CSC373

Week 2: Greedy Algorithms

Nisarg Shah

Recap

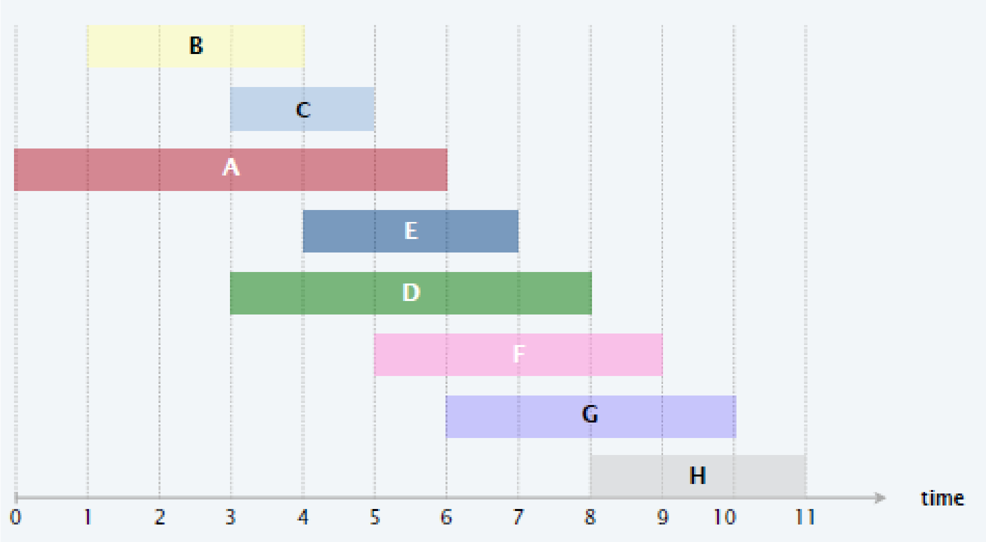
- Divide & Conquer
  - Master theorem
  - Counting inversions in $O(n \log n)$
  - Finding closest pair of points in $\mathbb{R}^2$ in $O(n \log n)$
  - Fast integer multiplication in $O\left(n^{\log_2 3}\right)$
  - Fast matrix multiplication in $O\left(n^{\log_2 7}\right)$
  - Finding k^{th} smallest element (in particular, median) in $O(n)$

Greedy Algorithms

- Greedy (also known as myopic) algorithm outline
  - We want to find a solution x that maximizes some objective function f
  - But the space of possible solutions x is too large
  - The solution x is typically composed of several parts (e.g. x may be a set, composed of its elements)
  - Instead of directly computing x…
    - Compute it one part at a time
    - Select the next part "greedily" to get maximum immediate benefit (this needs to be defined carefully for each problem)
    - May not be optimal because there is no foresight
    - But sometimes this can be optimal too!

Interval Scheduling

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



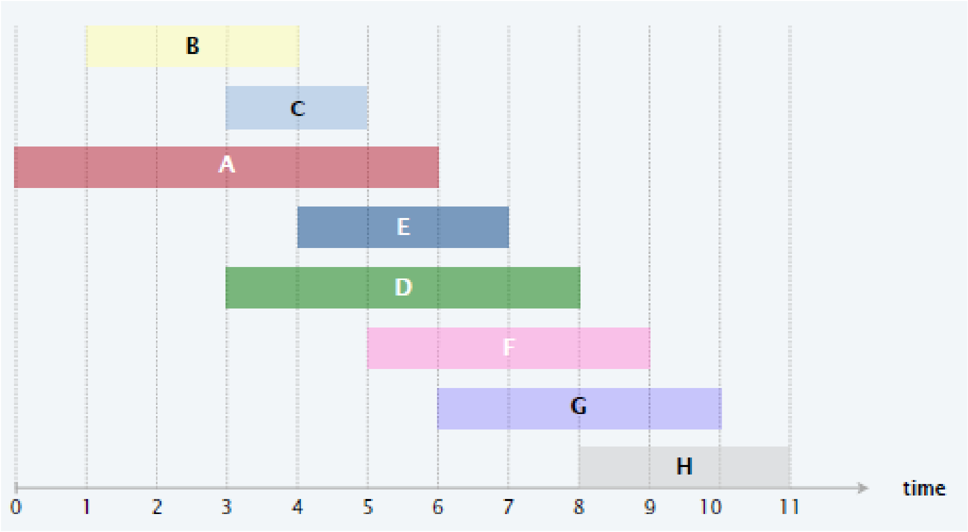
Interval Scheduling

- Greedy template
  - Consider jobs in some "natural" order
  - Take each job if it's compatible with the ones already chosen

- What order?
  - Earliest start time: ascending order of s_j
  - Earliest finish time: ascending order of f_j
  - Shortest interval: ascending order of $f_j - s_j$
  - Fewest conflicts: ascending order of c_j, where c_j is the number of remaining jobs that conflict with j

Example

- Earliest start time: ascending order of s_j
- Earliest finish time: ascending order of f_j
- Shortest interval: ascending order of $f_j - s_j$
- Fewest conflicts: ascending order of c_j, where c_j is the number of remaining jobs that conflict with j



Interval Scheduling

- Does it work?

Counterexamples for

earliest start time

shortest interval

fewest conflicts

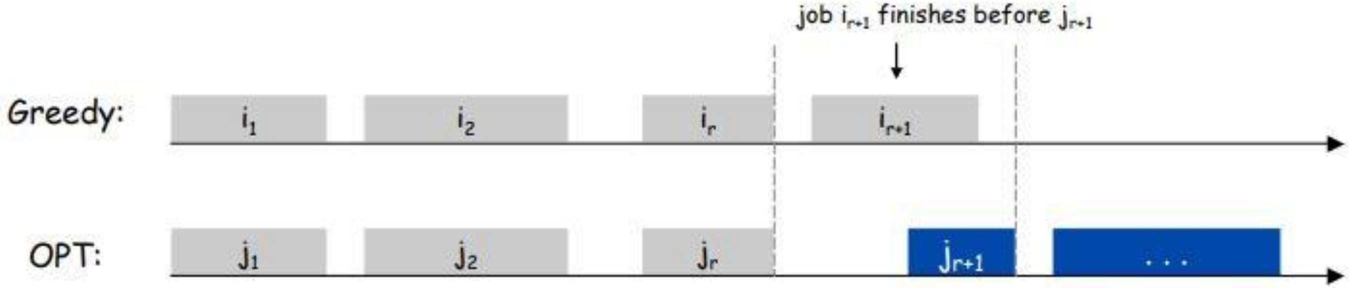
Interval Scheduling

- Implementing greedy with earliest finish time (EFT)
  - Sort jobs by finish time. Say $f_1 \leq f_2 \leq \cdots \leq f_n$
  - When deciding whether job j should be included, we need to check whether it's compatible with all previously added jobs
    - We only need to check if $s_j \geq f_{i^*}$, where i^* is the *last added job*
    - This is because for any jobs i added before i^*, $f_i \leq f_{i^*}$
    - So we can simply store and maintain the finish time of the last added job

  - Running time: $O(n \log n)$

Interval Scheduling

- Optimality of greedy with EFT
  - Suppose for contradiction that greedy is not optimal
  - Say greedy selects jobs $i_1, i_2, \ldots, i_k$ sorted by finish time
  - Consider the optimal solution $j_1, j_2, \ldots, j_m$ (also sorted by finish time) which matches greedy for as long as possible
    - That is, we want $j_1 = i_1, \ldots, j_r = i_r$ for greatest possible r



Interval Scheduling

Another standard method is induction

- Optimality of greedy with EFT
  - Both i_{r+1} and j_{r+1} were compatible with the previous selection ($i_1 = j_1, \ldots, i_r = j_r$)
  - Consider the solution $i_1, i_2, \ldots, i_r, i_{r+1}, j_{r+2}, \ldots, j_m$
    - It should still be feasible (since $f_{i_{r+1}} \leq f_{j_{r+1}}$)
    - It is still optimal
    - And it matches with greedy for one more step (contradiction!)

