Greedy Algorithms
CS 3AC3

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Acknowledgments: Material based on *Algorithm Design* by Jon Kleinberg and Éva Tardos (Chapter 4)

Informal Definition

- A **greedy algorithm** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- Greedy algorithms do not always yield optimal solution, but for many problems they do, and if they do, they are usually the most efficient.
- Popular Dijkstra's Shortest Paths algorithm is greedy.

Coin Changes

- Given currency denominations, say : 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins. For example: $34 = 25 + 5 + 4 \times 1$.

Algorithm (Cashier's algorithm)

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

CASHIERS-ALGORITHM $(x, c_1, c_2, \ldots, c_n)$

SORT n coin denominations so that $c_1 < c_2 < \ldots < c_n$

$S \leftarrow \phi$ ⟵ set of coins selected

WHILE $x > 0$

$k \leftarrow$ largest coin denomination c_k such that $c_k \leq x$

IF no such k, RETURN "no solution"

ELSE

$x \leftarrow x - c_k$

$S \leftarrow S \cup \{ k \}$

RETURN S

Analysis of cashier's algorithm

Proposition

Cashier's algorithm is optimal for coins: 1, 5, 10, 25, 100.

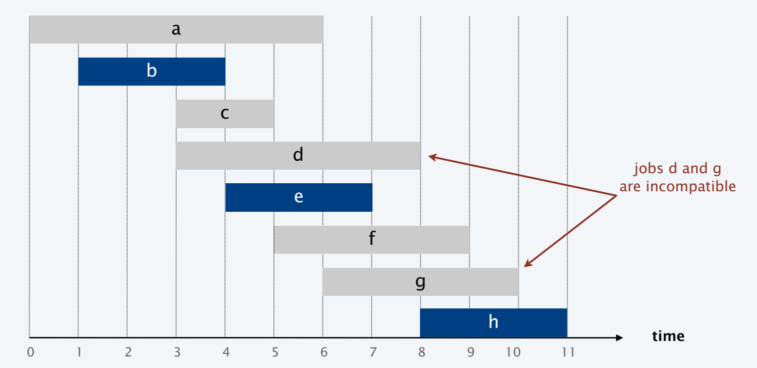
Proof.

By analysis cases: $1 \leq n \leq 4$, $5 \leq n \leq 9$, $10 \leq n \leq 24$, $25 \leq n \leq 99$, and $100 \leq n$. □

- Cashier's algorithm does not always work!
- Consider a denomination $1, 4, 5, 10$, and $n = 8$. Cashier's algorithm produces: 8=5+1+1+1, while the optimal solution is $8 = 4 + 4$.
- Cashier's algorithm may not even lead to a feasible solution if $c_1 > 1$! Consider a denomination: $7, 8, 9$ and $n = 15$. The optimal solution is $15 = 7 + 8$, but the algorithm gives $15 = 9+??$.

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.



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.

Usually only some templates work!

Templates that do not work for interval scheduling

counterexample for earliest start time

counterexample for shortest interval

counterexample for fewest conflicts

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$

$A \leftarrow \phi$ ⟵ set of jobs selected

FOR $j = 1$ TO n

  IF job j is compatible with A

    $A \leftarrow A \cup \{ j \}$

RETURN A

Claim

Time complexity of the part from 'FOR' to 'RETURN' is $O(n)$.

Proof.

- Keep track of job j^* that was added last to A (*constant time*).
- Job j is compatible with A iff $s_j \geq f_j^*$ (*constant time*) . □

However time complexity of the Earliest-finish-time-first algorithm is $O(n \log n)$! WHY?

Time complexity of earliest-finish-time-first algorithm

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$

$A \leftarrow \phi$ ⟵ set of jobs selected

FOR $j = 1$ TO n

  IF job j is compatible with A

    $A \leftarrow A \cup \{j\}$

RETURN A

Proposition

We can implement earliest-finish-time first (EFTF) in $O(n \log n)$ time.

Proof.

