COMP 6651 / Winter 2022
Dr. B. Jaumard

Greedy Algorithms

January 21, 2022

Outline

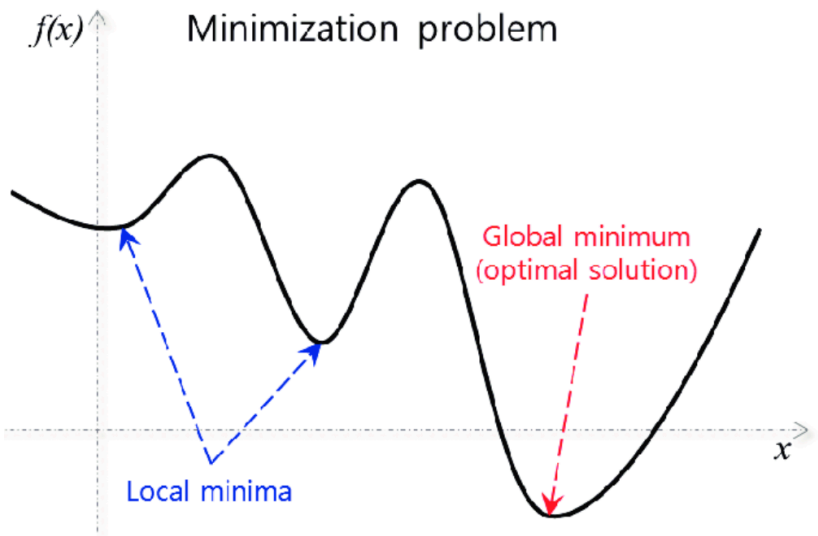
Heuristic vs. Metaheuristic vs. Exact Algorithm (1/3)

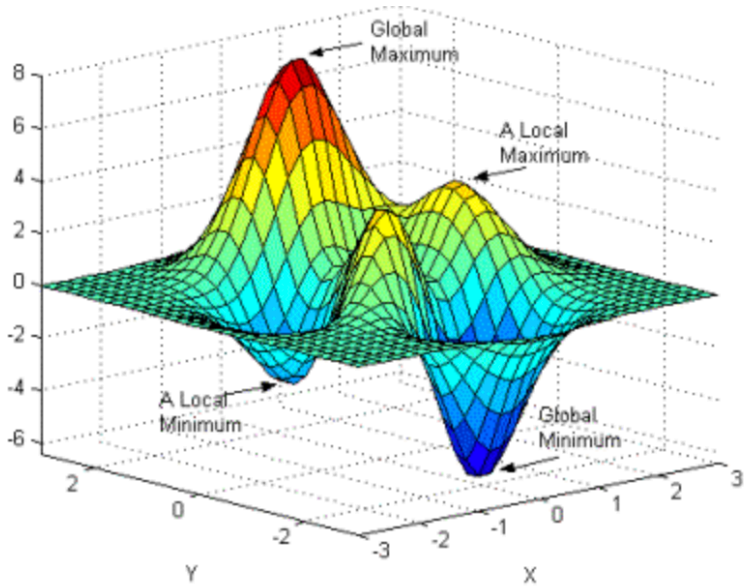
Optimization Problem

Optimize an objective function (z) subject to a set of constraints.

- A **feasible solution** is a vector that satisfies all the constraints.
- A **unique optimal value** ($z^\star$), but possibly **several optimal solutions** $x^\star$, $x^{\star\star}$, etc
- A feasible solution $x^\star$ is optimal if:
  $\forall$ feasible x, $\quad z(x)$ is not better than $z^\star = z(x^\star)$
- **Local optimum** of an optimization problem
  - Solution that is optimal within a neighbooring set of solutions.
  - This is in contrast to a global optimum. Locality of an optimum depends on the neighboorhood structure.

Heuristic vs. Metaheuristic vs. Exact Algorithm (2/3)





Heuristic vs. Metaheuristic vs. Exact Algorithm (3/3)

- **Heuristic**
  - Method that comes rapidly to a solution expected to be close to the best possible answer **or** the best solution we can get in a reasonable amount of time
  - Heuristics are often based on "rules of thumb", educated guesses, intuitive judgments or simply common sense
- **Metaheuristic** $\rightsquigarrow$ for solving difficult computational problems
  - Combines user-given black-box procedures, usually heuristics themselves
  - Metaheuristics are applied to problems for which there is no satisfactory problem-specific algorithm or heuristic.

Problem Types

Decision Problem

A problem with a "yes" or "no" answer

Example: Given a graph G, nodes v_s, v_t, is there a path from v_s to v_t in G?

Search Problem

A computational problem that requires identifying a solution from some, possibly infinite, solution space (set of possible solutions)

Example: Given a graph G, nodes v_s, v_t, find a $v_s - v_t$ path.

Optimization Problem

Find a best solution among all solutions for the input.

Example: Given a graph G, nodes v_s, v_t, find a shortest $v_s - v_t$ path.

Greedy Algorithms (1/2)

- Algorithms for optimization problems go through a sequence of steps, with a set of choices at each step.
- A **Greedy Algorithm** always makes the choice that looks best at the moment.
- It makes a locally optimal choice, in the hope that this choice leads to a globally optimal solution.

Greedy Algorithms (2/2)

- Do not always find optimal solutions, but for many problems they do.
- Greedy algorithms are quite powerful and work well for a wide range of problems
  - Minimum spanning tree algorithms
  - Shortest paths from a single source
  - Chvatal's greedy heuristic for set covering
  - ...

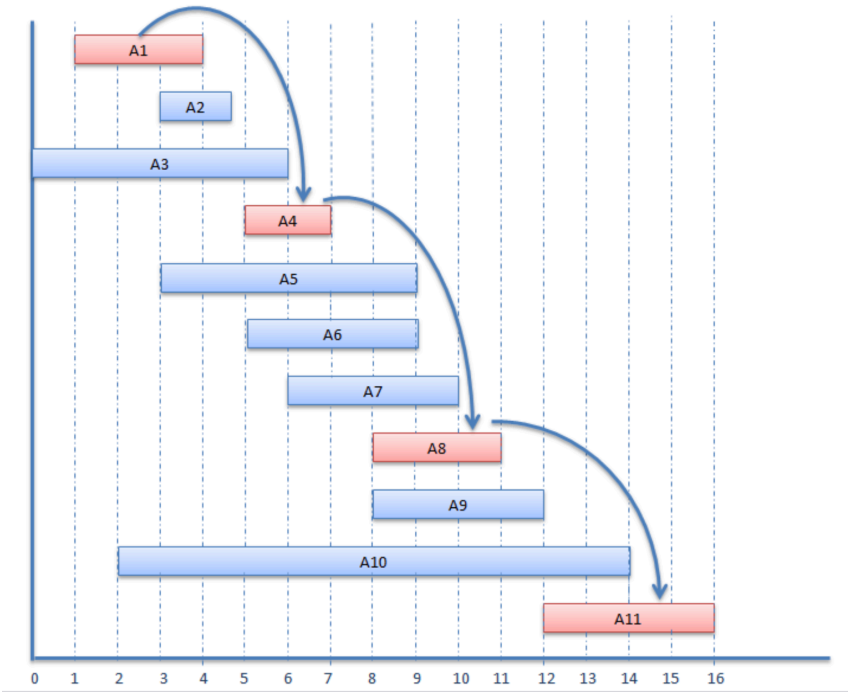
Activity Selection Problem

Scheduling Activities with a Common Resource (1/2)

- Problem: Scheduling several competing activities that require exclusive use of a common resource.
- Goal: Selecting a maximum-size set of mutually compatible activities.

Scheduling Activities with a Common Resource (2/2)

- Set $S = \{a_1, a_2, ..., a_n\}$ of n activities, that wish to use a resource which can be used by only one activity at a time.
- Activity $a_i \hookrightarrow$ start time s_i, finish time f_i, where $0 \leq s_i < f_i < \infty$
- a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not **overlap**, i.e., if $s_i \geq f_j$ **or** $s_j \geq f_i$
- The **activity-selection problem** is to select a maximum-size subset of **mutually compatible** activities.



Scheduling activities: An Example

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Activities have been sorted monotonically increasing order of finish times.

- Subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities, but it is **not a maximum subset** of compatible activities
- Subset $\{a_1, a_4, a_8, a_{11}\}$ is a larger subset of mutually compatible activities, that is maximum.
- Another maximum subset is $\{a_2, a_4, a_9, a_{11}\}$

An iterative greedy algorithm

GREEDY-ACTIVITY-SELECTOR(s,f)

```
1  n ← length[s]
2  A ← {a_1}
3  i ← 1
4  for m ← 2 to n
5     do if s_m ≥ f_i
6        then A ← A ∪ {a_m}
7             i ← m
8  return A
```

The GREEDY-ACTIVITY-SELECTOR algorithm provides an optimum solution

The GREEDY-ACTIVITY-SELECTOR Algorithm provides an Optimum Solution

Sort the intervals according to the f_i's. Denote the intervals in the sorted order by $I_1, I_2, ..., I_n$ (i.e., f_1 is the minimum among the right endpoints).