Two jobs compatible with both all future jobs and previous jobs, contradiction…

Interval Partitioning

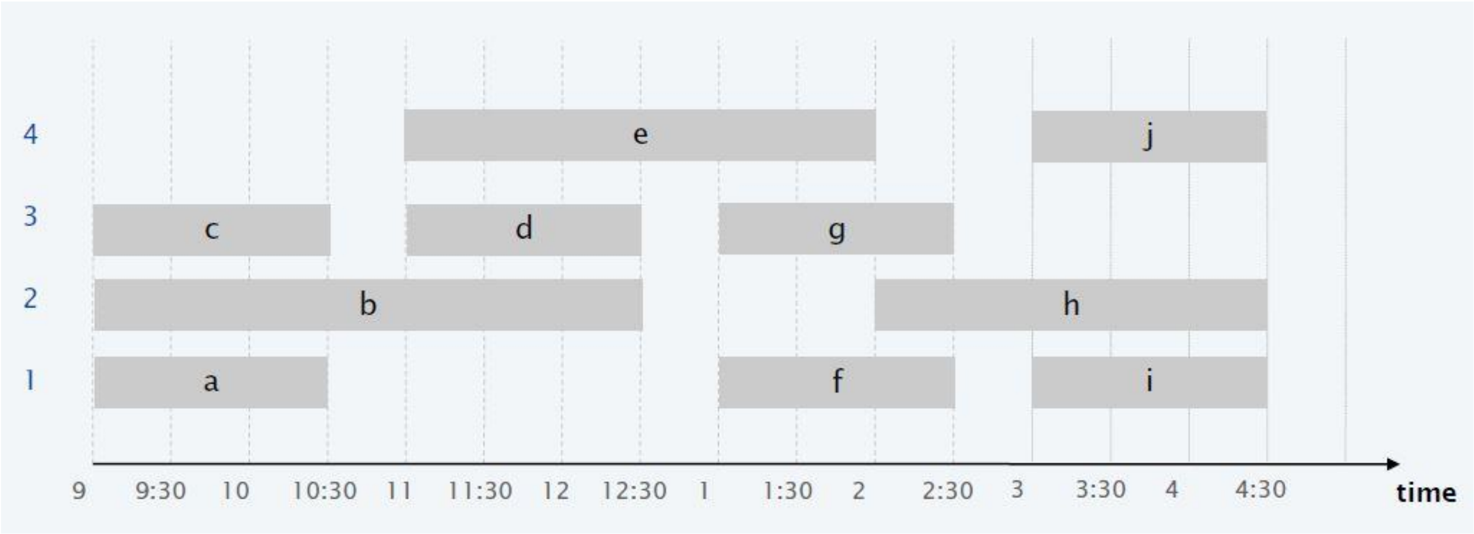
- Problem
  - Job j starts at time s_j and finishes at time f_j
  - Two jobs are compatible if they don't overlap
  - Goal: group jobs into fewest partitions such that jobs in the same partition are compatible

- One idea
  - Find the maximum compatible set using the previous greedy EFT algorithm, call it one partition, recurse on the remaining jobs.
  - Doesn't work (check by yourselves)

Interval Partitioning

- Think of scheduling lectures for various courses into as few classrooms as possible

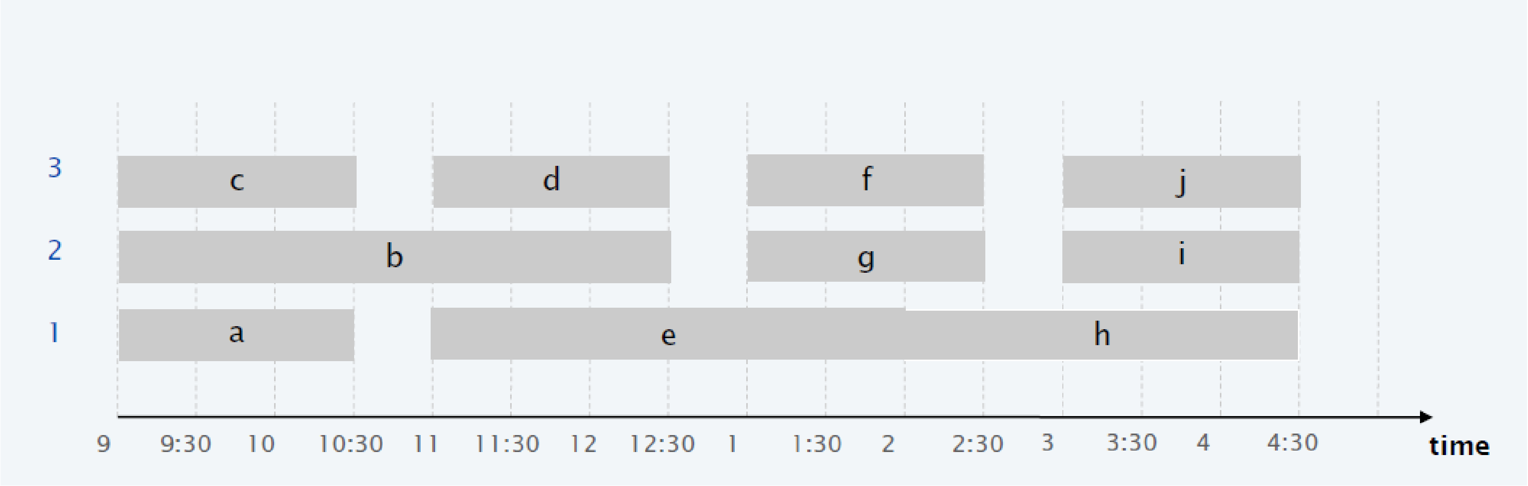
- This schedule uses 4 classrooms for scheduling 10 lectures



Interval Partitioning

- Think of scheduling lectures for various courses into as few classrooms as possible