$O(\text{EFTF}) = O(\text{SORT}) + O(A \leftarrow \emptyset) + O(\text{FOR...RETURN } A)$
$O(\text{SORT}) = O(n \log n)$
$O(A \leftarrow \emptyset) = O(1)$
$O(\text{FOR...RETURN } A) = O(n)$
Hence $O(\text{EFTF}) = O(n \log n) + O(1) + O(n) = O(n \log n)$. □

Analysis of of earliest-finish-time-first algorithm (1)

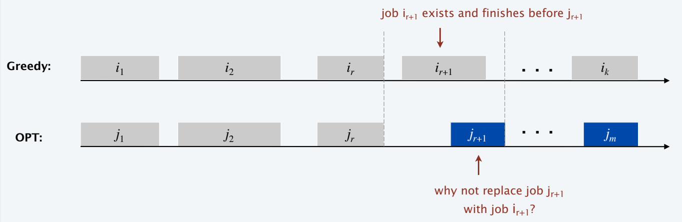
Theorem

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

Proof.

(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 an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



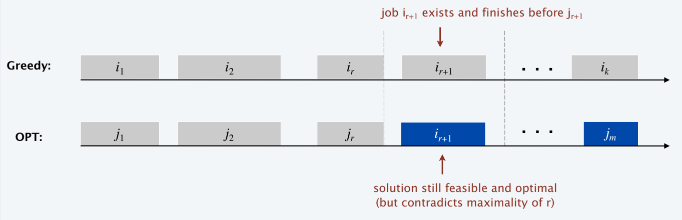
Theorem

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

Proof.

(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 an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

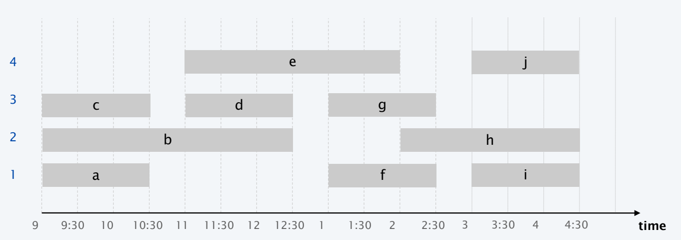


□

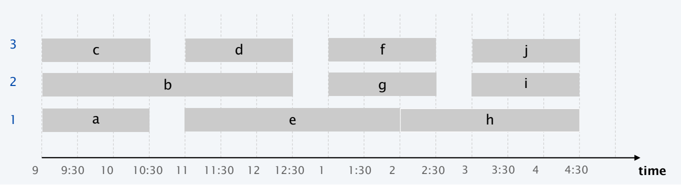
Interval partitioning

- 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 lectures occur at the same time in the same room.

Example. This schedule uses 4 classrooms to schedule 10 lectures.



Example. This schedule uses 3 classrooms to schedule 10 lectures.



Interval partitioning: greedy algorithms

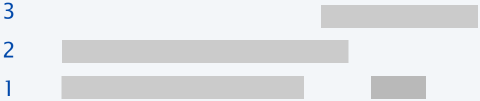
Greedy template. Consider jobs in some natural order. Assign each lecture to an available classroom (which one?); allocate a new classroom if none are available.

- 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.

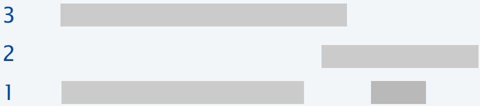
Usually only some templates work!

Templates that do not work for interval partitioning

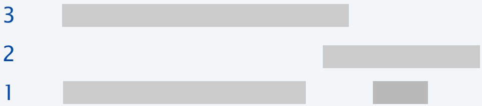
counterexample for earliest finish time



counterexample for shortest interval



counterexample for fewest conflicts



Interval partitioning: earliest-start-time-first algorithm

EARLIEST-START-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT lectures by start time so that $s_1 \leq s_2 \leq \ldots \leq s_n$.

$d \leftarrow 0$ ⟵ number of allocated classrooms

FOR $j = 1$ TO n

  IF lecture j is compatible with some classroom

    Schedule lecture j in any such classroom k.

  ELSE

    Allocate a new classroom $d + 1$.

    Schedule lecture j in classroom $d + 1$.

    $d \leftarrow d + 1$

RETURN schedule.

Time complexity of earliest-start-time-first algorithm

Proposition

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

Proof.

