Design and Analysis of Algorithms

Greedy Algorithms

- **Interval Scheduling**
- **Interval Partitioning**
- **Schedule for Minimizing Lateness**
- **Single Source Shortest Paths**
  - Dijkstra's Algorithm
- **Minimum Spanning Trees**
  - Kruskal's algorithm
  - Prim's algorithm
  - Boruvka's algorithm
- **Huffman Coding**

Zahed Rahmati
Dept of Math and CS
Tehran Polytechnic

Introduction

- **Optimization problem**: Given various constraints, minimizes cost, or maximizes profit.

We consider a *simple design technique* for optimization problems,

GREEDY ALGORITHMS:

- Intuitively, a greedy algorithm **builds up a solution** for some problem by **selecting the best alternative with each step**.
- It leads to very simple and efficient algorithms.
- They often provide fast heuristics.
- They are often used in finding good approximations.

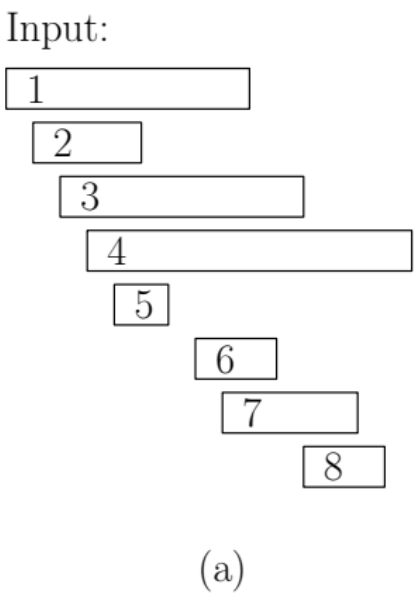
Interval Scheduling

- Among the most fundamental optimization problems.
- Given a set $R = \{1, 2, \dots, n\}$ of n activity requests, and some resource, where each activity must be started at a given start time s_i and ends at a given finish time f_i.
- We say that two activities i and j conflict if their start-finish intervals overlap, *i.e.*, $[s_i, f_i] \cap [s_j, f_j] \neq \emptyset$.

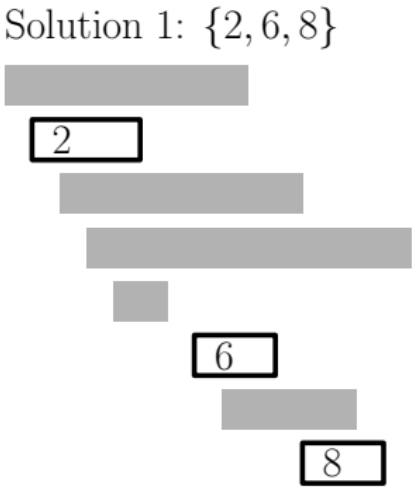
❖ **Interval scheduling problem:** Given a set R of n activities, **determine a subset of *R*** of **maximum cardinality** that are **mutually non-conflicting**.

An example

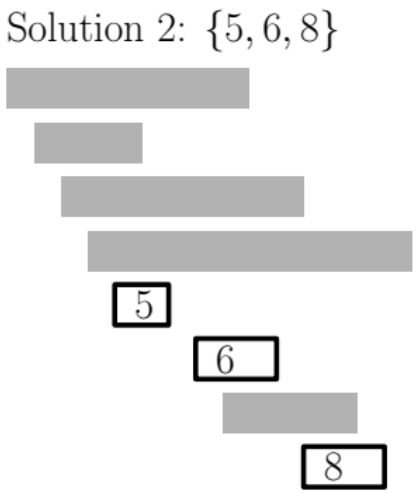
- An input (a) and two possible solutions (b and c):
- How do we schedule the *largest number of activities* on the resource?



(a)



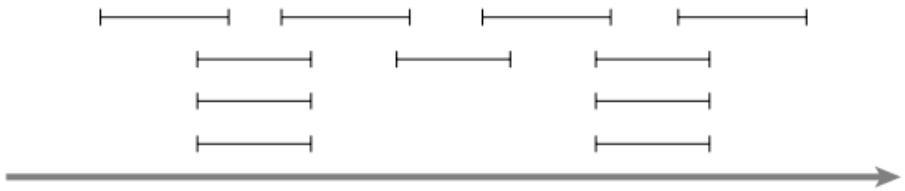
(b)



(c)

Interval scheduling solutions!