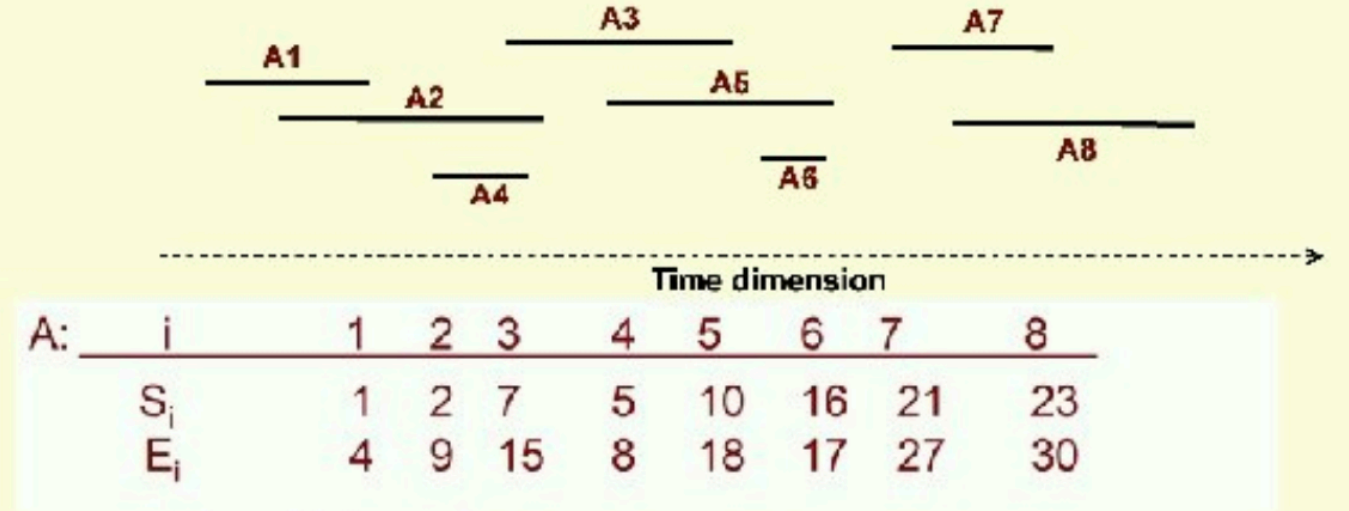
Claim: There is an optimum solution that includes I_1.

How to prove it. We can convert any optimal solution (S') to the solution of the greedy algorithm

Idea.

- Compare the activities in S' and S from left-to-right
- If they match in the selected activity $\rightsquigarrow$ skip
- if they do not match
  - Replace the activity in S' by that in S because the one in S finishes first

Example



A: i	1	2	3	4	5	6	7	8
S_i	1	2	7	5	10	16	21	23
E_i	4	9	15	8	18	17	27	30

- Greedy solution $S = \{a_1, a_4, a_6, a_7\}$
- Greedy solution $S' = \{a_1, a_4, a_5, a_8\}$
- a_5 in S' can be replaced by a_6 from S (finishes earlier)
- a_8 in S' can be replaced by a_7 from S (finishes earlier)

By these replacements, we save time
Greedy solution is one of the optimal solutions.

Huffman codes: Prefix codes

Fixed length vs. Variable length codes

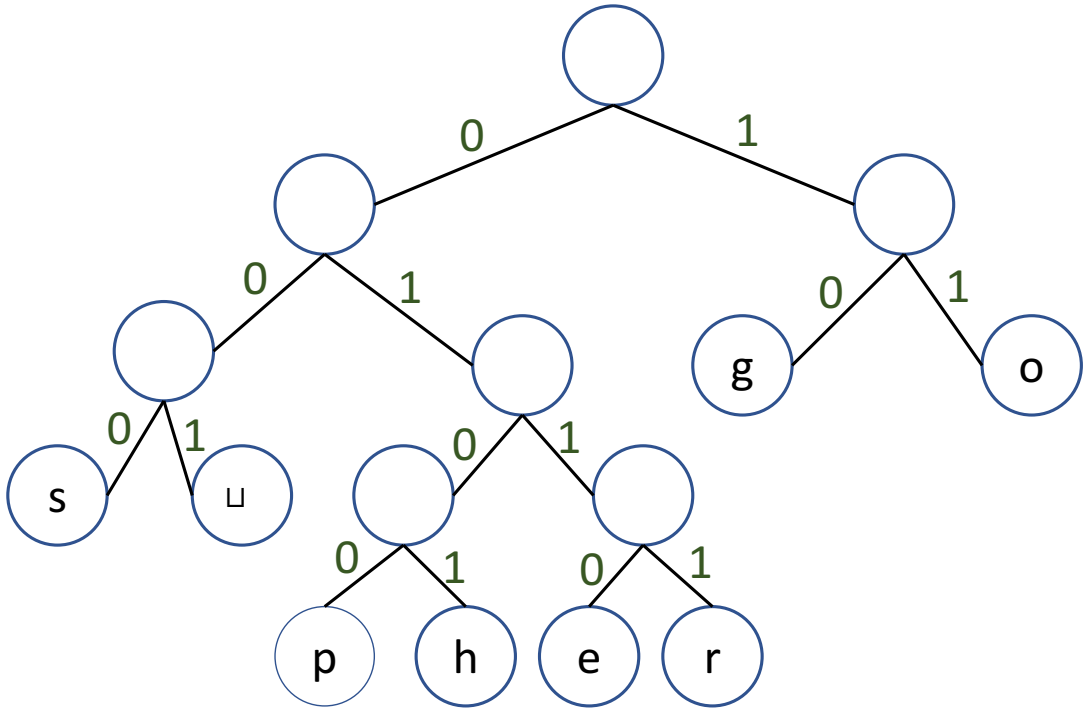
Instead of representing each character by a **fixed-length code** (7 or 8 bits typically),
most frequently used characters are given very **short code** while **less frequently** used characters are given **longer codes** ... **variable-length code**

To allow easy decoding of files written in variable length codes, we use **prefix-codes** in which no code is a prefix of another code.
(otherwise we would not know when a character ends)

100, 1001 are not prefix codes.
0, 10, 110, 1110, 11110 is prefix code.

Coding Example

char	binary
g	10
o	11
p	0100
h	0101
e	0110
r	0111
s	000
u	001



Using this coding, "go go gophers" is encoded (spaces wouldn't appear in the bitstream) as:

10 11 001 10 11 001 10 11 0100 0101 0110 0111 000

Decoding Example

The character-encoding induced by the tree can be used to decode a stream of bits as well as encode a string into a stream of bits.

0101011001110010000100010101100111011000110110110000001010101100111011 0

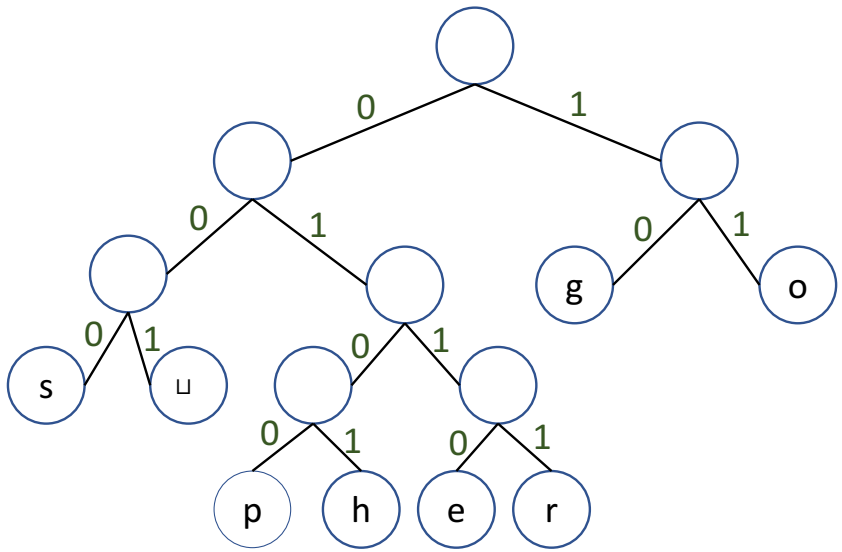
To decode the stream, start at the root of the encoding tree, and follow a left-branch for a 0, a right branch for a 1.

When you reach a leaf, write the character stored at the leaf, and start again at the top of the tree.

To start, the bits are 010101100111. This yields left-right-left-right to the letter 'h', followed (starting again at the root) with left-right-right-left to the letter 'e', followed by left-right-right-right to the letter 'r'.