Store classrooms in a priority queue (key = finish time of its last lecture).

- To determine whether lecture j is compatible with some classroom, compare s_j to key of min classroom k in priority queue.
- To add lecture j to classroom k, increase key of classroom k to f_j.
- Total number of priority queue operations is $O(n)$.
- Sorting by start time takes $O(n \log n)$ time.

$\square$

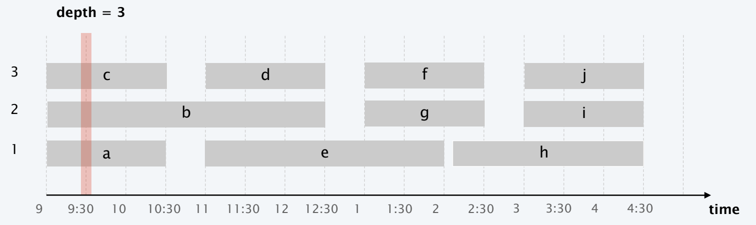
Definition

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.

Question. Does number of classrooms needed always equal depth?

Answer. Yes! Moreover, earliest-start-time-first algorithm finds one.



Analysis of earliest-start-time-first algorithm

Key Observation. The earliest-start-time first algorithm never schedules two incompatible lectures in the same classroom.

Theorem

Earliest-start-time-first algorithm is optimal.

Proof.

- Let $d =$ number of classrooms that the algorithm allocates.
- Classroom d is opened because we needed to schedule a lecture, say j, that is incompatible with all $d-1$ other classrooms.
- These d lectures 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 $s_j + \epsilon$.
- Key observation $\implies$ all schedules use $\geq d$ classrooms.

□

- 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. Dijkstra (shortest paths),Kruskal, Prim (spanning trees), Huffman (compression), Optimal Offline Caching, etc.
- Dijkstra (shortest paths),Kruskal and Prim (spanning trees) were discussed in CS 2C03, Huffman (compression) and Optimal Offline Caching will be now be discussed.

Huffman codes and data compression

- We have a set of characters $A = \{a_1, \ldots, a_k\}$.
- A string or message: $x_1 x_2 \ldots x_n$ where $x_i \in A$.
- For each $a_i \in A$, $f(a_i)$ is the frequency (or probability) of appearance a_i in the message.
- We assume $\sum_{a_i \in A} f(a_i) = 1$.
- **Encoding**: assign a *binary code* $c(a_i)$ for each a_i, and extend c to strings by $c(x_1 x_2 \ldots x_n) = c(x_1)c(x_2)\ldots c(x_n)$.
- **Decoding**: Given a code $b_1 b_2 \ldots b_m$ find the **unique** message $x_1 x_2 \ldots x_n$ such that $c(x_1 x_2 \ldots x_n) = b_1 b_2 \ldots b_m$.
- Encoding Length or Average Code Length:

$$\sum_{a_i \in A} f(a_i) length(c(a_i)).$$

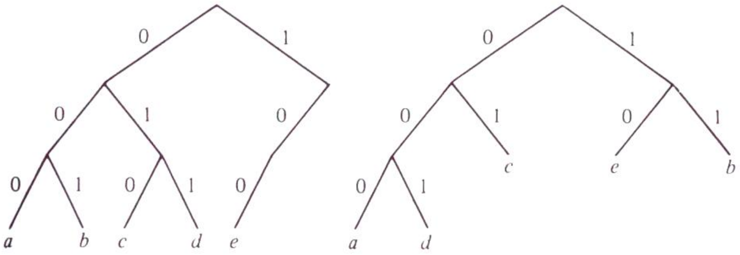
Prefix Property

Character	Frequency	Code 1	Code 2	Code 3
a	0.3	000	01	00
b	0.1	001	0010	01
c	0.1	010	0011	10
d	0.1	011	000	000
e	0.4	100	1	1
average code length		3.0	2.1	1.7

- **Prefix Property:** $c(a_i)$ **is not** a prefix of $c(a_j)$ for any $c \neq j$.
- No prefix property $\Longrightarrow$ no decoding!
- Code1 and Code 2 have prefix property.
- Code 3 does not have prefix property.