- **Earliest Activity First:**
  - Schedule the activity with the earliest start time that it does not overlap any of the previously scheduled activities.
  - **EX**: A single very long activity with an early start time.
- **Shortest Activity First:**
  - Repeatedly select the activity with the smallest duration $(f_i - s_i)$.
  - **EX**: Two long non-conflicting activities that a short activity overlaps both.
- **Lowest Conflict Activity First:**
  - Count for each activity the number of other activities it overlaps.
  - **EX**: ?



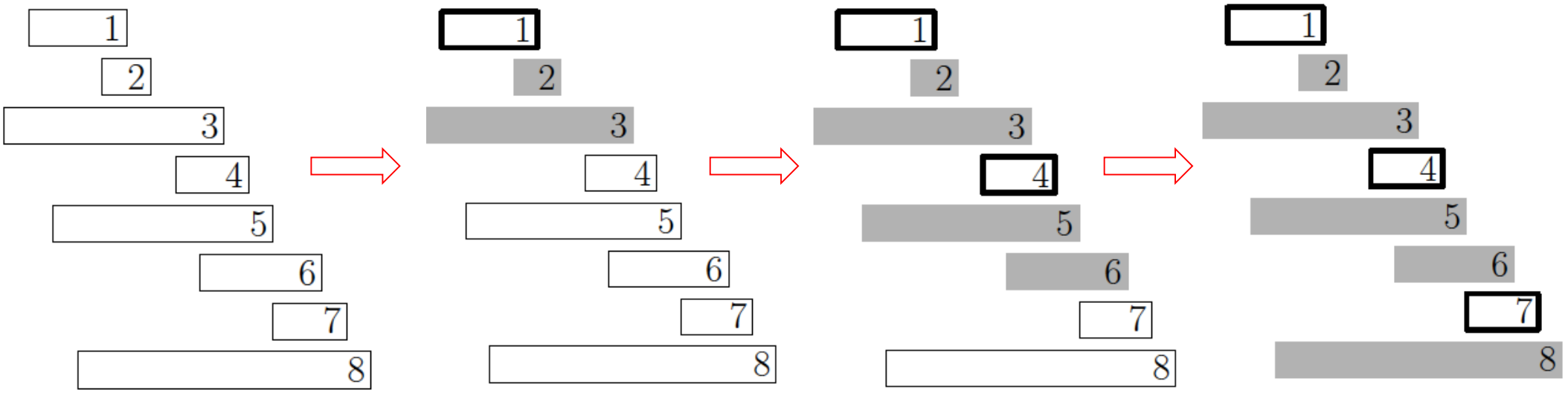
Interval scheduling: Algorithm

- We do not like activities that take a long time! Thus,
- **Earliest Finish First:**
  - Select the activity that **finishes first** and schedule it.

```
greedySchedule(R) {
    A = empty
    while (R is nonempty) {
        r = the request of R having the smallest finish time
        Append r to the end of A
        Delete from R all requests that overlap r
    }
    return A
}
```

  - Running time?
    - First sort the activities in increasing order of finishing time.
    - $O(n \log n)$

Interval scheduling: An example



Add 1 (skip 2,3)

Add 4 (skip 5,6)

Add 7 (skip 8)

Interval scheduling: Correctness

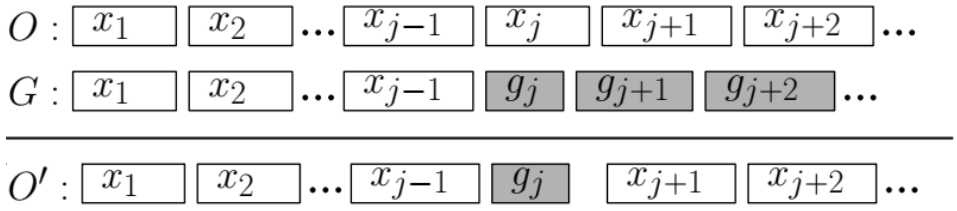
- Is this a **valid** schedule? Is this schedule **optimal**?
- Suppose that you have any non-greedy solution.
- **We prove**: **Any schedule that is not greedy** can **be made more greedy**, **without decreasing the number of activities**.

➢ Consider any optimal schedule O: $O = \langle x_1, \ldots, x_{j-1}, x_j, \ldots \rangle$

➢ Let G be the greedy schedule: $G = \langle x_1, \ldots, x_{j-1}, g_j, \ldots \rangle$

❑ **We will eventually convert O into G without decreasing the number of activities.**

- Sort the activities in increasing order of finish time.
  - If $O = G$, we are done! Otherwise, (consider first activity x_j, where $x_j \neq g_j$)
  - Construct **a new optimal schedule O'** that is more similar to G than O is.