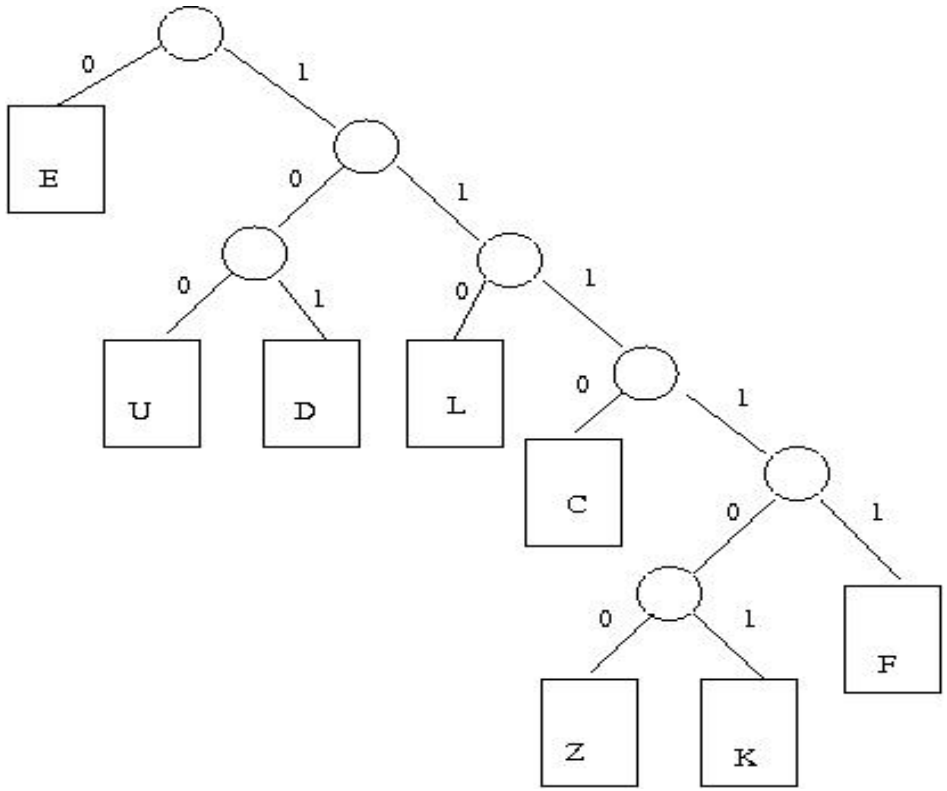
Continuing until all the bits are processed yields

her sphere goes here



Prefix codes

Prefix codes can be obtained from full binary trees in which the characters are leaves. Code = path to the character.



$E = 0, U = 100, D = 101, L = 110, F = 11111$ etc.

Constructing a Huffman Code

Build a coding tree T that optimizes the compression based on the known frequency of characters

Algorithm Huffman(C)

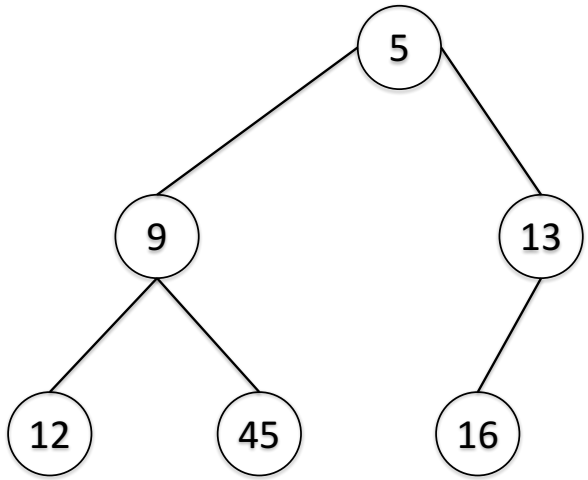
Input: A set $C = (c_1, \ldots, c_n)$ of n characters, each with weight/frequency $f[c]$
Output: A coding tree, T, for C, with minimum total path weight
Initialize a priority queue Q

For each character c in C **do**
  Create a single-node binary tree T storing c
  Insert T into Q with key $f[c]$

While size of $Q > 1$ **do**
  $f[x] \leftarrow$ EXTRACT-MIN(Q)
  $f[y] \leftarrow$ EXTRACT-MIN(Q)
  Allocate a new node z ; $f[z] \leftarrow f[x] + f[y]$
  Create a new binary tree T,
    with root z, left subtree $T_{f[x]}$ and right subtree $T_{f[y]}$
  INSERT($Q, f[z]$), i.e., insert T into Q with key $f[z]$
return EXTRACT-MIN(Q)

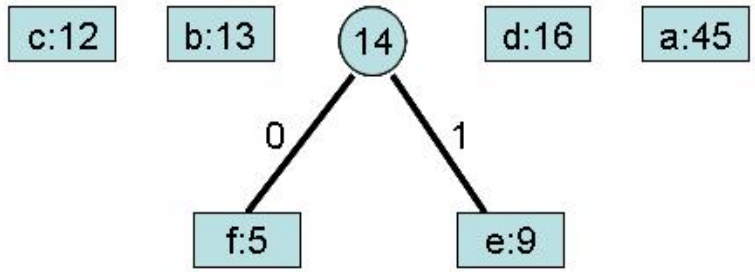
Huffman : An Example (1/6)

min-priority queue Q with set of characters C sorted w.r.t. frequencies $f[c], c \in C$.



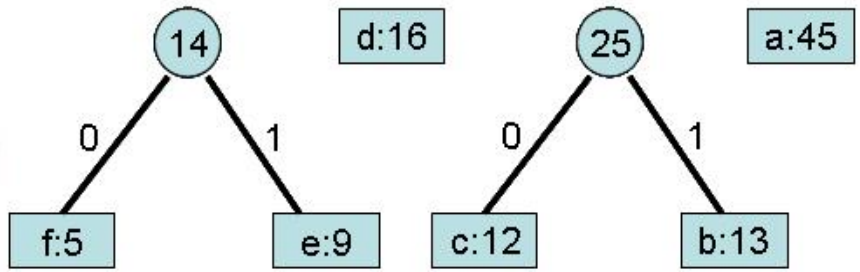
Huffman : An Example (2/6)

Iteration 1



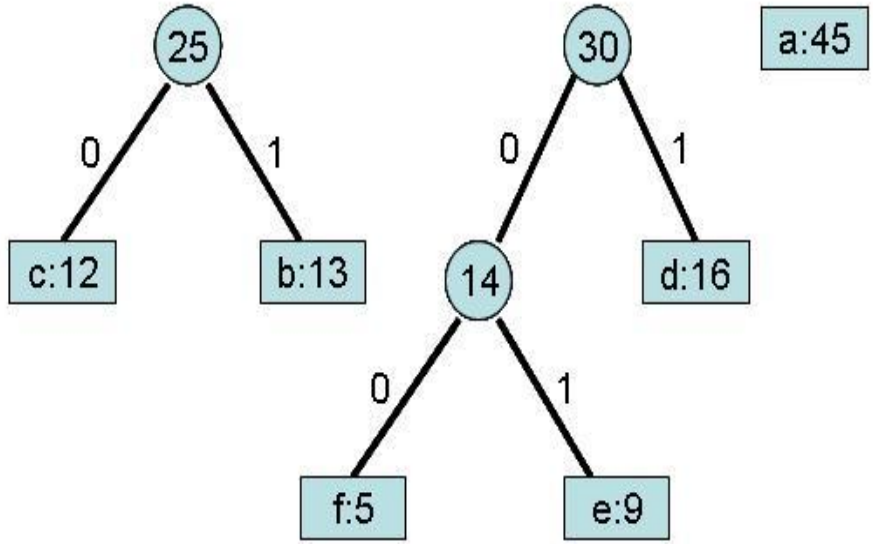
Huffman : An Example (3/6)

Iteration 2



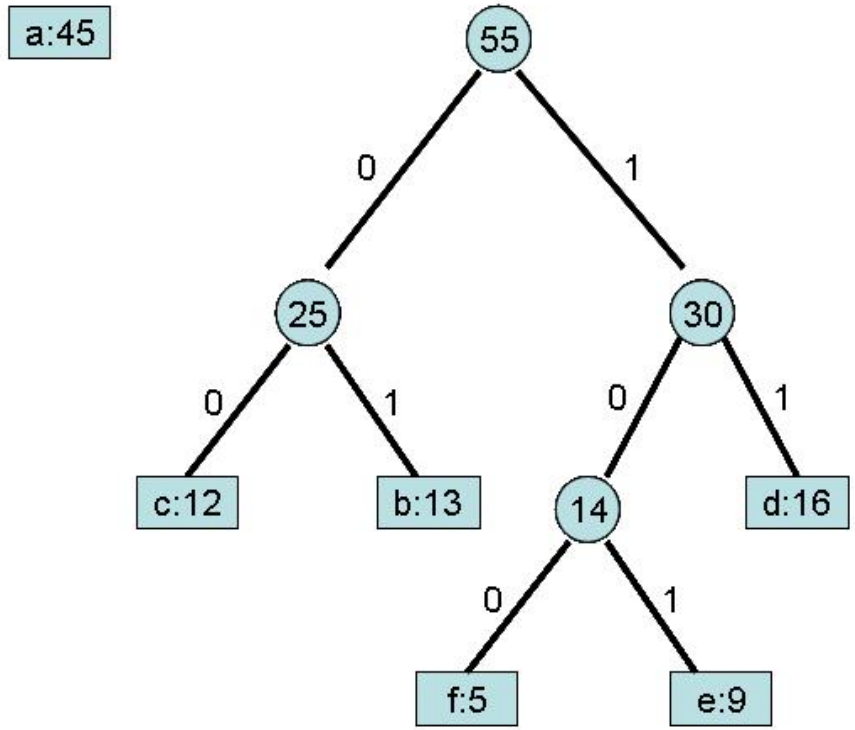
Huffman : An Example (4/6)

