S = \emptyset$;
3. FOR $j = 1..n$:
    1. IF (job j is compatible with S):
        1. $S = S \cup \{j\}$;
4. RETURN S;

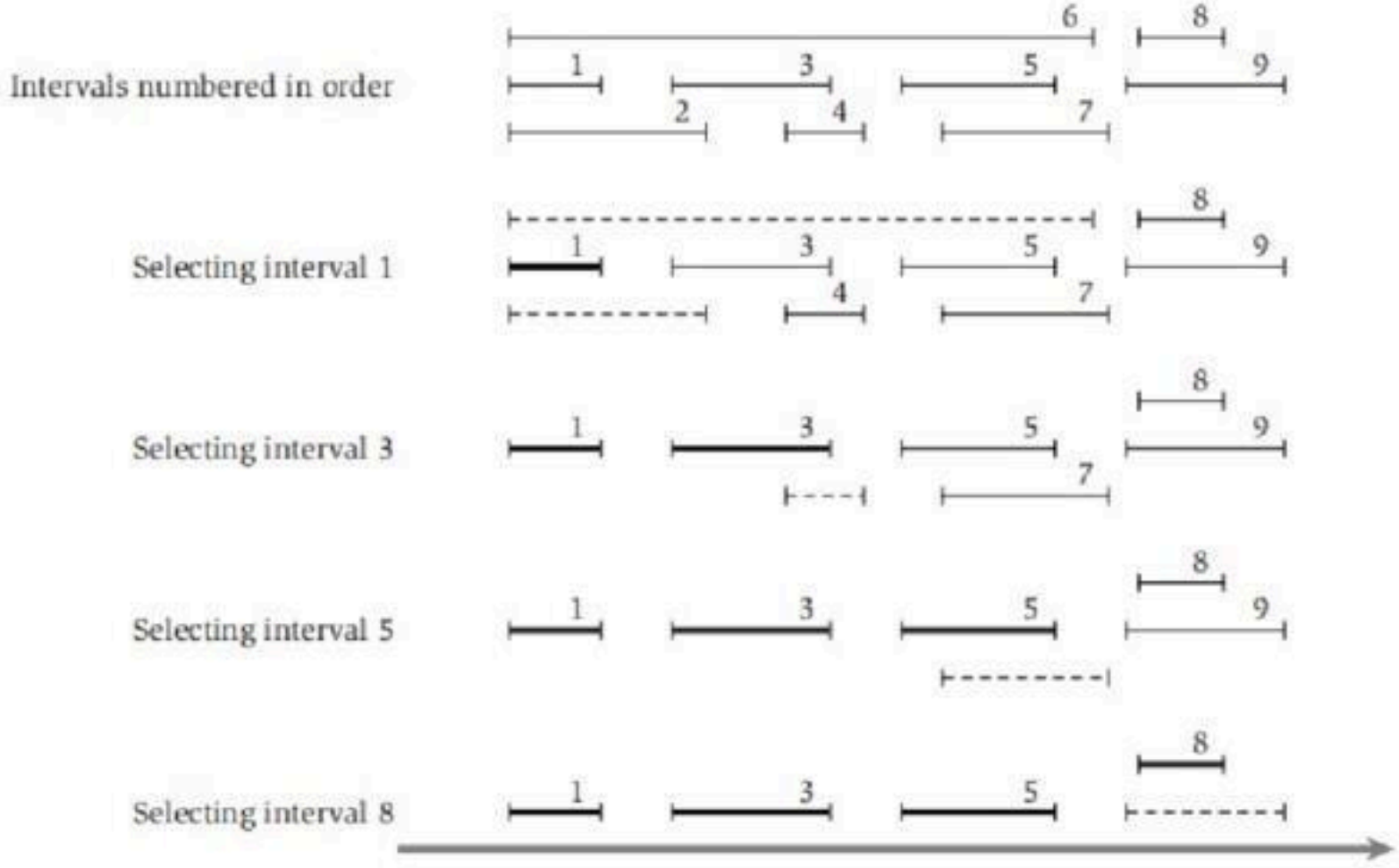
Earliest-finish-time-first algorithm

1. SORT jobs by finish times and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$;
2. $S = \emptyset$;
3. FOR $j = 1..n$:
    1. IF (job j is compatible with S):
        1. $S = S \cup \{j\}$;
4. RETURN S;

Proposition. Can implement earliest-finish-time-first in $O(n \log n)$ time.
Pf.