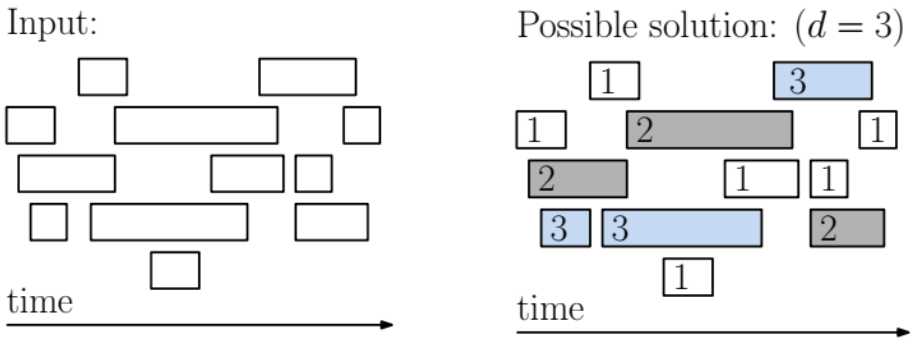
❑ **This new "greedier" schedule O′ is valid. Since it has the same number of activities as O, it is optimal.**

Interval Partitioning

- Let $R = \{x_1, x_2, \ldots, x_n\}$ be a set of n activity requests, with start and finish times s_i and f_i, respectively, of the ith request.
- Schedule **all the activities** using the **smallest number resources.**

❖ **Interval Partitioning Problem: Find the smallest number *d***, such that it is possible to partition R into ***d* disjoint** subsets $R_1, \ldots, R_d$, such that the events of R_j are non-conflicting.

- View this as a **coloring problem**!



Interval Partitioning: Algorithm

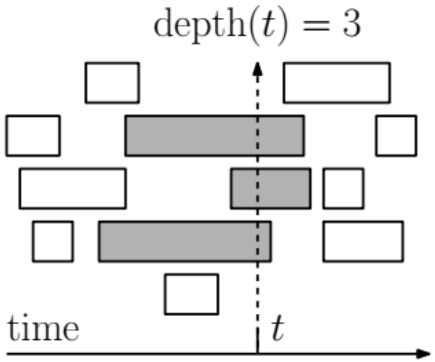
```
greedyPartition(R) {
    sort activities by increasing start times -- (x1, ..., xn)
    for i = 1 to n do {
        E = emptyset
        for j = 1 to i-1 do {
          if (xj conflicts with xi) add color[xj] to E
        }
        Let c be the smallest color not in E
        color[xi] = c
    }
    return color[1..n]
}
```

- Running time? Can be done faster?
- Is *sorting of the activities* essential to the algorithm's correctness? Why?

Interval Partitioning: Correctness

❑ **We observe that the algorithm never assigns the same color to two conflicting activities.**

- **depth(t)**: #activities whose interval contains t;
- **depth(R)**: max depth over all values of t.



❑ **Claim: Given R, #resources needed $\geq$ depth(R).**

- Activities that contribute to **depth(t)** conflict with one another, thus
- we need **at least this many resources** to schedule these activities. (**$d \geq$ depth(R)**)

❑ **Claim: Given R, #resources produced by the algorithm $\leq$ depth(R).**

We prove: #colors assigned to the activities at time *t* is at most depth(*t*):

- Consider an arbitrary start time s_i, during the execution of the algorithm.
- Let $t = s_i - \varepsilon$. Let h denote the depth at time t.
- By our hypothesis, just prior to time s_i #colors being used $\leq$ current depth ($= h$).
- When time s_i is considered, the depth increases by 1.
- Since $\leq h$ colors are in use prior to time s_i, then **there exists an unused color among the first $h+1$ colors**.
- **Thus the total number of colors used at time s_i is $h + 1$, which is not greater than the total depth.**