Iteration 3



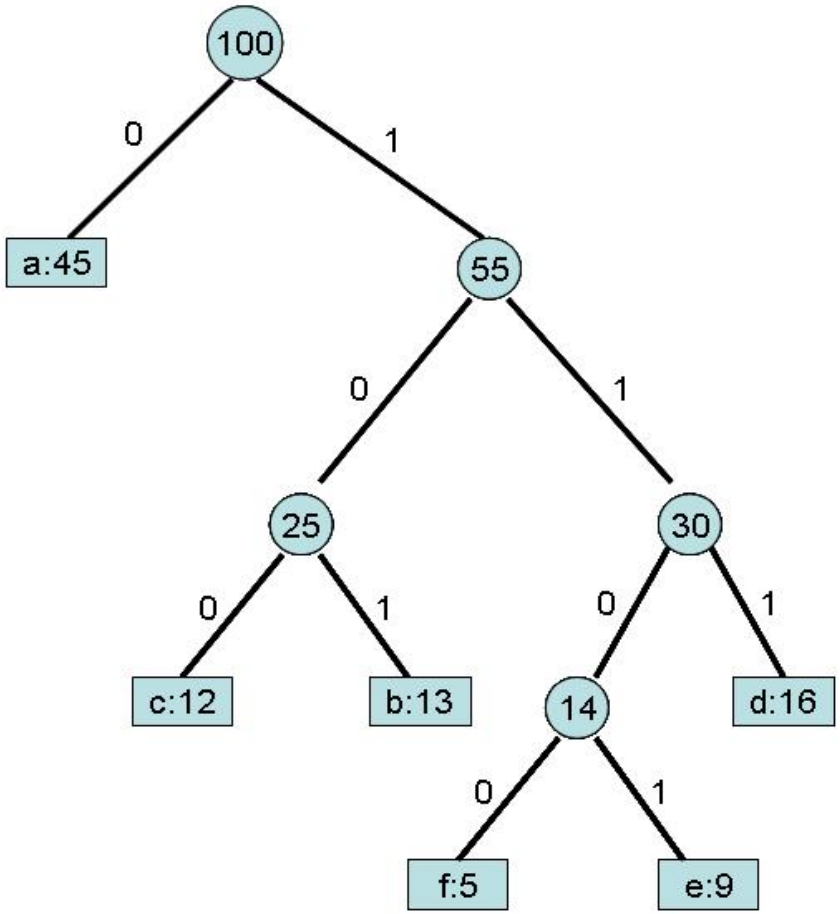
Huffman : An Example (5/6)

Iteration 4



Huffman : An Example (6/6)

Iteration $5 = n - 1$.
End of algorithm.
$a \leftarrow 0$,
$b \leftarrow 101$,
$c \leftarrow 100$,
$d \leftarrow 111$,
$e \leftarrow 1101$,
$f \leftarrow 1100$
Prefix codes



Correctness of Huffman's algorithm (1/2)

Lemma 1

Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code in which the codewords for x and y have the same length and differ only in the last bit.

Meaning: x and y are leaves of the same internal node.
Lemma 1 shows the **greedy** choice of Huffman's algorithm, merging :

- at the beginning those two characters with lowest frequency,
- at each step the one that incurs the least cost.

Cost of tree T: $B(T) = \sum_{c \in C} f(c) d_T(c)$ where

$f(c)$: frequency of c,
$d_T(c)$: depth of c in the tree, i.e., the length of the codeword for c.

Correctness of Huffman's algorithm (2/2)

Lemma 2

Let C be a given alphabet with frequency $f[c]$ defined for each character $c \in C$. Let x and y be two characters with minimum frequency. Let $C' = C \setminus \{x, y\} \cup \{z\}$; define f for C' as for C, except that $f[z] = f[x] + f[y]$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

- Meaning: inductively the construction is correct.
- It also shows that the problem of constructing optimal prefix codes (Huffman's algorithm) has the **optimal-substructure property**.

Memory Caching

Problem Statement: Motivation

How to deal with a draconian librarian

You are working on a long term project and your draconian librarian will only allow you to have eight books checked out at once.

You know that you'll probably need more than this over the course of working on the paper, but at any point in time, you'd like to have ready access to the eight books that are most relevant at that time.

How should you decide which books to check out, and when should you return some in exchange of others, to minimize the number of times you have to exchange a book at the library?

Problem Statement: Memory Hierarchy and Caching

- Memory hierarchy: A Ubiquitous feature of computers. Data in the main memory of a processor can be accessed much more quickly than the data stored on its hard disk; but the disk has much more storage capacity.
  $\hookrightarrow$ Important to keep the most regularly used pieces of data in main memory, and go to disk as infrequently as possible.
- Other examples: on-chip caches, Web browser.
- A small amount of data that can be accessed very quickly
- A large amount of data that requires more time to access
- Which pieces of data to have close at hand?

Statement of the Optimal Caching Problem (1/2)

- A set U of n pieces of data stored in **main memory**
- A fast memory, the **cache** can hold $k < n$ pieces of data at one time
- Assumption: The cache initially holds some set of k items
- A sequence of data items $D = d_1, d_2, \cdots, d_m$ drawn from U is presented to us
  $\hookrightarrow$ the sequence of memory references we must process
- When processing them, we must decide at all times, which k items to keep in the memory

Statement of the Optimal Caching Problem (2/2)

Eviction schedule: When item d_i is presented

- very quick access if d_i is already in the cache;
- otherwise, it is required to bring d_i from main memory into the cache
  - if the cache is full, evict some other piece of data that is currently in the cache to make room for d_i.

↪ **an eviction**, we want to have as few as possible.

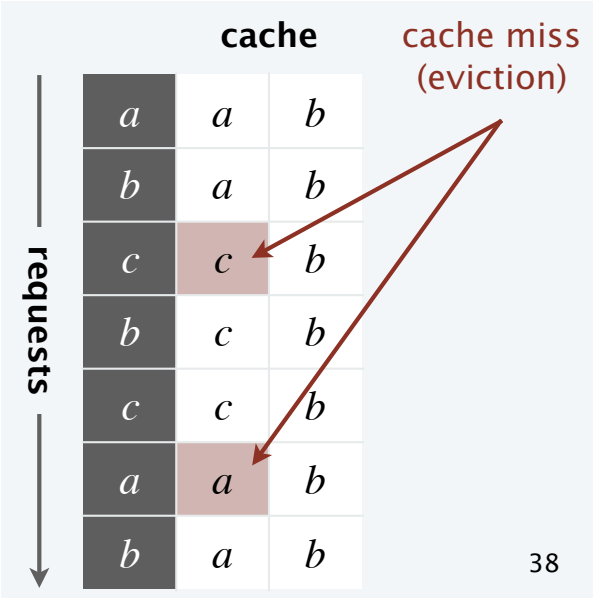
Goal. Eviction schedule that **minimizes the number of evictions**

An example

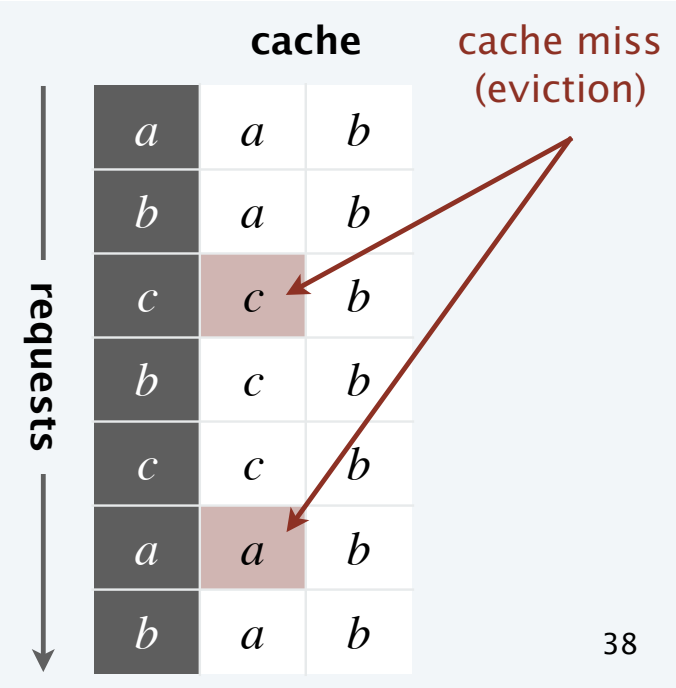
- Three items $\{a, b, c\}$; cache size $k = 2$
- Sequence of memory references we must process: a, b, c, b, c, a, b
- Initial cache content: $\{a, b\}$
- On the third item in the sequence:
  $\hookrightarrow$ Evict a and bring in c
- On the sixth item in the sequence:
  $\hookrightarrow$ Evict c and bring in a