- This schedule uses 3 classrooms for scheduling 10 lectures

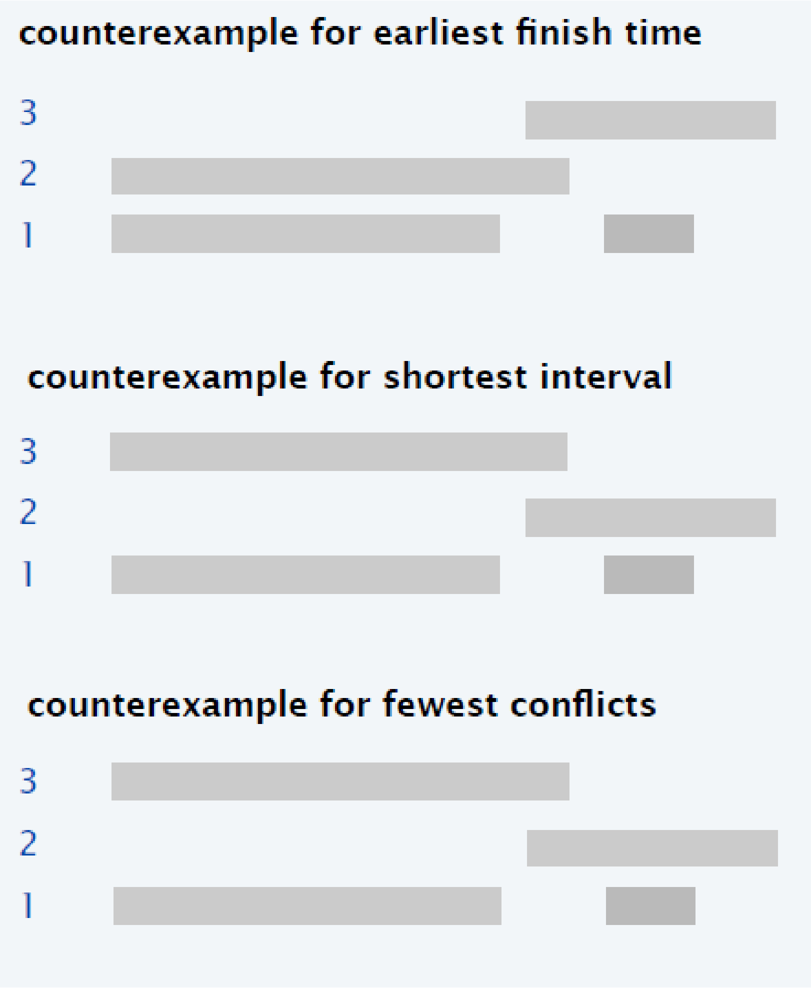


Interval Partitioning

- Let's go back to the greedy template!
  - Go through lectures in some "natural" order
  - Assign each lecture to a compatible classroom (which?), and create a new classroom if the lecture conflicts with every existing classroom

- Order of lectures?
  - Earliest start time: ascending order of s_j
  - Earliest finish time: ascending order of f_j
  - Shortest interval: ascending order of $f_j - s_j$
  - Fewest conflicts: ascending order of c_j, where c_j is the number of remaining jobs that conflict with j

Interval Partitioning



- At least when you assign each lecture to an arbitrary feasible classroom, three of these heuristics do not work.

- The fourth one works! (next slide)

Interval Partitioning

EARLIESTSTARTTIMEFIRST(n, s_1, s_2, …, s_n, f_1, f_2, …, 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 $d + 1$.

    $d \leftarrow d + 1$

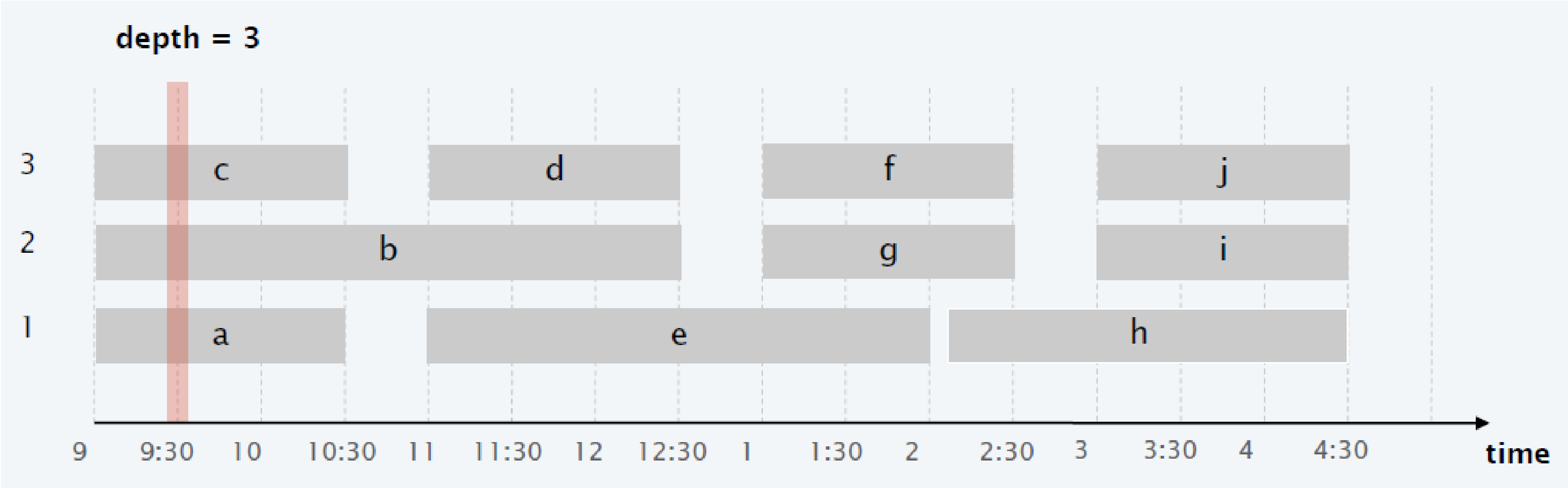
RETURN schedule.

Interval Partitioning

- Running time
  - Key step: check if the next lecture can be scheduled at some classroom
  - Store classrooms in a priority queue
    - key = finish time of its last lecture
  - Is lecture j compatible with some classroom?
    - Same as "Is s_j at least as large as the minimum key?"
    - If yes: add lecture j to classroom k with minimum key, and increase its key to f_j
    - Otherwise: create a new classroom, add lecture j, set key to f_j
  - $O(n)$ priority queue operations, $O(n \log n)$ time

Interval Partitioning

- Proof of optimality (lower bound)
  - # classrooms needed ≥ maximum "depth" at any point
    - depth = number of lectures running at that time
  - We now show that our greedy algorithm uses only these many classrooms!



Interval Partitioning

- Proof of optimality (upper bound)
  - Let d = # classrooms used by greedy
  - Classroom d was opened because there was a schedule j which was incompatible with some lectures already scheduled in each of $d - 1$ other classrooms
  - All these d lectures end after s_j
  - *Since we sorted by start time*, they all start at/before s_j
  - So at time s_j, we have d overlapping lectures
  - Hence, depth $\geq d$
  - So all schedules use $\geq d$ classrooms.
  - QED!