Binary Tree Representation of codes

- Codes **with** prefix properties: all letters as leaves.



code 1

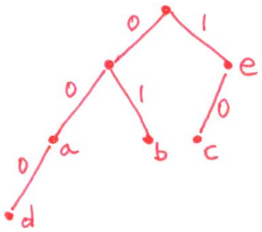
$c(a) = 000,\ c(b) = 001,$
$c(c) = 010,\ c(d) = 011,$
$c(e) = 100$

code 2

$c(a) = 000,\ c(b) = 11,$
$c(c) = 01,\ c(d) = 001,$
$c(e) = 10$

Binary Tree Representation of codes

- Codes **with out** prefix properties: some letters as interior nodes.



code 3

$c(a) = 00$, $c(b) = 01$,
$c(c) = 10$, $c(d) = 000$,
$c(e) = 1$

Huffman Algorithm

- Huffman code: an optimal (minimal length) code with prefix property.
- Huffman algorithm: it finds Huffman code $c(a_i)$ for each a_i.
- Huffman algorithm is *greedy* (see 'lowest frequencies' below).

HUFFMAN($\{a_1, a_2, \dots, a_n\}$)

Find a_i and a_j such that $f(a_i)$ and $f(a_j)$ are the lowest frequencies among $f(a_1), \dots, f(a_n)$
Define a new character a' and set
$f(a') \leftarrow f(a_i) + f(a_j)$.
$A' \leftarrow (\{a_1, a_2, \dots, a_n\} \setminus \{a_i, a_j\}) \cup \{a'\}$
HUFFMAN(A')
$c(a_i) \leftarrow c(a')0$
$c(a_j) \leftarrow c(a')1$
END HUFFMAN

Example

$A = \{a, b, c\}$, $f(a) = 0.5$, $f(b) = 0.3$, $f(c) = 0.2$.

HUFFMAN($\{a, b, c\}$) $\implies$ we assume $a' = [bc]$

$f([bc]) = f(b) + f(c) = 0.4$ and next HUFFMAN($\{a, [bc]\}$) $\implies$

$c(a) = 0, c(bc]) = 1 \implies c(b) = 10, c(c) = 11.$

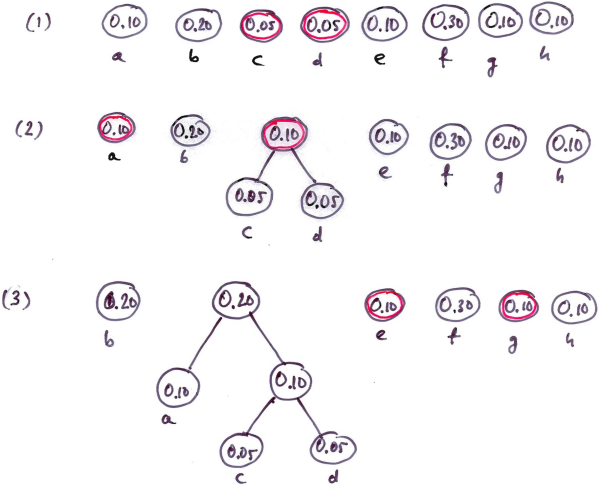
Hence we have $c(a) = 0, c(b) = 10, c(c) = 11$.

- The procedure is better understood if presented in *bottom up* version with trees.