$\hookrightarrow$ 2 evictions

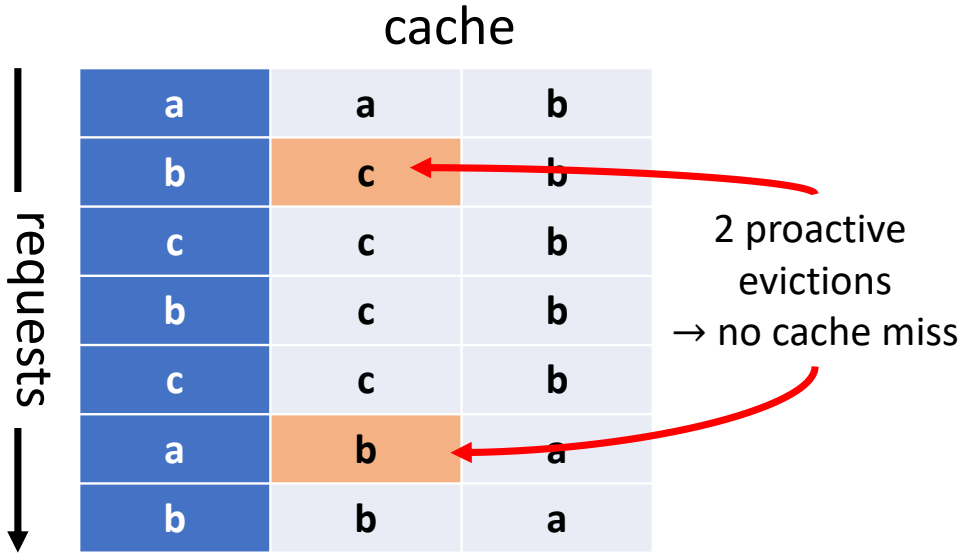
- Any eviction schedule needs at least 2 evictions.



An example - Cont'd



Do not confuse number of evictions with number of cache misses



Designing and Analyzing the Algorithm

In the 1960s, Les Belady showed that the following simple rule will always incur the minimum number of evictions

Farthest-in-Future Algorithm

When d_i needs to be brought into the cache,
evict the item that is needed the farthest into the future.

Objective: Minimum number of evictions

Farthest-in-Future Algorithm is optimal

More subtle than it first appears: Why evict the item that is needed farthest in time, as opposed, for example, to the one that will be used least frequently in the future?

An example

Sequence $\{a, b, c, d, a, d, e, a, d\}$

$k = 3$ items

$\{a, b, c\}$ in the cache

Farthest-in-Future Schedule S:
Evicts c on the fourth step and b on the seventh step.

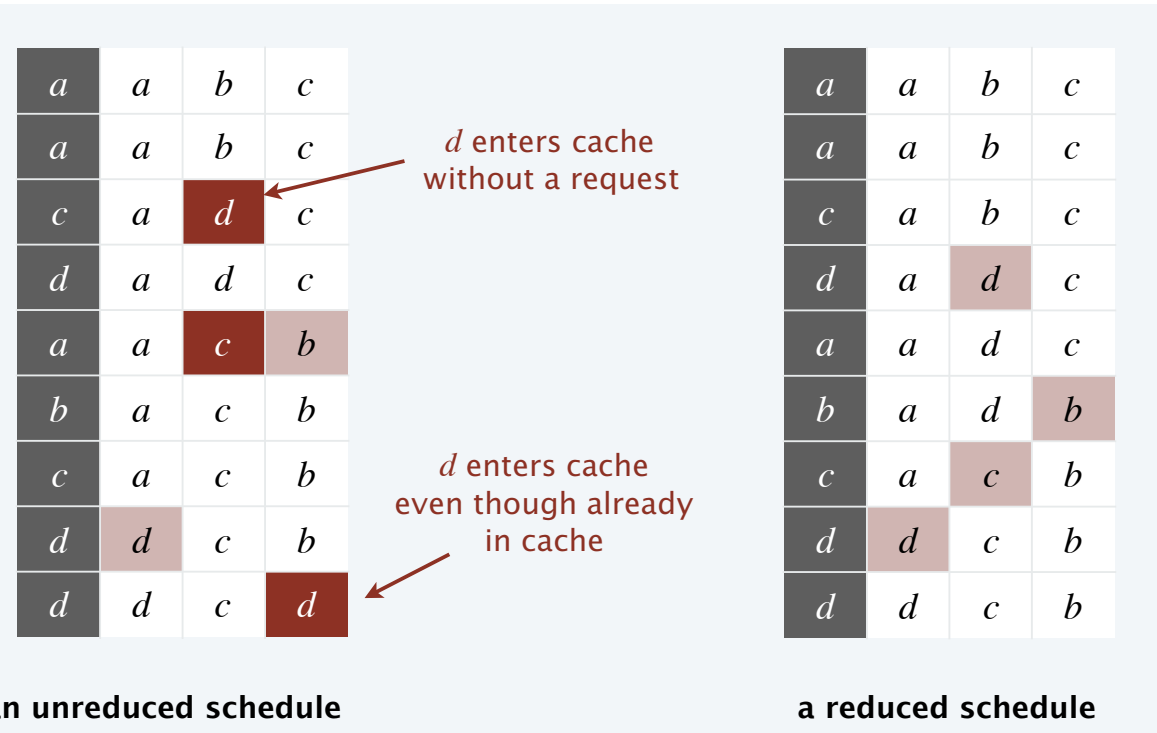
Another possible schedule S':
Evicts b on the fourth step and c on the seventh step.

Reduced Schedule

Reduced Schedule

A reduced schedule is a schedule that brings an item d into the cache in a step i if there is request to d in step i, and d is not already in the cache.

Sequence $\{a, a, c, d, a, b, c, d, d\}$; $k = 3$ items ; $\{a, b, c\}$ in the cache

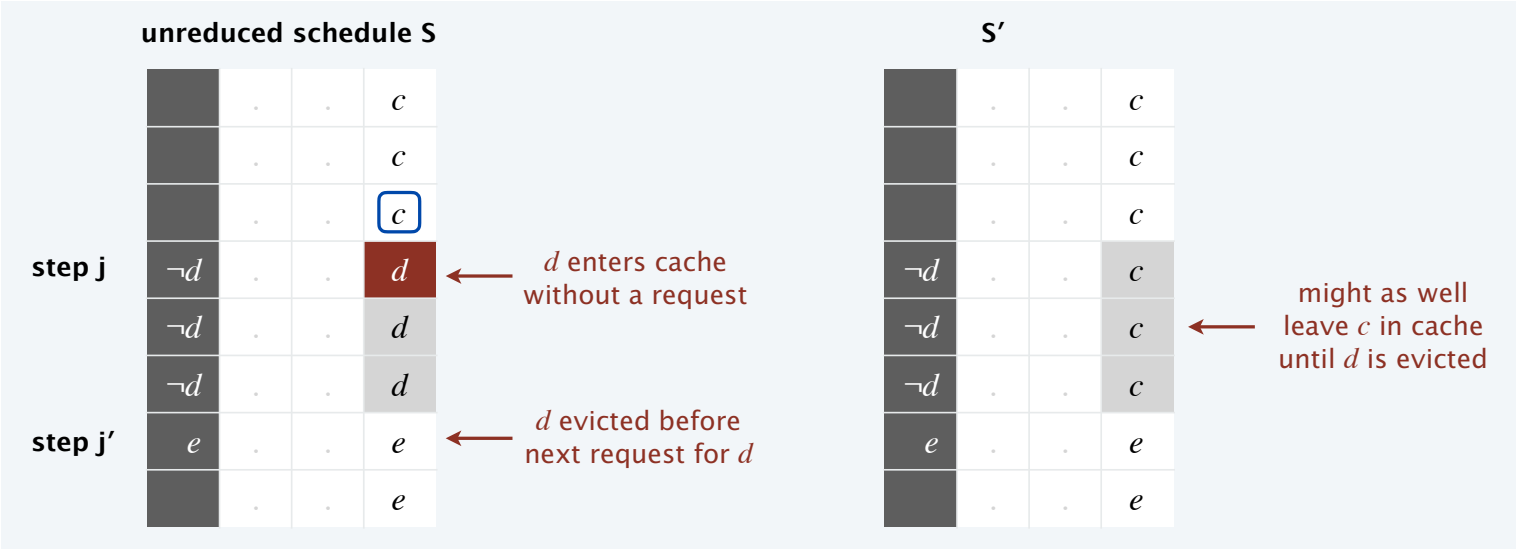


Reduced Schedule

Given any unreduced schedule S, it can be transformed into a reduced schedule S' with no more evictions

Proof by induction on the number of steps (j)

- Suppose S brings d into the cache in step j without a request
- Let c be the item S evicts when it brings d into the cache
- **Case 1a:** d evicted before next request for d