Scheduling to Minimize Lateness

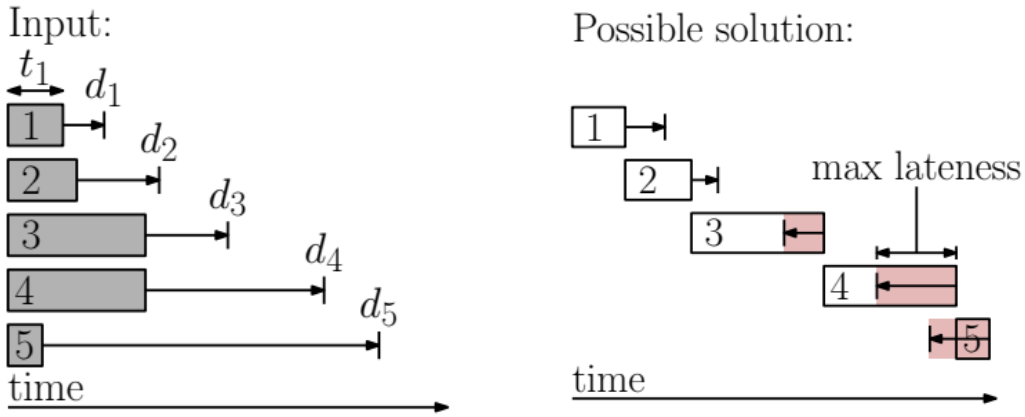
- Let $T = \{x_1, x_2, \ldots, x_n\}$ be a set of n tasks, where each task $x_i = (t_i, d_i)$ has a deadline d_i and the time t_i needed to perform the task.
- Suppose task x_i starts at time s_i. It finishes at time $f_i = s_i + t_i$.

- We say that task x_i is ***late*** if $f_i > d_i$, and
- its ***lateness*** is $\boldsymbol{l_i = max(0, f_i - d_i)}$

Maximum Lateness: $\max_i l_i$

❖ **Our goal is to compute a schedule that minimizes the Maximum Lateness.**

Minimizing lateness: Example & Solutions



Greedy algorithms?

- **Smallest duration first**:
  - Ex. A very short job with a deadline way in the future!

- **Smallest slack-time ($d_i - t_i$) first**:
  - Ex. $(t_1, d_1) = (1, 2)$ and $(t_2, d_2) = (10, 10)$!

Minimizing lateness: Algorithm

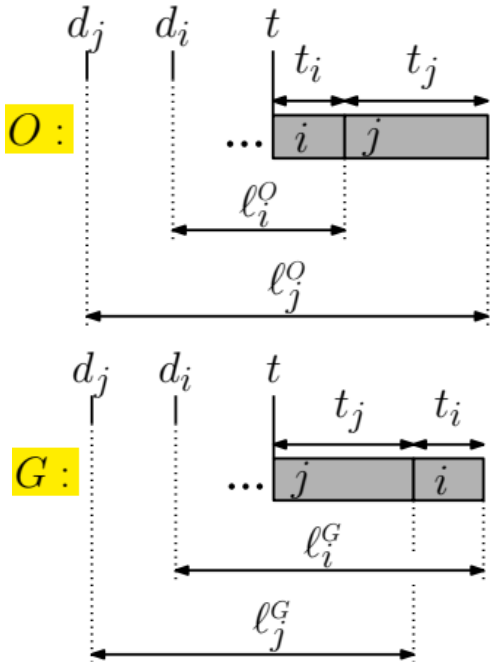
- **Shortest deadline first:**

```
greedySchedule(T=(t,d) {
    sort tasks by increasing deadline (d[1] <= ... <= d[n])
    f = 0
    for (i = 1 to n) do {
        assign task i to start at s[i] = f
        f = f[i] = s[i] + t[i]
        lateness[i] = max(0, f[i] - d[i])
    }
    return arrays s and lateness
}
```

- Running time: $O(n \log n)$.
- This algorithm has no idle time: by moving tasks up to fill in any idle times, we can only reduce lateness. Thus:
  - **There is an optimal schedule with no idle time.**

Minimizing lateness: Correctness

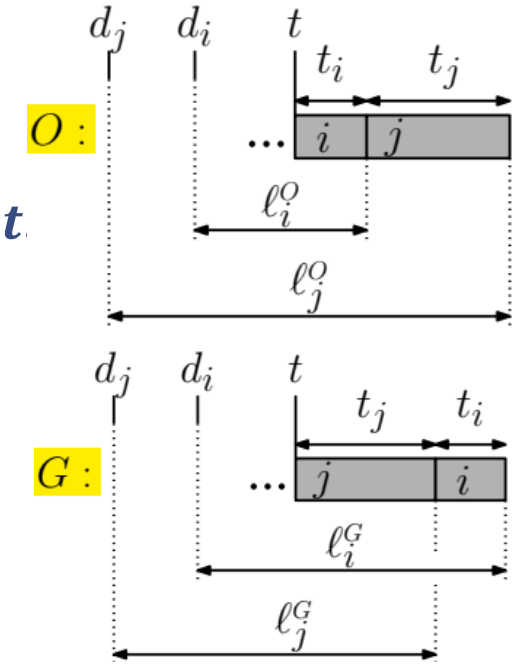
- **Morph any optimal schedule to look like our greedy schedule:**
  - Let G be the schedule produced by the algorithm.
  - Let O be any optimal schedule.