- Keep track of job j^* that was added last to S.
    - Job j is compatible with S iff $s_j \geq f_{j^*}$: $O(1)$ time.
- Sorting by finish times takes $O(n \log n)$ time.

Earliest-finish-time-first: example



Earliest-finish-time-first: Demo

Optimality I: “stay ahead”

Remember: greedy rules *do not* always lead to optimal solution.

- when it does, typically reveal certain interesting structure of the problem.

Optimality I: "stay ahead"

Remember: greedy rules *do not* always lead to optimal solution.

- when it does, typically reveal certain interesting structure of the problem.

Stay ahead: greedy algorithm is doing better in each step.

- attain local optimal under greedy criterion

Optimality I: "stay ahead"

Remember: greedy rules *do not* always lead to optimal solution.

- when it does, typically reveal certain interesting structure of the problem.

Stay ahead: greedy algorithm is doing better in each step.

- attain local optimal under greedy criterion

Claim. Earliest-finish-time-first is doing better in each step.

- Let $A = \{i_1, .., i_k\}$ be jobs selected by greedy
- Let $\mathcal{O} = \{j_1, .., j_m\}$ be jobs selected optimal
- Show $|A| = k = m = |\mathcal{O}|$

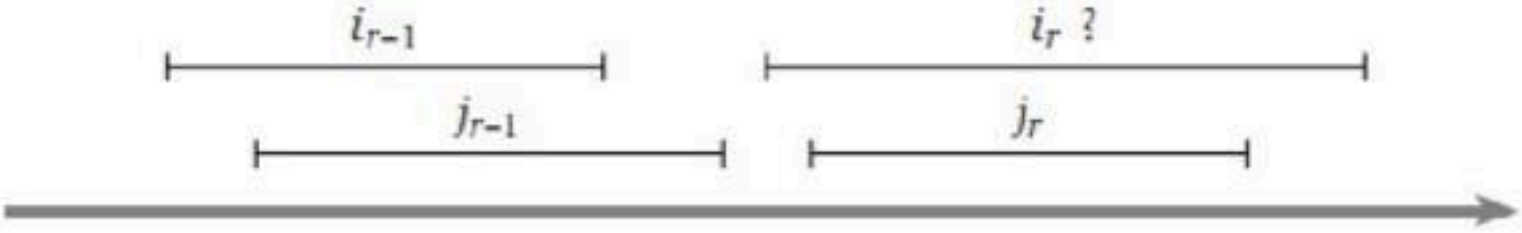
Earliest-finish-time-first: optimality

Observation. Greedy rule guarantees that $f(i_1) \leq f(j_1)$.

Lemma. For all indices $r \leq k$ we have $f(i_r) \leq f(j_r)$.
Pf. observed true for $r = 1$,

now suppose true for $r - 1$: $f(i_{r-1}) \leq f(j_{r-1})$

- compatibility of $\mathcal{O}$: $f(j_{r-1}) \leq s(j_r)$
  - $f(i_{r-1}) \leq s(j_r)$
  - j_r is a valid choice, when greedy selects i_r
    - $f(i_r) \leq f(j_r)$ due to greedy rule



Earliest-finish-time-first: optimality (cont.)

Theorem. The earliest-finish-time-first algorithm is optimal.
Pf. otherwise $m > k$.

$f(i_k) \leq f(j_k)$ by lemma

- $\exists j_{k+1} \in \mathcal{O}$: starts after j_k
  - $f(i_k) \leq f(j_k) \leq s(j_{k+1})$
  - algorithm stopped while there is a valid candidate, contradiction.

Quiz: Weighted Interval Scheduling

Suppose that each job also has a positive weight and the goal is to find a maximum weight subset of mutually compatible intervals. Is earliest-finish-time-first algorithm still optimal?

- Yes, because greedy algorithms are always optimal.
- Yes, because the same proof of correctness is valid.
- No, because the same proof of correctness is no longer valid.
- No, because you could assign a huge weight to a job that overlaps the job with earliest finish time.

Quiz: Weighted Interval Scheduling

Suppose that each job also has a positive weight and the goal is to find a maximum weight subset of mutually compatible intervals. Is earliest-finish-time-first algorithm still optimal?

- Yes, because greedy algorithms are always optimal.
- Yes, because the same proof of correctness is valid.
- No, because the same proof of correctness is no longer valid.
- No, because you could assign a huge weight to a job that overlaps the job with earliest finish time.

Weighted Interval Scheduling will be solved by Dynamic Programming optimally.