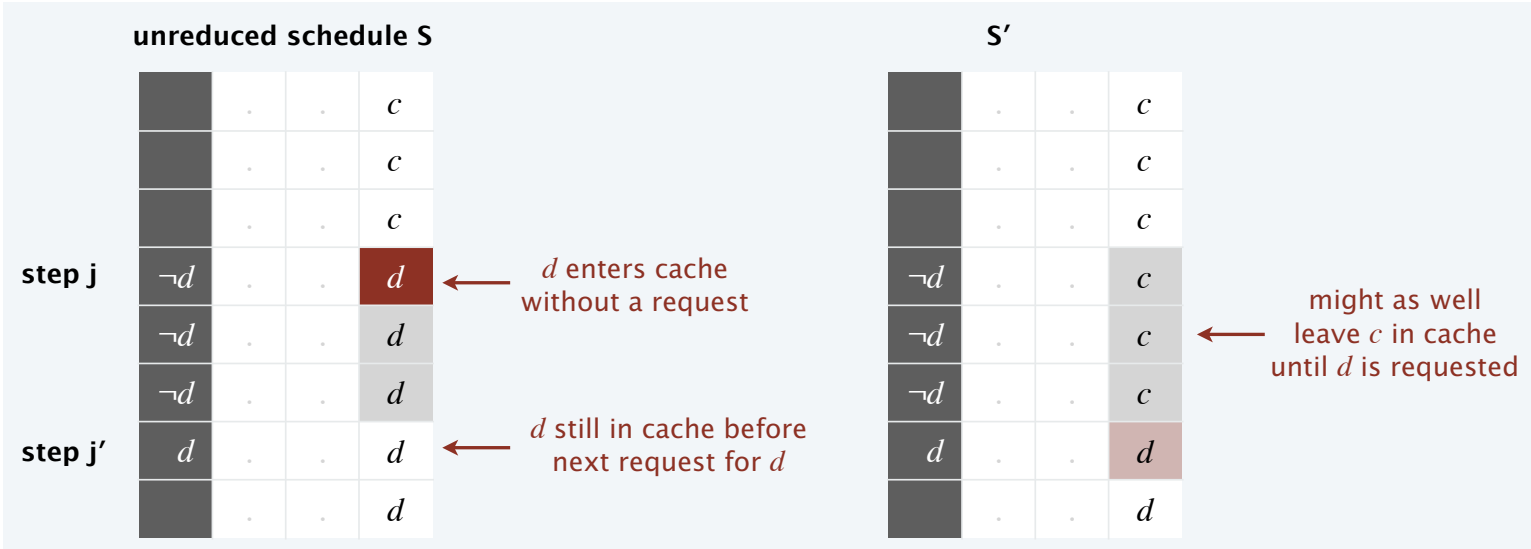


Reduced Schedule

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

Proof by induction on the number of steps (j)

- Suppose S brings d into the cache in step j without a request
- Let c be the item S evicts when it brings d into the cache
- **Case 1a:** d evicted before next request for d
- **Case 1b:** next request for d occurs before d is evicted.

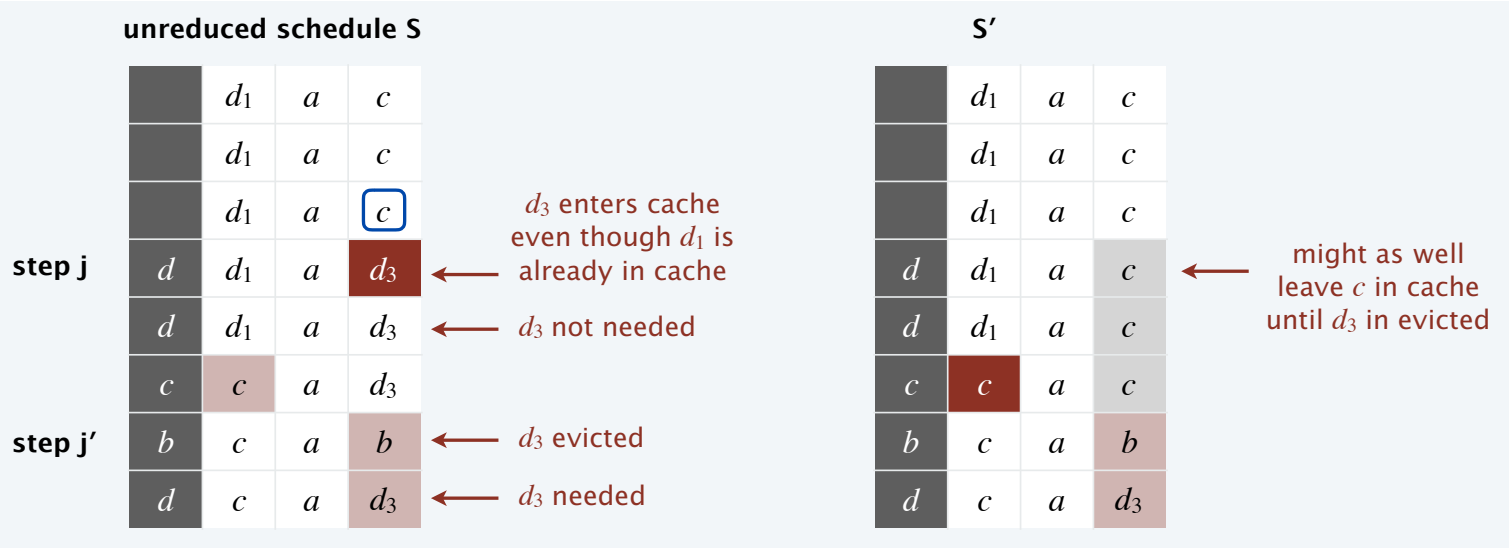


Reduced Schedule

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

Proof by induction on the number of steps (j)

- Suppose S brings d into the cache in step j even though d is in cache
- Let c be the item S evicts when it brings d into the cache.
- **Case 2a:** d evicted before it is needed

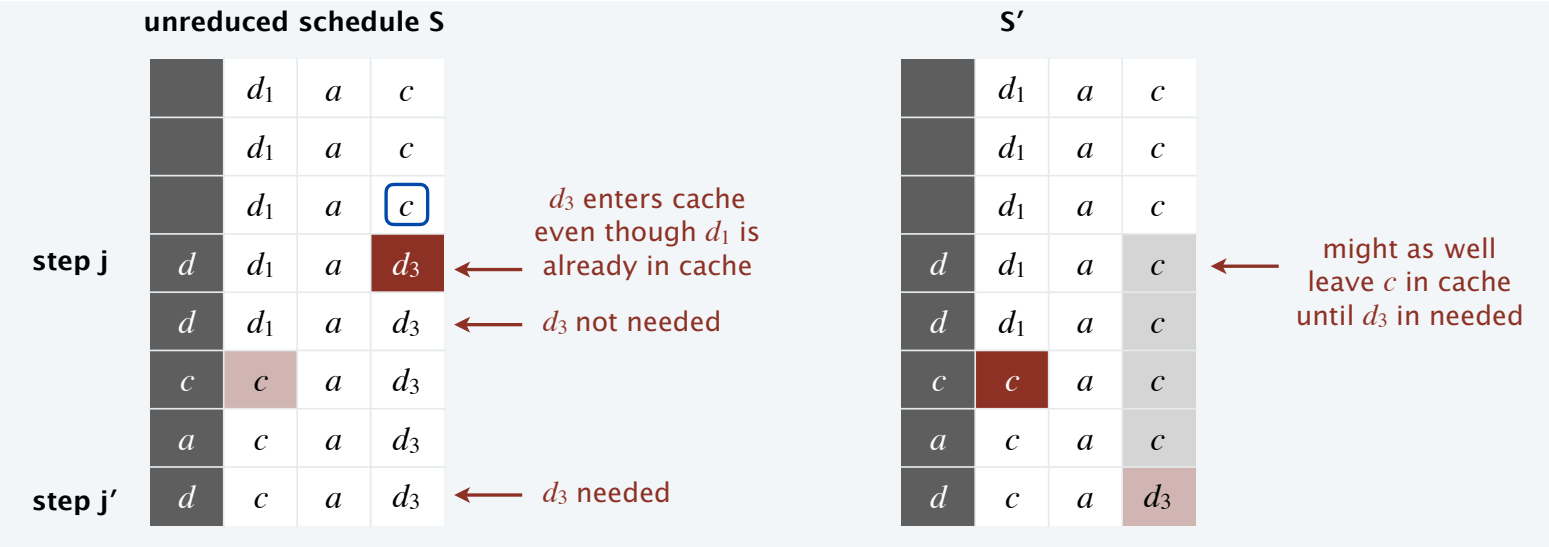


Reduced Schedule

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

Proof by induction on the number of steps (j)

- Suppose S brings d into the cache in step j even though d is in cache.
- Let c be the item S evicts when it brings d into the cache.
- **Case 2a:** d evicted before it is needed
- **Case 2b:** d needed before it is evicted.