- **If** $G = O$, then greedy is optimal!
- **Otherwise**, $(G \neq O)$, O must contain at least one inversion:
  - **The schedules O and G are identical up to these two tasks.**
  - **Let x_i and x_j be the first two consecutive tasks such that $d_j < d_i$.**
  - Since $d_j < d_i$, task x_j is scheduled before x_i **in schedule G**.
  - Task x_i is scheduled before x_j **in schedule O**.

Minimizing lateness: Correctness

Swapping x_i and x_j in O does not increase lateness.



- Since there is no slack time, and both schedules are identical up to these two tasks, the **two tasks starts at the same time t**.

$$\ell_i^O = \max(0, t + t_i - d_i) \qquad \ell_i^G = \max(0, t + (t_i + t_j) - d_i)$$
$$\ell_j^O = \max(0, t + (t_i + t_j) - d_j) \qquad \ell_j^G = \max(0, t + t_j - d_j)$$

- Let $L^O = \max(\ell_i^O, \ell_j^O)$ and $L^G = \max(\ell_i^G, \ell_j^G)$

$$\ell_j^O = t + (t_i + t_j) - d_j > t + t_i - d_i = \ell_i^O$$
$$\ell_i^G = t + (t_i + t_j) - d_i < t + (t_i + t_j) - d_j = \ell_j^O = L^O$$
$$\ell_j^G = t + t_j - d_j \leq t + (t_i + t_j) - d_j = \ell_j^O = L^O$$

$$L^G = \max(\ell_i^G, \ell_j^G) \leq L^O$$

❑ **The greedy scheduling algorithm minimizes maximum lateness.**

Shortest Paths

- Finding shortest paths from a single source vertex to all other vertices:
  - Breadth-first search is an $O(V + E)$ algorithm.
  - Assuming that the graph has no edge weights.
- The length of a path is the sum of weights along the edges of the path.
- The distance between u and v, $\delta(u, v)$:
  - minimum length of any path between u and v.
- Shortest path problems:
  - Single source shortest-path problem: Given a source s, compute $\delta(s, u)$, for all u.
  - Single sink shortest-path problem: Given a sink t, compute $\delta(u, t)$, for all u.
  - All-pairs shortest path problem: Compute $\delta(u, v)$, for all u and v.

Single Source Shortest Paths

- Are negative-valued edge weights allowed?
  - Do not usually arise in transportation networks, but
  - can arise in financial transaction networks, where a transaction (edge) may result in either a lost or a profit.
- Shortest path problem is ***well defined, even if having negative edge*** weights, **assuming there are no negative cycles**.
- Dijkstra's algorithm (Dutch computer scientist, 1959):
  - A simple greedy algorithm!
  - Among the most famous algorithms in Computer Science.

Dijkstra's algorithm: Relaxation

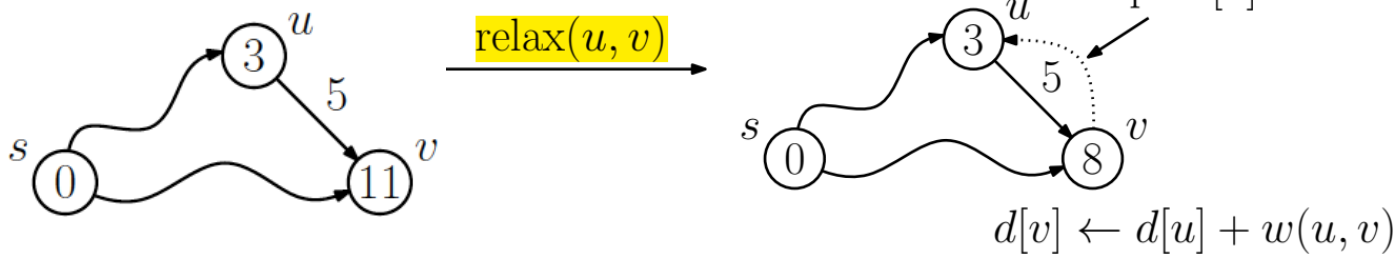
- Dijkstra's algorithm maintains $d[v]$, **length of the shortest path from s to v** that the algorithm currently knows of.

➢ At the beginning, there is no paths, so $d[v] = \infty$, and $d[s] = 0$.

➢ Next as the algorithm sees more vertices, it updates $d[v]$ for each vertex.