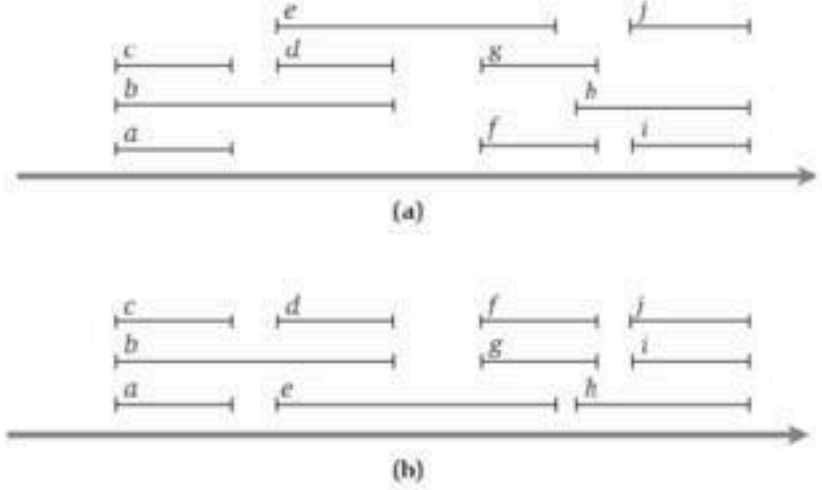
Interval Partitioning

Interval Partitioning Problem

Consider n subjects are sharing *identical* resources.

- resources are plenty now, but do not waste

Goal: find minimum number of resources so that no incompatibility for each resource.



Quiz: Interval Partitioning

Which rule is optimal?

- [Earliest start time] Consider jobs in ascending order of s_j.
- [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.
- [Earliest finish time] Consider jobs in ascending order of f_j.
- [Fewest incompatible] Pick the one has fewest number of incompatible requests.
- None of the above.

Depth of intervals

Depth. Maximum number that pass over *any single point* on the time-line.

Claim. In any instance of Interval Partitioning, the number of resources needed is *at least* the depth of intervals set.

Depth of intervals

Depth. Maximum number that pass over *any single point* on the time-line.

Claim. In any instance of Interval Partitioning, the number of resources needed is *at least* the depth of intervals set.

Key Observation. If an algorithm *always* produce a schedule using resources *equal* to the depth, then it must be optimal.

- Number of resources needed $\geq$ depth.

Pf. Optimality: "characteristic bound"

"**Characteristic bound**": greedy algorithm *always* attain a certain characteristic value.

- if the value bounds *every possible* optimal solution, greedy must be optimal.
- **Key**: find the characteristic value of the problem.

Pf. Optimality: "characteristic bound"

"**Characteristic bound**": greedy algorithm *always* attain a certain characteristic value.

- if the value bounds *every possible* optimal solution, greedy must be optimal.
- **Key**: find the characteristic value of the problem.

Claim. Earliest-start-time-first algorithm uses a number of resources *equal* to the depth of intervals.

- let D be the depth, d the current allocations
  - so $d \leq D$

Earliest-start-time-first algorithm

1. SORT requests by start times and renumber so that $s_1 \leq s_2 \leq \ldots \leq s_n$;
2. $d = 0$;
3. FOR $j = 1..n$:
    1. IF (request j is compatible with resource k):
        1. schedule request j with resource k;
    2. ELSE:
        1. $++d$;
        2. schedule request j with resource d;
4. RETURN schedule;

Earliest-start-time-first: Demo

Earliest-start-time-first: analysis

Proposition. The earliest-start-time-first algorithm can be implemented in $O(n \log n)$ time.

Pf.

- Sorting by start times takes $O(n \log n)$ time.
- Store resources in a `priority queue` (`key` = *finish time of its last request*).
  - to allocate a new resource, `INSERT` resource onto priority queue.
  - to schedule request j in resource k, `INCREASE-KEY` of resource k to f_j.
  - to determine whether request j is compatible with any resource, compare s_j to `FIND-MIN`
- Total # of priority queue operations is $O(n)$; each takes $O(\log n)$ time.

Remark. This implementation chooses a resource k whose *finish time of its last request is the earliest.*

- already sorted by start time

Earliest-start-time-first: optimality

Observation. ESTF *never* schedules two incompatible requests with same resource; *every* request will be fulfilled (by new allocation).

Theorem. Earliest-start-time-first algorithm is optimal.
Pf.