Reduced Schedule

Proposition

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

Proof by induction on the number of steps (j)

- **Case 1:** S brings d into the cache in step j without a request. ✓
- **Case 2:** S brings d into the cache in step j even though d is in cache. ✓
- If multiple unreduced items in step j, apply each one in turn, dealing with Case 1 before Case 2.
  - resolving Case 1 might trigger Case 2

Note

Note that for any reduced schedule the number of items that are brought in is exactly the number of evictions

Proving the optimality of Farthest-in-Future (1/7)

- An arbitrary sequence D of memory references
- S_{FF} = schedule produced by the Farthest-in-Future algorithm
- $S^\star$ = a schedule that incurs the minimum possible number of evictions

Idea of the proof

Gradually transform the schedule $S^\star$ into the schedule S_{FF}, one eviction at a time, without increasing the number of evictions.

Proof by induction

Let S be a reduced schedule that makes the same eviction decisions as S_{FF} through the first j items in the sequence, for a number j. Then, there is a reduced schedule S' that makes the same eviction decisions as S_{FF} through the first $j+1$ items, and incurs no more evictions than S does.

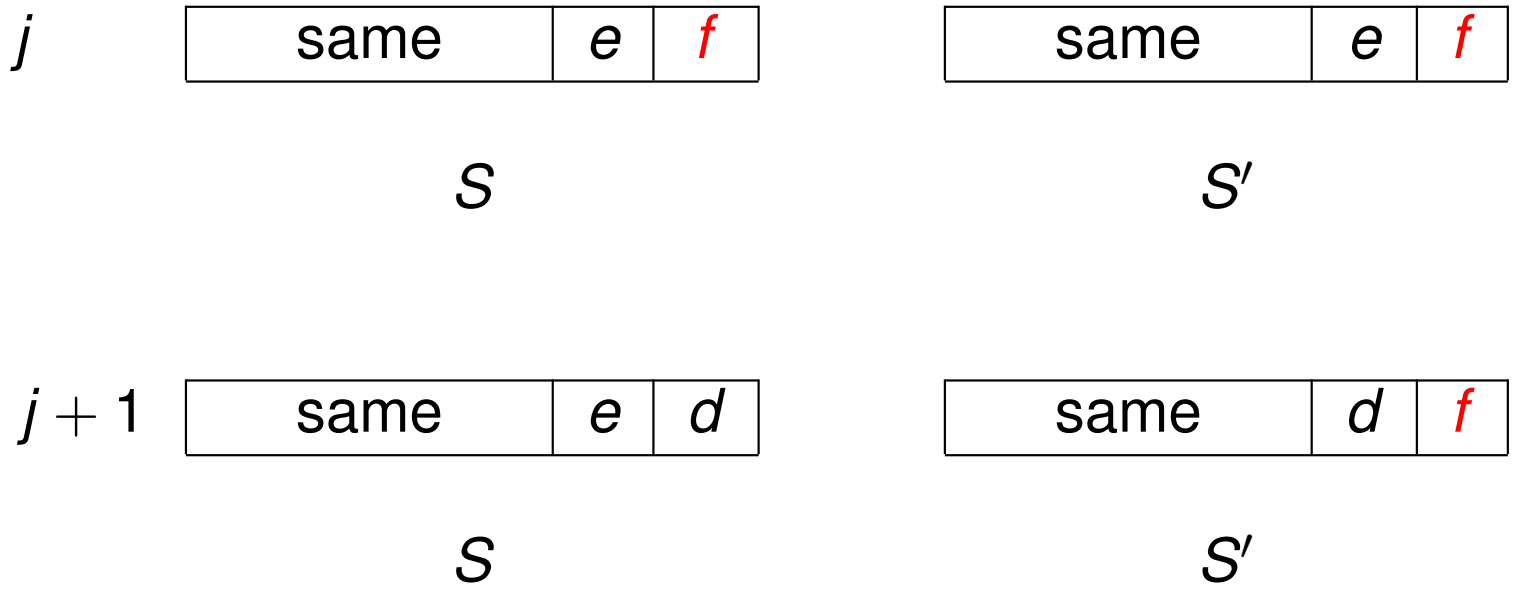
Proving the optimality of Farthest-in-Future (2/7)

Consider the $(j+1)^{\text{th}}$ request, to item $d = d_{j+1}$. Since S and S_{FF} have agreed up to the first j items, they have the same cache contents.

- **Case 1.** If d is in the cache for both, then no eviction is necessary (both schedules are reduced). Consequently S agrees with S_{FF} through step $j+1$, and we can set $S' = S$.
- If d needs to be brought into the cache
  - **Case 2.** S and S_{FF} both evict the same item to make room for d: We can again set $S' = S$.
  - **Case 3.** S evicts item f while S_{FF} evicts item $e \neq f$. Here, S and S_{FF} do not agree through step $j+1$ since S has e in cache while S_{FF} has f in cache. How to construct S'?

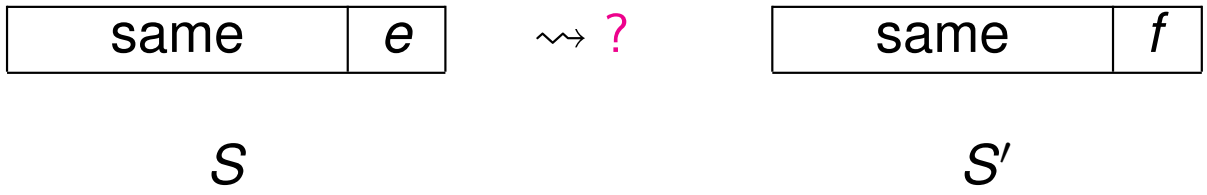
Proving the optimality of Farthest-in-Future (3/7)

Case 3. S evicts item f while S_{FF} evicts item $e \neq f$. Here, S and S_{FF} do not agree through step $j+1$ since S has e in cache while S_{FF} has f in cache. How to construct S'?



Proving the optimality of Farthest-in-Future (4/7)

Idea: Have S' try to get its cache back to the same state as S as quickly as possible, while not incurring unnecessary evictions. Once the caches are the same, we can finish the construction of S' by just having it behave like S.



From request $j+2$ onward, S' behaves exactly like S until one of the following cases happens for the first time.

- **Case 3a.** There is a request to item e. Cannot happen with Farthest-in-Future since there must be a request for f before e.
- **Case 3b.** There is a request to item $g \neq e, f$, g is not in the cache of S, and S evicts e to make room for it. Since S' and S only differ on f and e, it must be that g is not in the cache of S' either; so we can have S' evict f, and now the caches of S and S' are the same.

Proving the optimality of Farthest-in-Future (5/7)

- Case 3c. There is a request to f,and S evicts an item e'.
  - $e' = e$. We are all set: S' can simply access f from the cache, and after this step the caches of S and S' will be the same.
  - $e' \neq e$. We have S' evict e' as well, bring in e from main memory; this too results in S and S' having the same cache. BUT S' is no longer a reduced schedule.

    We further transform S' to its reduction $\overline{S}'$: this does not increase the number of items brought in by S', and it still agrees with S_{FF} through step $j + 1$.

Proving the optimality of Farthest-in-Future (6/7)

Next step of the proof.

- Begin with an optimal schedule $S^\star$
- Use induction (see previous page) to construct a schedule S_1 that agrees with S_{FF} through the first step
- Continue applying the induction for $j = 1, 2, 3, \ldots, n$, producing schedules S_j that agree with S_{FF} through the first j steps.

- Each schedule incurs no more evictions than the previous one
- By definition $S_m = S_{FF}$, since it agrees with it through the whole sequence.

Proof - Part 2

S_{FF} incurs no more evictions than any other schedule $S^\star$ and hence is optimal.

Proving the optimality of Farthest-in-Future (7/7)

Details of the proof can be found in:

J. Kleinberg and E. Tardos, *Algorithm Design*, Pearson, Addison-Wesley, 2005, Chapter 4, pp. 131 - 137.

Belady. L.A., A study of replacement algorithms for virtual storage computers, *IBM Systems Journal*, Vol. 5, Issue 2, (1966), 78-101.

Another reference of interest:

D.D. Sleator and R.E. Tarjan, Amortized efficiency of list update and paging rules, *Communications of the ACM* 28(2) (1985) 202-208.

Caching under Real Operating Operations

In applications, one generally must **make evictions on the fly** without knowledge of future requests.

Experimentally, the best caching algorithms under this requirement seem to be variants of the **Least-Recently-Used (LRU) Principle**:
Evict the item from the cache that was referenced longest ago.
It is Belady's algorithm with the direction of time reversed

- *Longest in the past rather than farthest in the future.*

Why is it effective? Applications generally exhibit locality of reference: a running program will generally keep accessing the things it has just been accessing.
$\hookrightarrow$ One wants to keep the more recently referenced items in the cache.