Relaxation on (u, v): The process by which an estimate $d[v]$ is updated.

```
relax(u, v) {
    if (d[u] + w(u, v) < d[v]) {
        d[v] = d[u] + w(u, v)
        pred[v] = u
    }
}
```



Dijkstra's algorithm: Key properties

- **If perform relax (u, v) repeatedly over all edges, $d[v]$ values will eventually converge to the final true distance from s.**

- **Best possible** would be to **relax each edge exactly once**.
  - Assuming the edge weights are nonnegative, Dijkstra's algorithm achieves this.

- Dijkstra's algorithm maintains a subset $S \subseteq V$ of vertices, for which we know their true distances, that is $d[v] = \delta(s, v)$, $v \in S$.

- Dijkstra recognized this: The **best way to perform relaxations** is **by increasing order of distance** from the source s
  - Take the vertex of $V \backslash S$ for which $d[u]$ is minimum.
  - Store the vertices of $V \backslash S$ in a **priority queue**, where the key of each u is $d[u]$.
    - **Build**: Create a priority queue from a list of n elements, in $O(n)$ time.
    - **Extract min**: Remove/return the element with the smallest key value, in $O(\log n)$ time.
    - **Decrease key**: Given an element, decrease its key value, in $O(\log n)$ time.

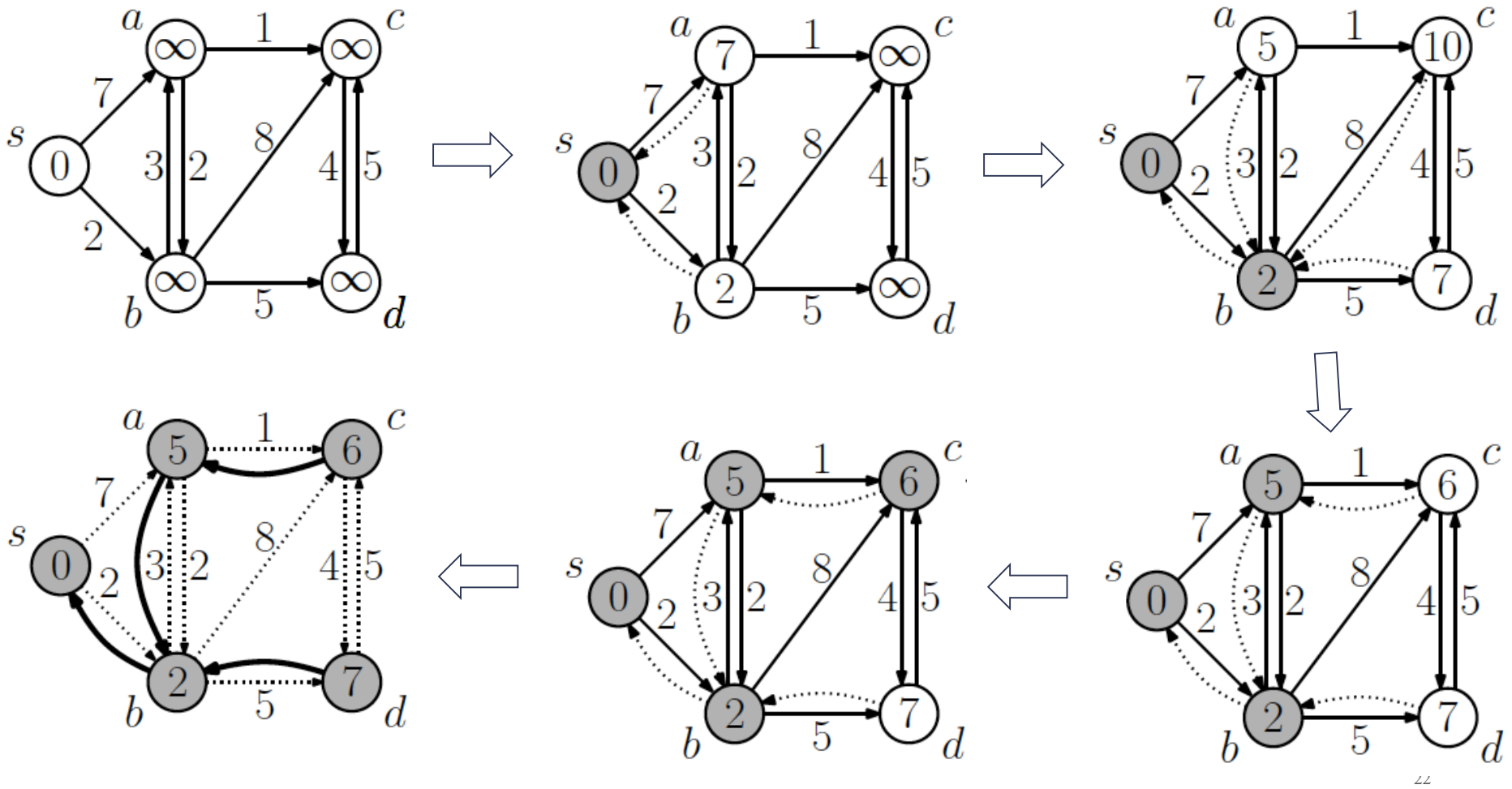
Dijkstra's algorithm

```
dijkstra(G,w,s) {
    for each (u in V) {
        d[u] = +infinity
        mark[u] = undiscovered
        pred[u] = null
    }
    d[s] = 0
    Q = a priority queue of all vertices u sorted by d[u]
    while (Q is nonEmpty) {
        u = extract vertex with minimum d[u] from Q
        for each (v in Adj[u]) {
            if (d[u] + w(u,v) < d[v]) { // relax(u,v)
                d[v] = d[u] + w(u,v)
                decrease v's key in Q to d[v]
                pred[v] = u
            }
        }
        mark[u] = finished
    }
    [The pred pointers define an ``inverted'' shortest path tree]
}
```