- When algorithm terminates, d resources are allocated in total.
  - Resource d is allocated because we needed to schedule a request, say j, that is incompatible with a request in each of d–1 other resources.
  - Thus, these d requests each end after s_j.
  - Since we sorted by start time, each of these incompatible requests start no later than s_j.
  - Thus, we have d requests overlapping at time $s_j + \epsilon$.
    - By definition of depth, $d \leq D$
- Key observation $\Rightarrow$ all schedules use $\geq D$ resources.

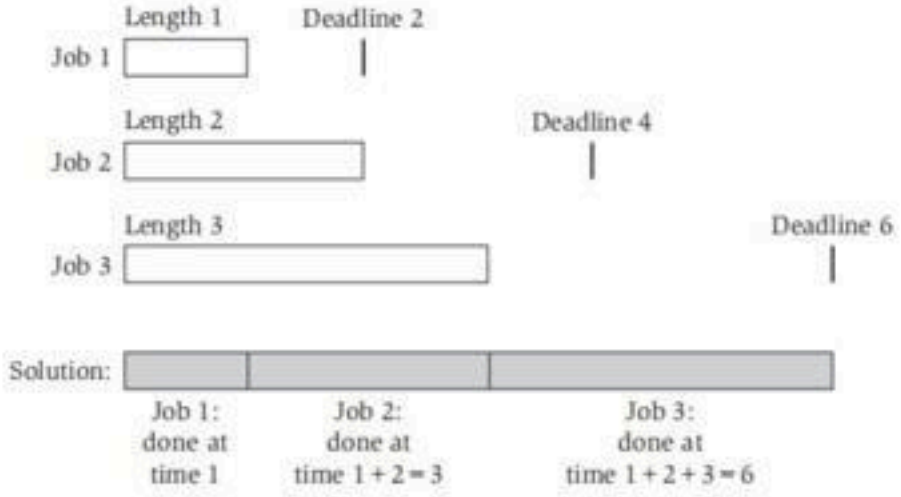
Minimize Lateness

Scheduling to minimize lateness

Consider n subjects are sharing a *single* resources.

- job j requires t_j unit time and due at d_j
  - determine start time s_j, so finish at $f_j = s_j + t_j$
- **Lateness**: $l_j = \max\{0, f_j - d_j\}$

Goal: minimize the **maximum lateness**: $L = \max_j l_j$.



Quiz: Minimize Lateness

Which order minimizes the maximum lateness?

- [shortest processing time] Ascending order of processing time t_j.
- [smallest slack] Ascending order of slack: $d_j - t_j$.
- [earliest deadline first] Ascending order of deadline d_j.
- None of the above.

Quiz: Minimize Lateness

Which order minimizes the maximum lateness?

- [shortest processing time] Ascending order of processing time t_j.
- [smallest slack] Ascending order of slack: $d_j - t_j$.
- [earliest deadline first] Ascending order of deadline d_j.
- None of the above.

shortest processing time

- $t_1 = 1, d_1 = 100$
- $t_2 = 10, d_2 = 10$: optimal

smallest slack

- $t_2 = 10, d_2 = 10$
- $t_1 = 1, d_1 = 2$: best

length is not even related?

Earliest-deadline-first: algorithm

1. SORT jobs by due times and renumber so that $d_1 \leq d_2 \leq \ldots \leq d_n$;
2. $t = 0$;
3. FOR $j = 1..n$:
   1. Assign job j to interval $[t, t + t_j]$;
   2. $s_j = t$; $f_j = t + t_j$;
   3. $t = t + t_j$;
4. RETURN intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$;

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

Minimize Lateness: no idle time

Observation 1 Earliest-deadline-first has no "gap".

- each interval starts just when the previous ends.