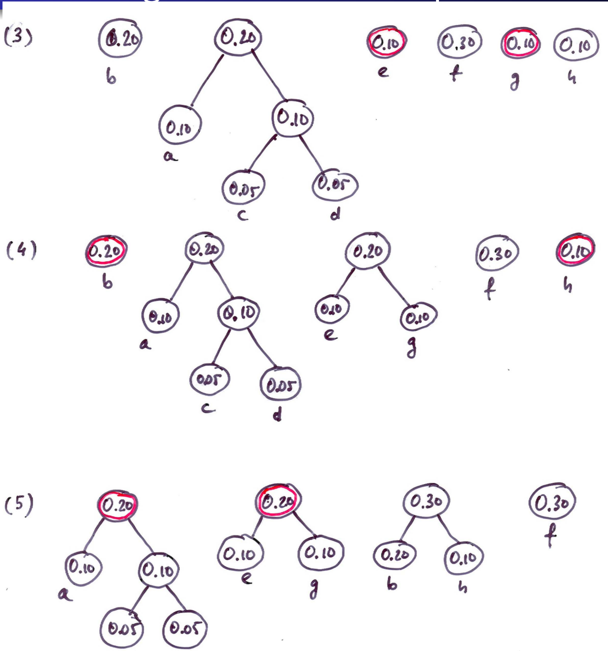
Huffman Algorithm: An Example

	a	b	c	d	e	f	g	h
f	0.10	0.20	0.05	0.05	0.10	0.30	0.10	0.10

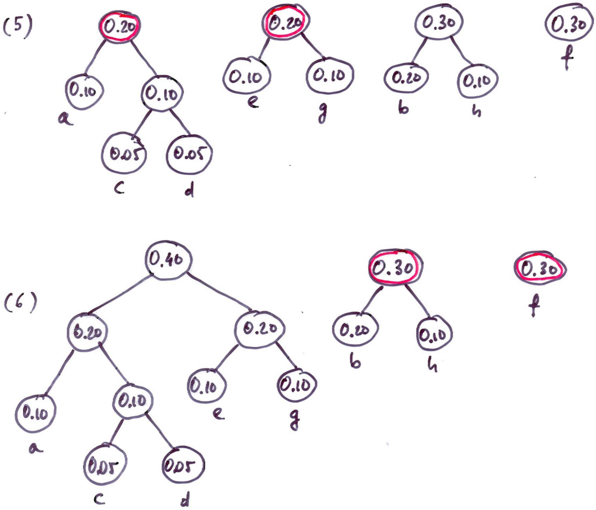
We start with the forest:



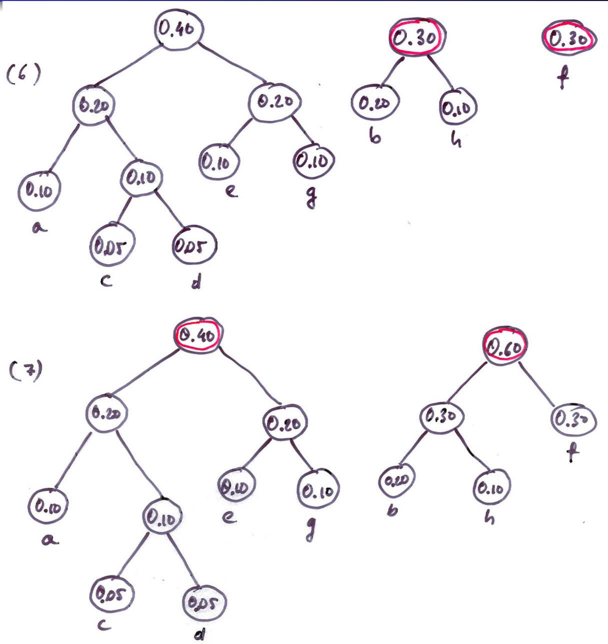
Huffman Algorithm: An Example



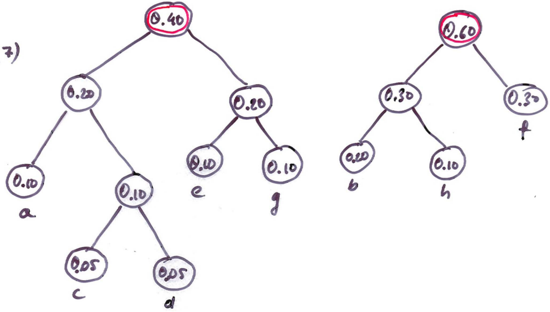
Huffman Algorithm: An Example



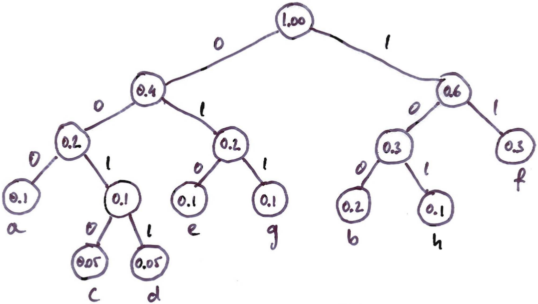
Huffman Algorithm: An Example



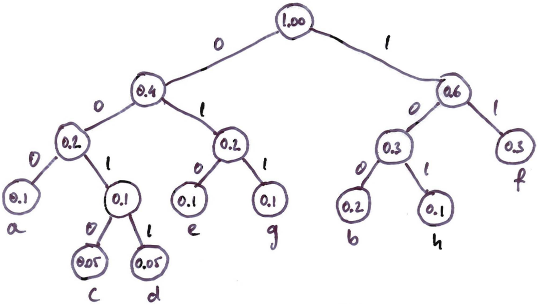
(7)


0.40
0.20
0.20
0.10
a
0.10
0.10
e
0.10
g
0.05
c
0.05
d
0.60
0.30
0.30
f
0.20
b
0.10
h

(8)


1.00
0
1
0.4
0.6
0.2
0.2
0.3
0.3
f
0.1
a
0.1
0.1
e
0.1
g
0.2
b
0.1
h
0.05
c
0.05
d

(8)



a	000
b	100
c	0010
d	0011
e	010
f	11
g	011
h	101

Huffman Algorithm: Properties

- Solutions are not unique but the value of

$$\sum_{i=1}^{n} g(a_i)length(c(a_i))$$

is always the same and it is the minimal value for all binary prefix codes.

- TIME COMPLEXITY
    - $k-1$ iterations, each consists of finding two minimal values and merging, can easily be done in $O(k)$¡ so totally $O(k^2)$.
    - if *priority queues implemented as heaps* (see CS/SE 2C03 course last year) are used, then finding two minimal elements is $O(\log k)$, merging is $O(1)$, so totally $O(k \log k)$.
- Ideas can be extended, see pages 175-177 in the textbook.

Optimal offline caching

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 evictions.

Ex. $k = 2$, initial cache = ab, requests: a, b, c, b, c, a, a.
Optimal eviction schedule. 2 evictions.

cache miss
(eviction)

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

Optimal offline caching: Greedy algorithms

LIFO / FIFO. Evict element brought in most (east) recently.

LRU. Evict element whose most recent access was earliest.

LFU. Evict element that was least frequently requested.

previous queries

	⋮						
	a	a	w	x	y	z	**FIFO: eject a**
	d	a	w	x	d	z	**LRU: eject d**
	a	a	w	x	d	z	
	b	a	b	x	d	z	
	c	a	b	c	d	z	
current cache	e	a	b	c	d	e	**LIFO: eject e**
cache miss (which item to eject?) →	g						
	b						
	e						
	d						
	⋮						

future queries

Optimal offline caching: farthest-in-future (clairvoyant algorithm)

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

current cache

cache miss
(which item to eject?)

a	a	b	c	d	e
f					
a					
b					
c					
e					
g					
b					
e					
d					
⋮					

future queries

FF: eject d

Optimal offline caching: farthest-in-future (clairvoyant algorithm)

Theorem

Farthest-in-future is optimal eviction schedule.

Proof.

Algorithm and theorem are intuitive; proof is subtle, uses the concept of *Reduced Eviction Schedules* see details on pages 135-136 in the textbook. □

Reduced eviction schedules

Definition

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

item inserted
when not requested

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	b	c
a	a	b	c

a reduced schedule

Farthest-in-future: analysis (sketch)

Lemma

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

Proof.

By induction on number of unreduced items. □

Lemma (Invariant Property)

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

Proof.

By induction j. □

Theorem

Farthest-in-future is optimal eviction algorithm.

Proof.

Follows directly from Invariant Property. □

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 Computer Science.

LIFO. Evict page brought in most recently.
LRU - Least-Recently-Used. Evict page whose most recent access was earliest (i.e. Farthest-in-Future with direction of time reversed!).
Farthest-in-Future is optimal offline eviction algorithm.

- Provides basis for understanding and analyzing online algorithms.
- Randomized version of LRU is efficient (k-competitive)
- LIFO is arbitrarily bad.