Interval Graphs

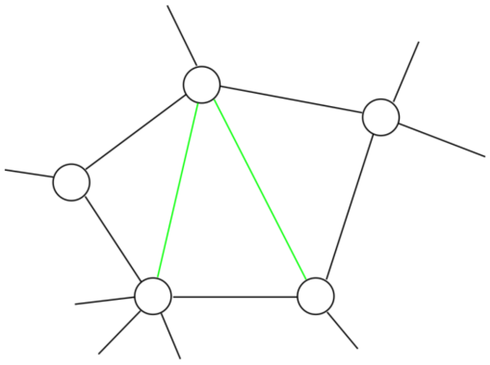
- Interval scheduling and interval partitioning can be seen as graph problems

- Input
  - Graph $G = (V, E)$
  - Vertices V = jobs/lectures
  - Edge $(i, j) \in E$ if jobs i and j are incompatible

- Interval scheduling = maximum independent set (MIS)
- Interval partitioning = graph colouring

Interval Graphs

- MIS and graph colouring are NP-hard for general graphs
- But they're efficiently solvable for interval graphs
  - Interval graphs = graphs which can be obtained from incompatibility of intervals
  - In fact, this holds even when we are not given an interval representation of the graph
- Can we extend this result further?
  - Yes! Chordal graphs
    - Every cycle with 4 or more vertices has a chord



Minimizing Lateness

- Problem
  - We have a single machine
  - Each job j requires t_j units of time and is due by time d_j
  - If it's scheduled to start at s_j, it will finish at $f_j = s_j + t_j$
  - Lateness: $\ell_j = \max\{0, f_j - d_j\}$
  - Goal: minimize the maximum lateness, $L = \max_j \ell_j$

- Contrast with interval scheduling
  - We can decide the start time
  - All jobs must be scheduled on a single machine

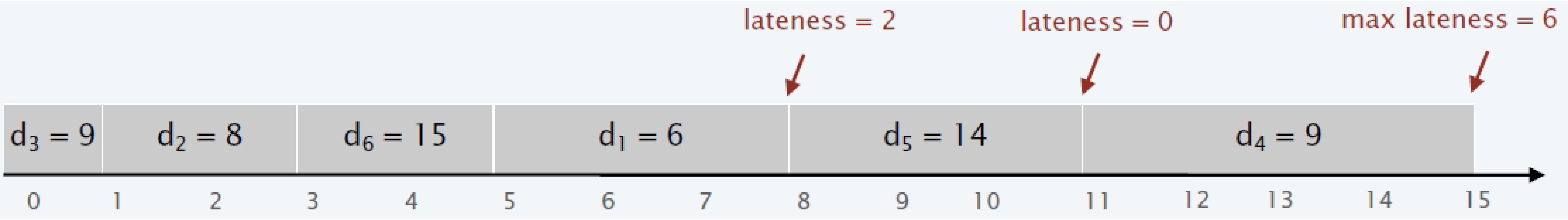
Minimizing Lateness

- Example

Input

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

An example schedule



Minimizing Lateness

- Let's go back to greedy template
  - Consider jobs one-by-one in some "natural" order
  - Schedule jobs in this order (nothing special to do here, since we have to schedule all jobs and there is only one machine available)

- Natural orders?
  - Shortest processing time first: ascending order of processing time t_j
  - Earliest deadline first: ascending order of due time d_j
  - Smallest slack first: ascending order of $d_j - t_j$

Minimizing Lateness

- Counterexamples

  - Shortest processing time first
    - Ascending order of processing time t_j

	1	2
t_j	1	10
d_j	100	10

  - Smallest slack first
    - Ascending order of $d_j - t_j$

	1	2
t_j	1	10
d_j	2	10

Minimizing Lateness

- By now, you should know what's coming…
- We'll prove that earliest deadline first works!

EarliestDeadlineFirst(n, t_1, t_2, …, t_n, d_1, d_2, …, d_n)

Sort n jobs so that $d_1 \leq d_2 \leq \ldots \leq d_n$.

$t \leftarrow 0$

For $j = 1$ to n

  Assign job j to interval $[t, t + t_j]$.

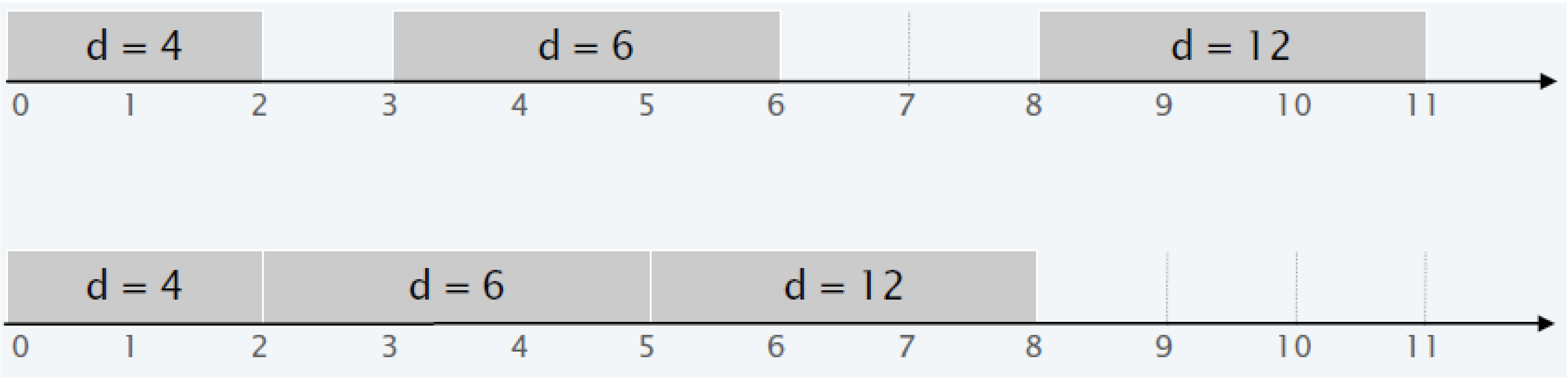
  $s_j \leftarrow t$; $f_j \leftarrow t + t_j$

  $t \leftarrow t + t_j$

Return intervals $[s_1, f_1]$, $[s_2, f_2]$, …, $[s_n, f_n]$.

Minimizing Lateness

- Observation 1
  - There is an optimal schedule with no idle time

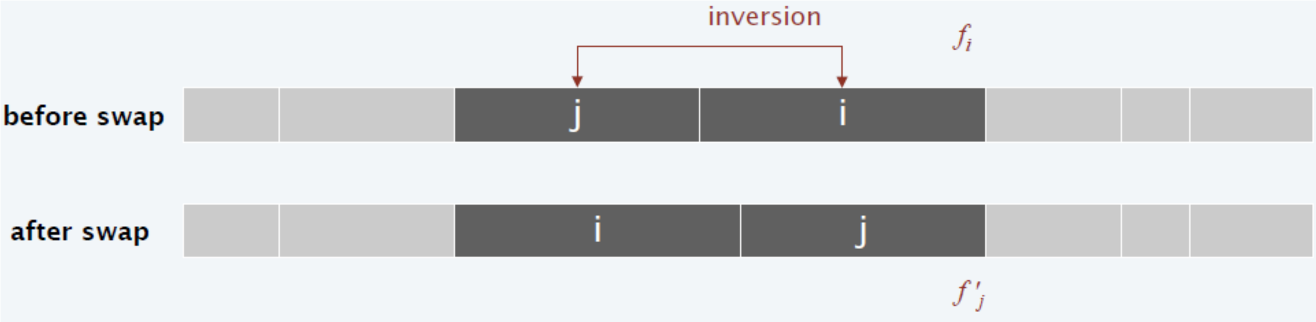


Minimizing Lateness

- Observation 2
  - Earliest deadline first has no idle time
- Let us define an "inversion"
  - (i, j) such that $d_i < d_j$ but j is scheduled before i
- Observation 3
  - By definition, earliest deadline first has no inversions
- Observation 4
  - If a schedule with no idle time has an inversion, it has a pair of inverted jobs scheduled consecutively

Minimizing Lateness

- Claim
  - Swapping adjacently scheduled inverted jobs doesn't increase lateness but reduces #inversions by one
- Proof
  - Let ℓ and ℓ' denote lateness before/after swap
  - Clearly, $\ell_k = \ell'_k$ for all $k \neq i, j$
  - Also, clearly, $\ell'_i \leq \ell_i$