- **Running time:**

$$
\begin{aligned}
T(n,m) &= \sum_{u \in V} (\log n + \deg(u) \cdot \log n) \\
&= \sum_{u \in V} (1 + \deg(u)) \log n \\
&= \log n \sum_{u \in V} (1 + \deg(u)) \\
&= (\log n)(n + 2m) \\
&= \Theta((n+m) \log n)
\end{aligned}
$$

Dijkstra's algorithm: Example



Dijkstra's algorithm: Correctness

Claim: When a vertex u is added to S, $d[u] = \delta(s,u)$.

Proof (by contradiction):

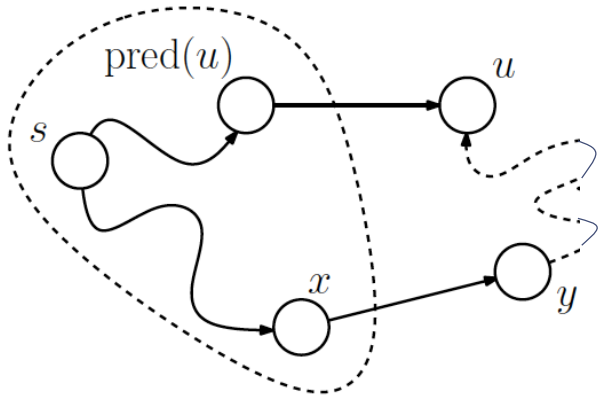
- Suppose the algorithm ***first*** attempts to add u to S ***for which*** $\boldsymbol{d[u] \neq \delta(s,u)}$. Thus we have $d[u] > \delta(s,u)$.
- Just prior to insertion of u, consider the true shortest path from s to u:
  - Let (x,y) be the first edge taken by the shortest path, where $x \in S$ and $y \in V \backslash S$.

$$d[y] = d[x] + w(x,y) = \delta(s,x) + w(x,y) = \delta(s,y)$$

$$\delta(s,u) < d[u] \leq d[y] = \delta(s,y) \leq \delta(s,u)$$

Since u (not y) was chosen next for processing in the alg.

Since y appears before u along the shortest path



Single-source shortest path prob.: Variants

- **Vertex weights**: There is a cost associated with each vertex. The overall cost is the sum of vertex and/or edge weights on the path.
- **Single-Sink Shortest Path**: Find the shortest path from each vertex to a sink vertex t.
- **Multi-Source/Multi-Sink**: You are given a collection of source vertices $\{s_1, s_2, \ldots, s_k\}$. For each vertex $v_i \in V$ find the shortest path from its nearest source s_j. (Analogous for multi-sink.)
- **Multiplicative Cost**: Define the cost of a path to be the product of the edge weights (rather than the sum.) If all the edge weights are at least 1, find the single-source shortest path.