Minimize Lateness: no idle time

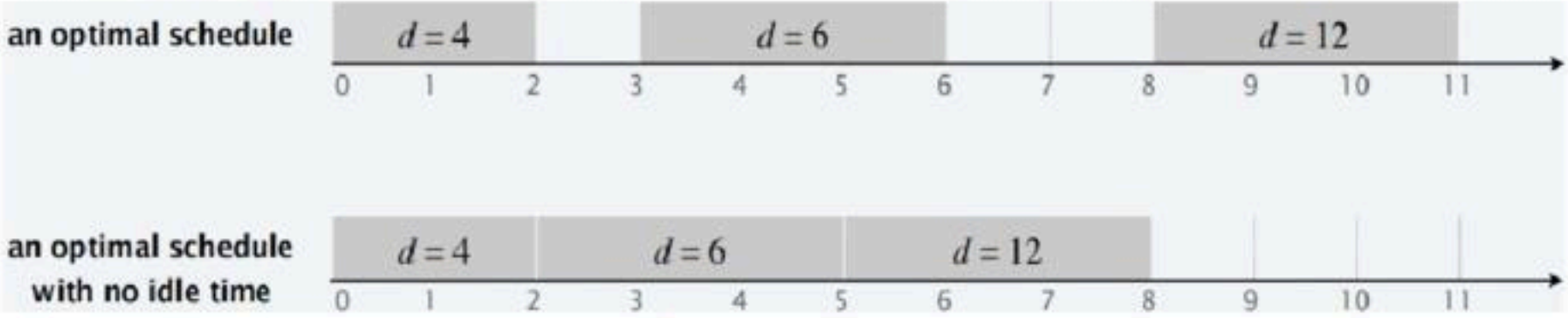
Observation 1 Earliest-deadline-first has no "gap".

- each interval starts just when the previous ends.

Idle time: there is work to be done, yet machine is sitting idle.

Observation 2. There exists an optimal schedule with no idle time.

- given an optimal schedule, just move each interval earlier to squeeze out gaps.



Optimality II: exchange argument

Note: no-idle-time is a common property shared by greedy and optimal.

Optimality II: exchange argument

Note: no-idle-time is a common property shared by greedy and optimal.

Exchange argument: given an optimal $\mathcal{O}$, gradually modify it

- preserving optimality at each step
- eventually transforming it into a solution identical to greedy

Optimality II: exchange argument

Note: no-idle-time is a common property shared by greedy and optimal.

Exchange argument: given an optimal $\mathcal{O}$, gradually modify it

- preserving optimality at each step
- eventually transforming it into a solution identical to greedy

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

..	d_i	..	d_j	..
..	j	..	i	..

Observation 3 Earliest-deadline-first has no inversion (and no idle time).

Exchange argument: properties to keep

Claim. All schedules with no inversions and no idle time have the same maximum lateness.

Pf.

- only differ in the order of jobs with identical deadlines.
  - they are scheduled consecutively
- excluding these jobs, the order is fixed (no inversion) and compact (no idle time)

Exchange argument: properties to keep

Claim. All schedules with no inversions and no idle time have the same maximum lateness.

Pf.

- only differ in the order of jobs with identical deadlines.
  - they are scheduled consecutively
- excluding these jobs, the order is fixed (no inversion) and compact (no idle time)

Key to show optimality: there is an optimal schedule that has no inversions and no idle time.

- start with any optimal schedule with no idle time
- convert it into schedule with no inversion, without increasing maximum lateness

Optimality: adjacent inversion

Observation 4. If an idle-free schedule has an inversion, then it has an adjacent inversion (scheduled consecutively).

Pf. Let i–j be the *closest* inversion.

Let k be element immediately to the right of j.

- $[j > k]$ Then j–k is an adjacent inversion.
- $[j < k]$ Then i–k is a *closer* inversion since $i < \ldots < j < k$, contradiction.

..	d_i	..	d_j	d_k	..
.. k	j	..	i		..
..	j k	..	i		..

Optimality: swap

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

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

- $l'_k = l_k$ for all $k \neq i, j$.
  - $l'_j \leq l_j$
- if i is late: $l'_i = f'_i - d_i = f_j - d_i$
  - $i - j$ is inversion: $d_i \geq d_j$
  - $l'_i = f_j - d_i \leq f_j - d_j = l_j$

\	..	d_i	..	d_j	..
before swap	..	j	..	i	..
after swap	..	i	..	j	..

Earliest-deadline-first: optimality

Theorem. The earliest-deadline-first schedule is optimal.
Pf.

- an optimal schedule with no inversions and no idle time exists
  - all schedules with no inversions and no idle time have same maximum lateness
  - earliest-deadline-first always produces such schedule

Greedy analysis strategies

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

Characteristic bound. Discover a simple characteristic 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 greedy algorithm without hurting its quality.

Greedy analysis strategies

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

Characteristic bound. Discover a simple characteristic 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 greedy algorithm without hurting its quality.

Other greedy algorithms. Gale–Shapley, Kruskal, Prim, Dijkstra, Huffman, etc.

Interested in solving problem?

LeetCode: https://leetcode.com

Optimal Caching

Caching

Caching.

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

Applications. CPU, RAM, hard drive, web, browser, etc.