Minimizing Lateness

- Claim
  - Swapping adjacently scheduled inverted jobs doesn't increase lateness but reduces #inversions by one
- Proof
  - $\ell'_j = f'_j - d_j = f_i - d_j \le f_i - d_i = \ell_i$
  - $L' = \max\left\{\ell'_i, \ell'_j, \max_{k \neq i,j} \ell'_k\right\} \le \max\left\{\ell_i, \ell_i, \max_{k \neq i,j} \ell_k\right\} \le L$

Minimizing Lateness

- Proof of optimality of earliest deadline first
  - Suppose for contradiction that it's not optimal
  - Consider an optimal schedule S^* which has fewest inversions among all optimal schedules
    - We can assume it has no idle time
    - If S^* has zero inversions, it's exactly earliest deadline first
    - So assume S^* has at least one inversion
    - So it must have an adjacent inversion (i, j)
    - But swapping these jobs doesn't increase lateness (so new schedule stays optimal) and reduces the number of inversions by 1
    - Contradiction given that S^* has fewest inversions among all optimal schedules.
    - QED!

Lossless Compression

- Problem
  - We have a document that is written using n distinct labels
  - Naïve encoding: represent each label using $k = \log n$ bits
  - If the document has length m, this uses $m \log n$ bits

- Say for English documents with no punctuations etc, we have $n = 26$, so we can use 5 bits.
  - $a = 00000$
  - $b = 00001$
  - $c = 00010$
  - $d = 00011$
  - …

Lossless Compression

- Is this optimal?
  - What if a, e, r, s are much more frequent in the document than x, q, z?
  - Can we assign shorter codes to more frequent letters?

- Say we assign…
  - $a = 0, b = 1, c = 01, \ldots$
  - See a problem?
    - What if we observe the encoding '01'?
    - Is it 'ab'? Or is it 'c'?

Lossless Compression

- To avoid conflicts, we need *prefix-free encoding*
  - ➢ Map each label x to a bit-string $c(x)$ such that for all distinct labels x and y, $c(x)$ is not a prefix of $c(y)$
  - ➢ Then it's impossible to have a scenario like this

$$\underbrace{\underbrace{\ldots\ldots\ldots\ldots\ldots}_{c(x)}\ldots\ldots\ldots\ldots}_{c(y)}$$

  - ➢ So we can read left to right, find the first point where it becomes a valid encoding, decode the label, and continue

Lossless Compression

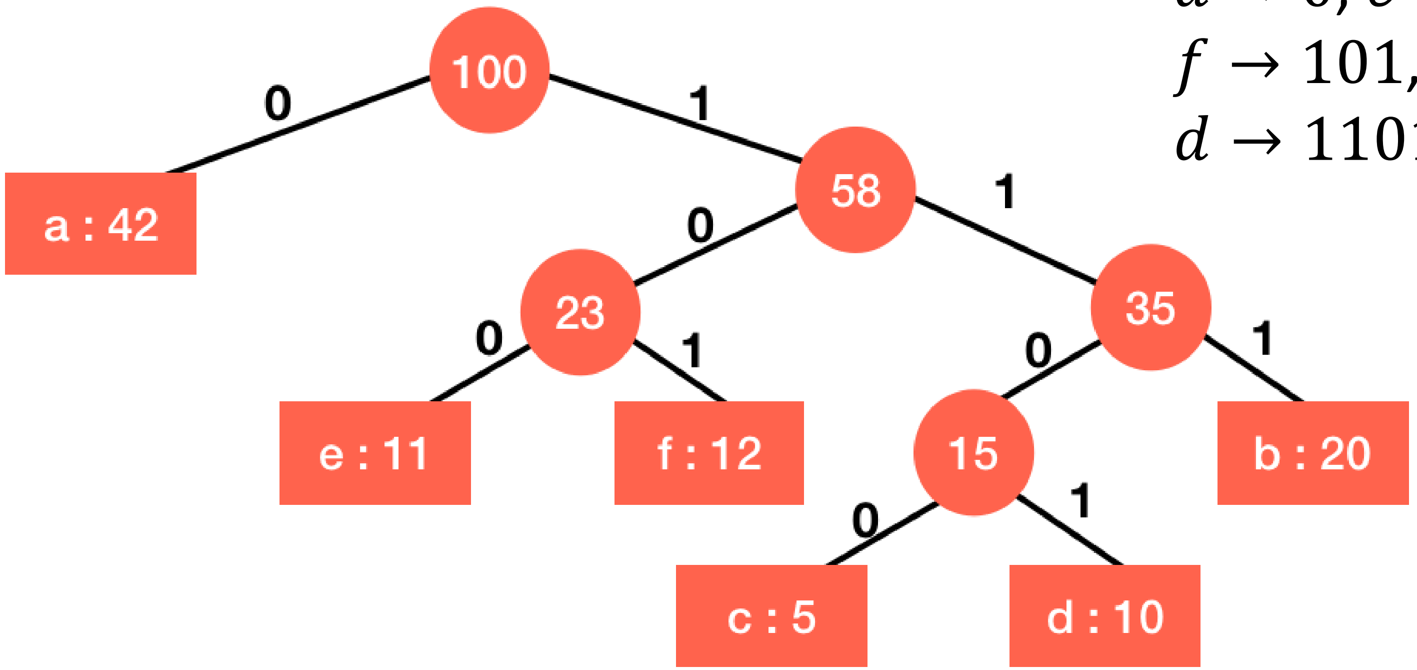
- Formal problem
  - Given n symbols and their frequencies $(w_1, \dots, w_n)$, find a prefix-free encoding with lengths $(\ell_1, \dots, \ell_n)$ assigned to the symbols which minimizes $\sum_{i=1}^{n} w_i \cdot \ell_i$
    - Note that $\sum_{i=1}^{n} w_i \cdot \ell_i$ is the length of the compressed document

- Example
  - $(w_a, w_b, w_c, w_d, w_e, w_f) = (42,20,5,10,11,12)$
  - No need to remember the numbers ☺