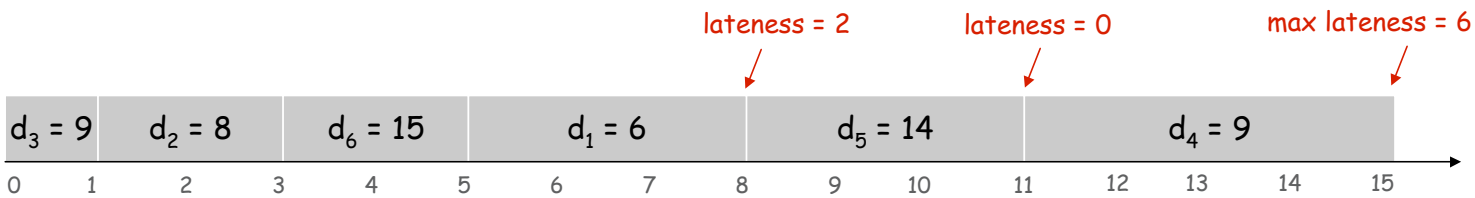
Sleator and Tarjan (1985) provided some theoretical analysis of the performance of LRU, **bounding the number of evictions** it incurs relative to Farthest-in-Future.

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 $L = \max_{j \in J} \ell_j$.

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



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 $d_j - t_j$

Greedy Template

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

	1	2
t_j	1	10
d_j	100	10

$\rightsquigarrow$ counterexample

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

	1	2
t_j	10	1
d_j	12	10

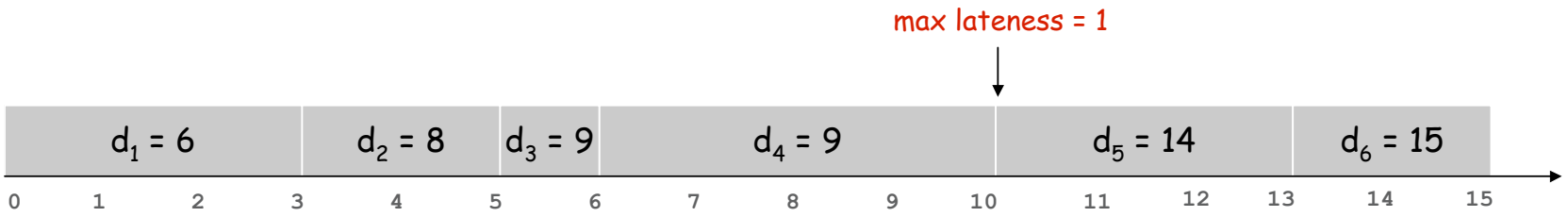
$\rightsquigarrow$ counterexample

Minimizing Lateness: Greedy Scheduling Algorithm

Greedy Scheduling Algorithm: Earliest Deadline First

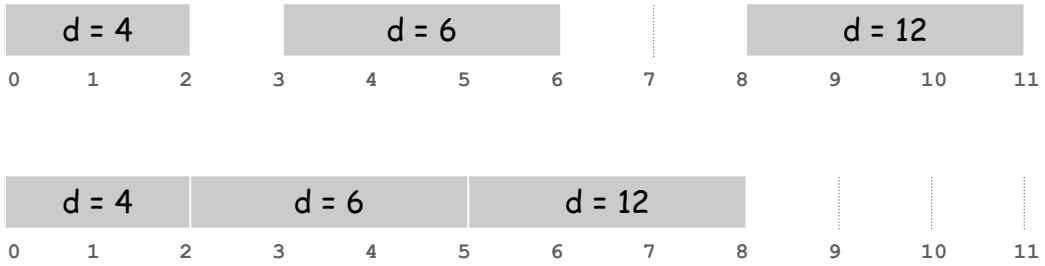
Sort the n jobs by deadline so that $d_1 \leq d_2 \leq \cdots \leq d_n$ (possible re-indexing of the jobs)

$t \leftarrow 0$
For $j = 1$ to n
    Assign job j to interval $[t, t + t_j]$
    $s_j \leftarrow t$; $f_j \leftarrow t + t_j$
    $t \leftarrow t + t_j$
Output: intervals $[s_j, f_j]$



Minimizing Lateness: No Iddle Time

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



Observation. The greedy schedule has no idle time.

Minimizing Lateness: Inversions

Definition

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.

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

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.

Inversion

Observation

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

Inversion for i and $j \rightsquigarrow d_j > d_i$, while i and j are not adjacent.

Case 1. $\mathbf{d_k} < d_i < d_j$ → inversion for k and j with j,k adjacent

Case 2. $d_i < \mathbf{d_k} < d_j$ → inversion for k and j with j,k adjacent

Case 3. $d_i < d_j < \mathbf{d_k}$ → inversion for k and i with i,k adjacent

Minimizing Lateness: Analysis of Greedy Schedule Algorithm (1/2)

Theorem

Let S be the schedule output by the Greedy Scheduling Algorithm (Slide 57).
S is an optimal schedule.

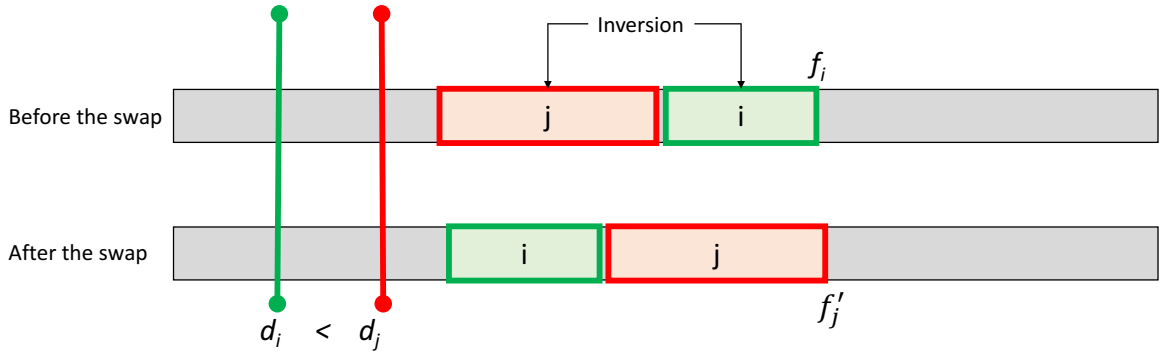
Proof. Let $S^\star$ to be an optimal schedule with the fewest number of inversions.

- Can assume $S^\star$ has no idle time.
- If $S^\star$ has no inversions, then $S = S^\star$.
- If $S^\star$ 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 $\rightsquigarrow$ proof on next slide
  - this contradicts definition of $S^\star$

Minimizing Lateness: Analysis of Greedy Schedule Algorithm (2/2)

Definition of an inversion

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.



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

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

$$\ell'_j = f'_j - d_j = f_i - d_j \leq f_i - d_i = \ell_i \quad \rightsquigarrow \max\{\ell'_i, \ell'_j\} \leq \ell_i \leq \max\{\ell_i, \ell_j\}.$$

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, ...

Suggested Exercises

- Course Pack: Exercises 6.1, 6.2, 6.3, 6.5.
- The four exercises (no available solution) in the next slides

CD burner's problem

We want to burn a music CD with as many songs as possible from our music collection.

Input: A set of n songs of durations $d_1, d_2, \ldots, d_n$ and a CD of capacity M.

Output: The largest set of songs that fit on the CD.

Suppose that in addition we require that the songs on the CD are by different artists. Give also an algorithm for this problem.

Find a subset J of 20 songs such that $\max_{i,j \in J} |d_i - d_j|$ to be minimized.

Google Interview Question for Software Developers

Input: A string that only contains lowercase.

Output: .You need to delete the repeated letters, only leave one, and try to make the lexicographical order of new string is smallest. For instance: **bcabc**.
You need to delete 1 **b** and 1 **c**, so you delete the first **b** and first **c**, the new string will be **abc**, which is smallest.

If you try to use a greedy algorithm to solve this problem, you must make sure that you could pass this case: **cbacdcbc**.
Answer is **acdb** and not **adcb**.

Minimum Number of Platforms Required for a Railway/Bus Station

Given arrival and departure times of all trains that reach a railway station, find the minimum number of platforms required for the railway station so that no train waits.
What is the time complexity of your heuristic?
We are given two arrays which represent arrival and departure times of trains that stop

Example

Input: $\texttt{arr}[] = \{9:00, 9:40, 9:50, 11:00, 15:00, 18:00\}$
$\texttt{dep}[] = \{9:10, 12:00, 11:20, 11:30, 19:00, 20:00\}$
Output: 3
There are at-most three trains at a time (time between 11:00 to 11:20)

Row of seats: Walmart Labs Interview Question for Senior Software Development Engineers

Consider *N* seats adjacent to each other. There is a group of people who are already seated in that row randomly, i.e., some are sitting together & some are scattered.
An occupied seat is marked with a character 'x' and an unoccupied seat is marked with a dot ('.')
Your target is to make the whole group sit together, i.e., next to each other, without having any vacant seat between them in such a way that the total number of jumps to move them is minimum.

Example

15 seats represented by the String (0, 1, 2, 3, ... , 14):
Input: ....*x*..*xx*...*x*..
Output: .....*xxxx*..... $\rightsquigarrow$ 5 jumps