Cache maintenance

Goal. Eviction schedule that minimizes the number of evictions.

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

- Optimal eviction schedule: 2 evictions.

a	**b**	**c**	**b**	**c**	**a**	**b**
a	a	**c**	c	c	**a**	a
b	b	b	b	b	b	b

Cache maintenance: greedy options

LIFO/FIFO. Evict item brought in least (most) recently.

LRU. Evict item whose most recent access was earliest.

LFU. Evict item that was least frequently requested.

Farthest-in-future

Farthest-in-future. Evict item in the cache that is not requested until farthest in the future.

Theorem. [Bélády 1966] FF is optimal eviction schedule.

- as usual, design is simple but analysis is subtle

FF: 2 evictions

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

Farthest-in-future: example

FF: 2 evictions

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

Farthest-in-future: example

FF: 2 evictions

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

Note: other options may be just as good.

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

So why FF is optimal?

Exchange argument: intuition

Observation. Swapping one decision for another does not change cost.

- the other decision will be taken anyway

Exchange argument: intuition

Observation. Swapping one decision for another does not change cost.

- the other decision will be taken anyway

Def. A **reduced** schedule does minimal amount of work necessary in a given step.

- only bring d into cache if there is a request to d, and d is missing
  - exactly the number of misses
- sometimes also called "lazy" strategy

Exchange argument: intuition

Observation. Swapping one decision for another does not change cost.

- the other decision will be taken anyway

Def. A **reduced** schedule does minimal amount of work necessary in a given step.

- only bring d into cache if there is a request to d, and d is missing
  - exactly the number of misses
- sometimes also called "lazy" strategy

Note: FF is clearly reduced

- Key: prove for every non-reduced schedule, there is an equally good reduced schedule

Reduced schedule

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

Pf. [induction on step j] case-by-case discussion:

- Case 1: S bring in d without a request
- Case 2: S bring in d even thought d is in cache
- if multiple happens, apply each in turn
  - Case 1 should applied first, which may trigger Case 2

Or just "pretends" to cache but actually leaves d out

- only bring in d when requested.

Optimality of FF

Note: for any reduced schedule, the number of items that are brought in is exactly the number of misses.

Let S_{FF} denote schedule produced by Farthest-in-Future,
Let S^* denote a schedule that incurs minimum possible number of misses

- now gradually "transform" schedule S^* into S_{FF}, one eviction decision at a time
- without increasing number of misses.

Optimality of FF

Note: for any reduced schedule, the number of items that are brought in is exactly the number of misses.

Let S_{FF} denote schedule produced by Farthest-in-Future,
Let S^* denote a schedule that incurs minimum possible number of misses

- now gradually "transform" schedule S^* into S_{FF}, one eviction decision at a time
- without increasing number of misses.

Theorem. FF is optimal eviction algorithm, since it incurs no more misses than any other schedule.

Lemma. If S_{FF} and S make same eviction decision through first j steps, then there is a reduced schedule S' that make same eviction decision as S through first $j+1$ steps, and incurs no more misses than S.

Proving Optimality of FF

S_{FF} and S have same cache contents.
Let d denote the item requested in step $j+1$.

- d in cache: $S' = S$
- S_{FF} and S evict same item: $S' = S$
- S_{FF} evict e, S evict $f \neq e$
  - S' should evict e, but sync with S ASAP
    - S' agrees with S_{FF} through first $j+1$ steps
    - show having f is no worse than having e
  - from step $j+2$ onward, S' behaves exactly like S, until ...

S			steps	S'		
same	e	f	step j	same	e	f
same	e	d	step j + 1	same	d	f

Proving Optimality of FF (cont.)

... until following happens *for the first time*:

- A. request to $g(\neq e, f)$ not in cache, and S evict e
  - S' evict f
- request to f, and S evict e'
  - B. if $e' = e$: no change
  - C. otherwise: S' evict e' and take e
    - S' may not reduced: transform it to a reduction
- request to e: impossible
  - S_{FF} evict e in step $j + 1$, request to f must come earlier

S			steps	S'		
same	e	d	step j + 1	same	d	f
same	g	d	step A	same	d	g
same	f	d	step B	same	d	f
same	e	d	step C	same	d	e

Cache maintenance: extensions

Problem of FF: in practice, generally impossible to know future request order.

- often used as baseline for offline comparisons
- Experimentally, Least-Recently-Used (LRU) is the best
- Caching is among most fundamental online problems in CS.