Lossless Compression

- Observation: prefix-free encoding = tree

$a \to 0$, $e \to 100$,
$f \to 101$, $c \to 1100$,
$d \to 1101$, $b \to 111$

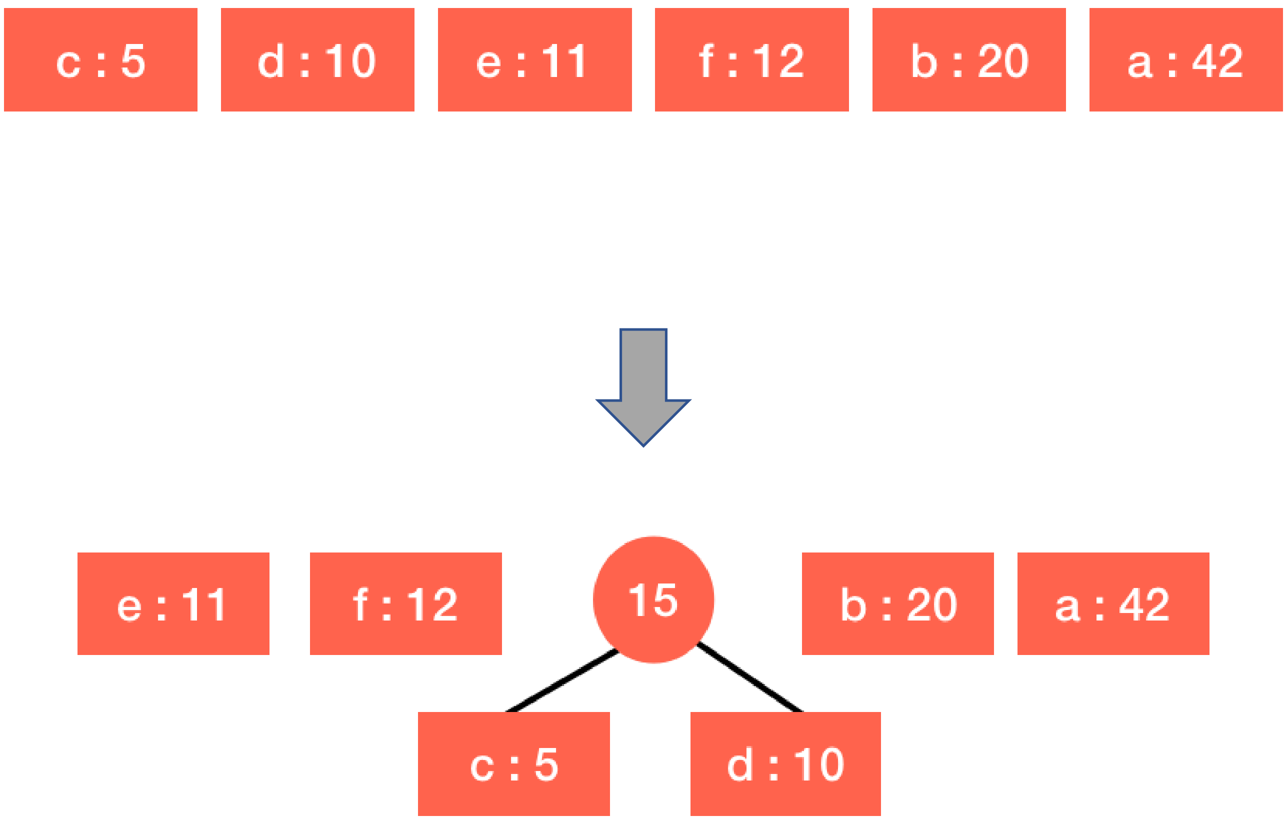


Lossless Compression

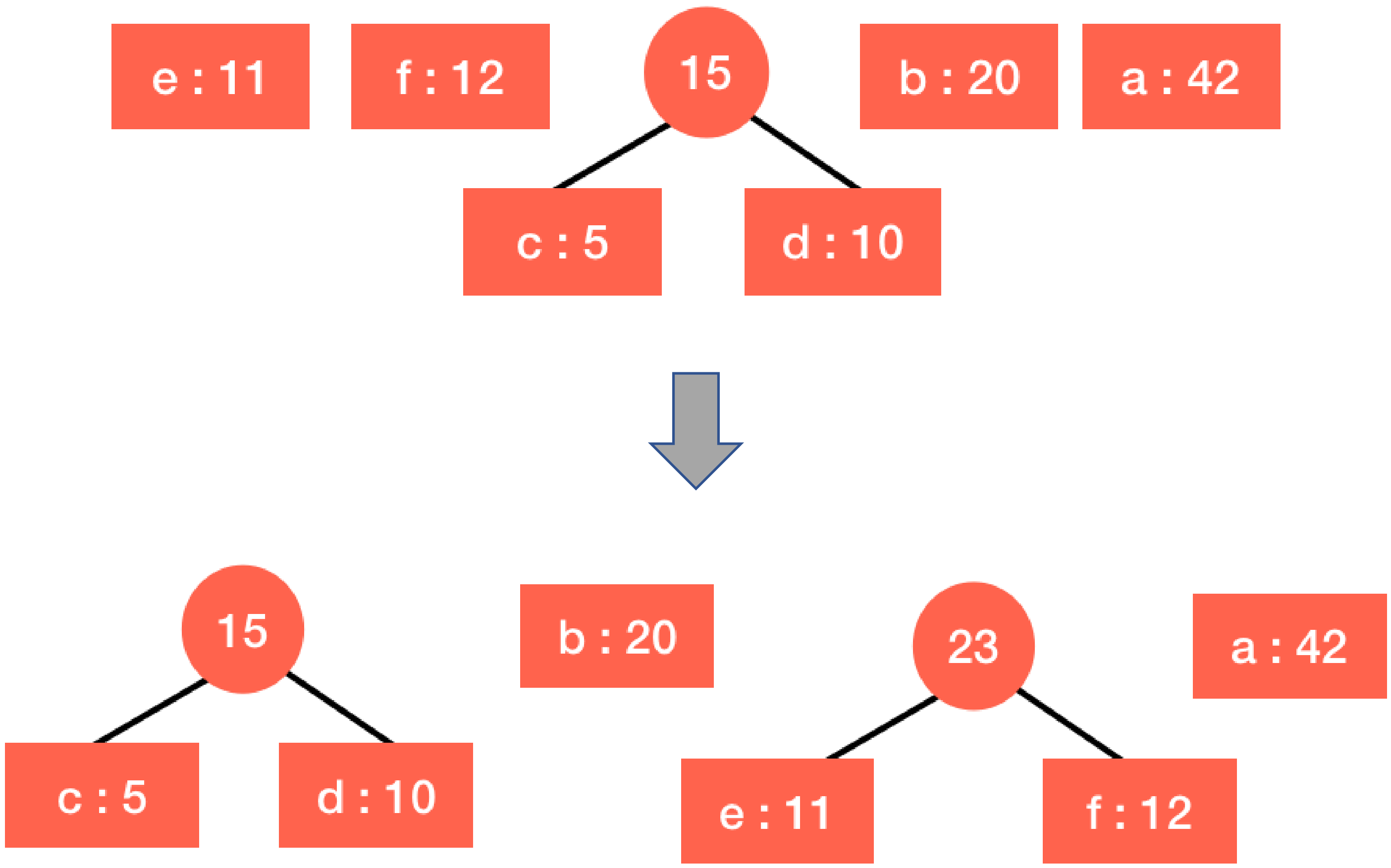
- Huffman Coding
  - Build a priority queue by adding (x, w_x) for each symbol x
  - While $|\text{queue}| \geq 2$
    - Take the two symbols with the lowest weight (x, w_x) and (y, w_y)
    - Merge them into one symbol with weight $w_x + w_y$

- Let's see this on the previous example

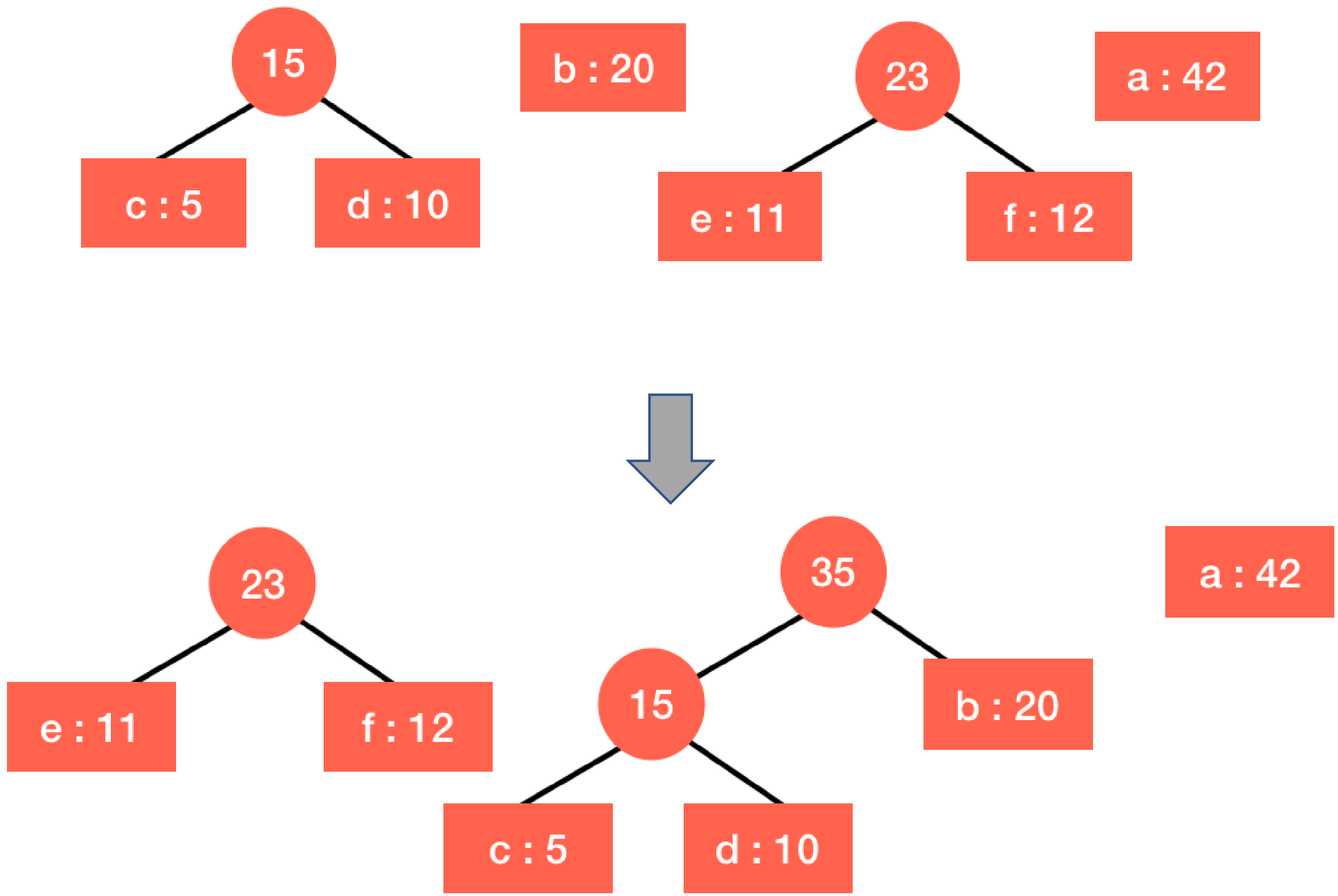
Lossless Compression



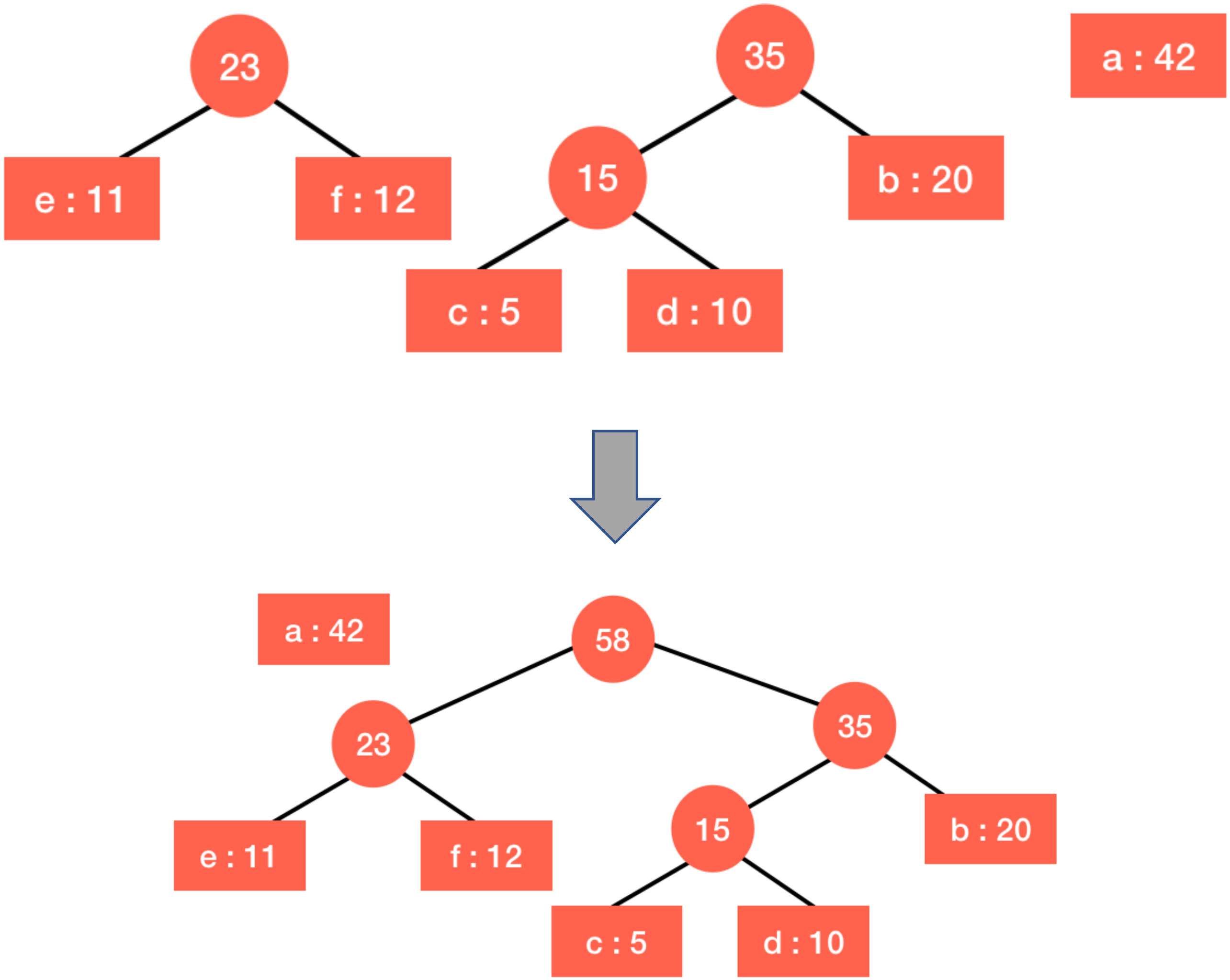
Lossless Compression



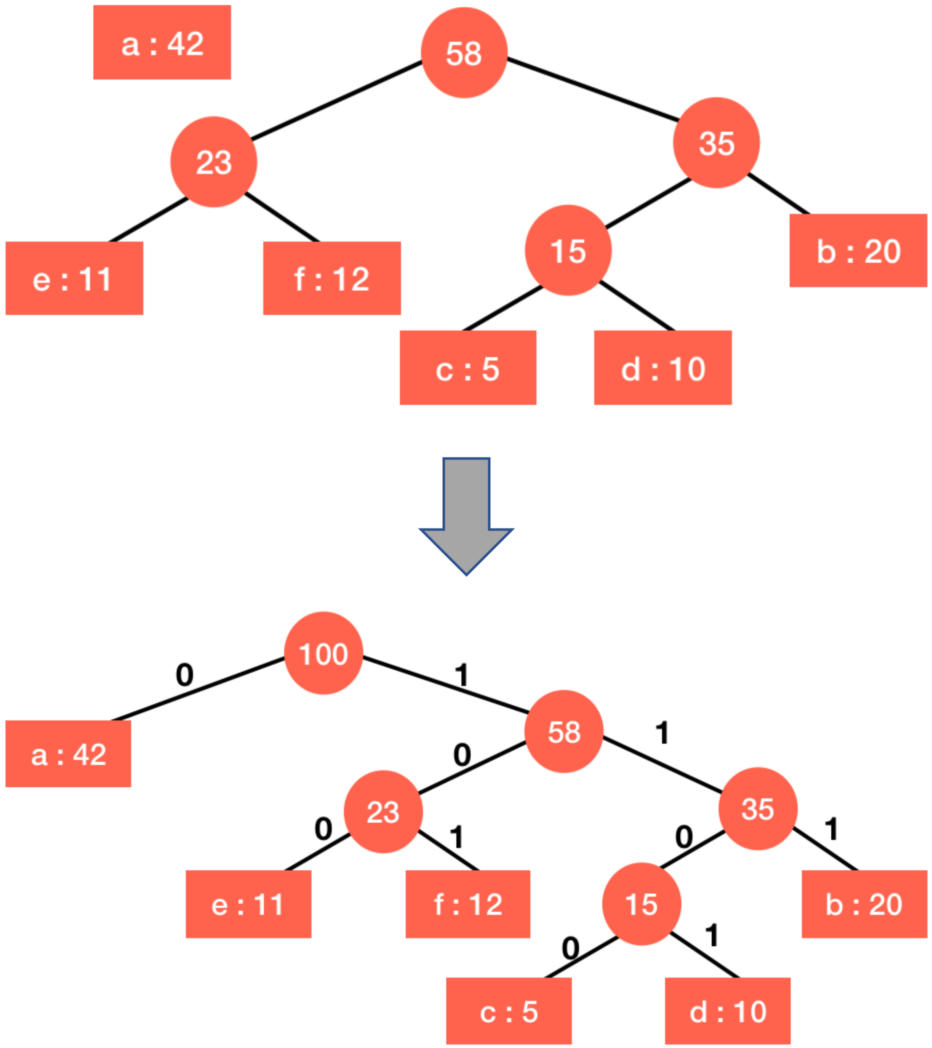
Lossless Compression



Lossless Compression



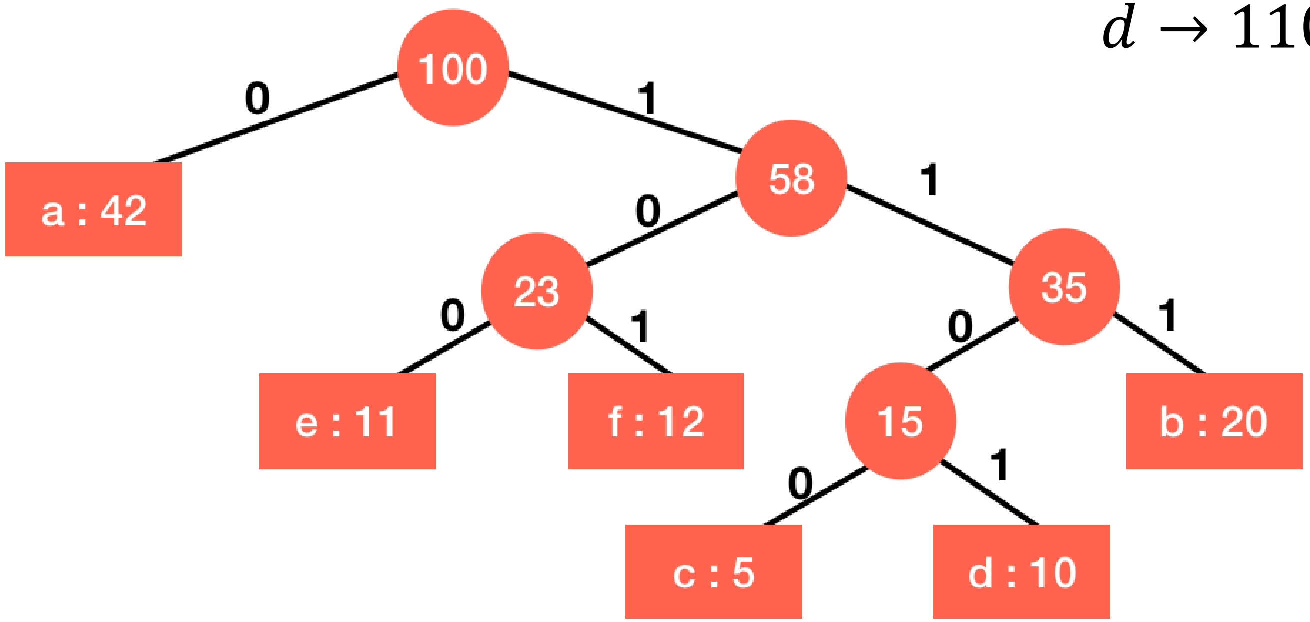
Lossless Compression



Lossless Compression

- Final Outcome

$a \to 0, e \to 100,$
$f \to 101, c \to 1100,$
$d \to 1101, b \to 111$



Lossless Compression

- Running time
  - $O(n \log n)$
  - Can be made $O(n)$ if the labels are given to you sorted by their frequencies

- Proof of optimality
  - Induction on the number of symbols n
  - Base case: For $n = 2$, there are only two possible encodings, both are optimal, assign 1 bit to each symbol
  - Hypothesis: Assume it returns an optimal encoding with $n - 1$ symbols

Lossless Compression

- Proof of optimality
  - Consider the case of n symbols
  - Lemma 1: If $w_x < w_y$, then $\ell_x \geq \ell_y$ in any optimal tree.
    - Proof sketch: Otherwise, swapping x and y would strictly reduce the overall length (exercise!).
  - Lemma 2: There is an optimal tree T in which the two least frequent symbols are siblings.
    - Proof sketch: First prove that they must have the same longest length assigned to them. Then, if they're not siblings, chop and rearrange the tree to make them siblings (exercise!).
  - Now, we can compare the tree H produced by Huffman vs such an optimal tree T

Lossless Compression

- Proof of optimality
  - Let x and y be the two least frequency symbols
  - In Huffman, we combine them in the first step into "xy"
  - Let H' and T' be trees obtained from H and T by treating xy as one symbol with frequency $w_x + w_y$
  - Use induction hypothesis: $Length(H') \leq Length(T')$
  - $Length(H) = Length(H') + \left(w_x + w_y\right) \cdot 1$
  - $Length(T) = Length(T') + \left(w_x + w_y\right) \cdot 1$
  - QED!

Other Greedy Algorithms

- If you aren’t familiar with the following algorithms, spend some time checking them out!
  - Dijkstra’s shortest path algorithm
  - Kruskal and Prim’s minimum spanning tree algorithms