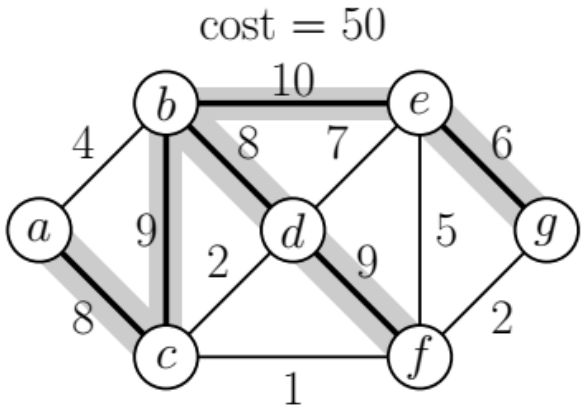
➢ **How can you modify Dijkstra's algorithm or underlying graph to solve these problems?**

Minimum Spanning Trees

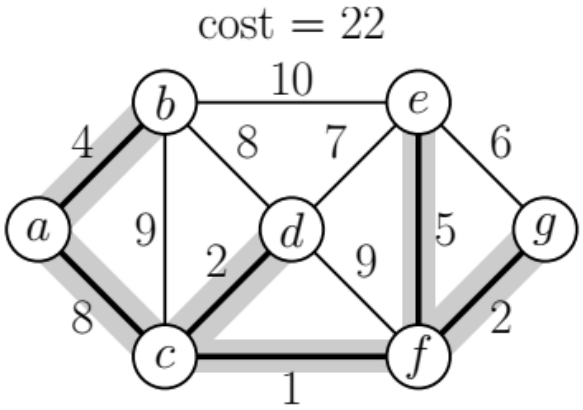
- **Motivation**: In communications networks and circuit design.
  - **Goal**: **Connecting together a set of nodes by a network of minimal total length**, where
  - the ***length*** is the sum of the lengths of connecting wires.
  - We assume the given network $G(V, E)$ is **connected** and **undirected**.
- **Spanning tree (ST):**
  - An **acyclic** subset of edges $T \subseteq E$ that connects all the vertices together.
  - Assume each edge (u, v) of G has a numeric weight/cost, $\boldsymbol{w(u, v)}$.
  - **Cost (weight) of a spanning tree *T***: $\boldsymbol{w(T)} = \sum_{(u,v)\in T} w(u, v)$
- **Minimum spanning tree (MST):**
  - A spanning tree of minimum weight $\boldsymbol{w(T)}$.
  - MST may not be unique, but if all edge weights are distinct, then MST will be distinct.

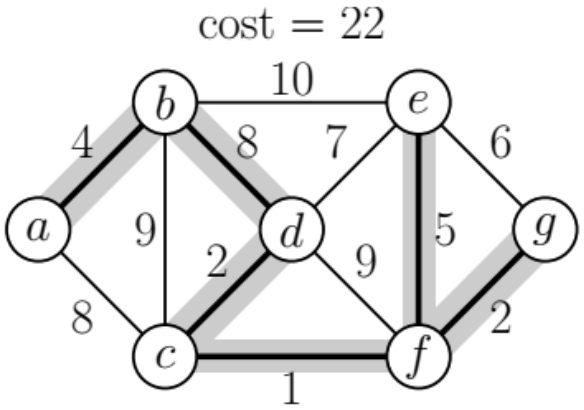
ST and MST: Example

- An spanning tree:
  - In fact, it is a maximum weight spanning tree.



- Two minimum spanning trees:





MST: Greedy algorithms

- We present three greedy algorithms:
  - Kruskal's algorithm, Prim's algorithm, and Boruvka's algorithm.
- A greedy algorithm **builds a solution by repeated selecting the cheapest** (or generally **locally optimal choice**) **among all options at each stage.**
- ***Once*** greedy algorithms ***make a choice***, they ***never unmake this choice***.
- **Basic facts about trees:**
  - A tree with n vertices has exactly $n - 1$ edges.
  - There exists a unique path between any two vertices of a tree.
  - Adding any edge to a tree creates a unique cycle, and breaking any edge on this cycle restores a tree.

Generic approach

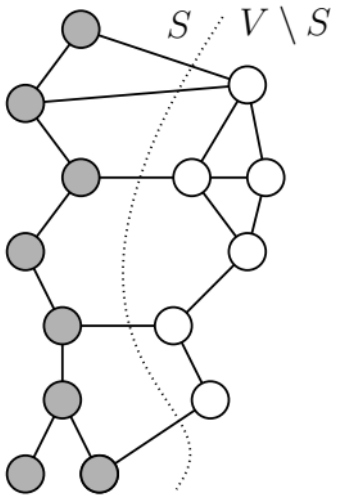
- Let G = (V, E) be the input graph.
- **Intuition**: We maintain a subset of edges A (which will initially be empty), and will add edges one at a time, until A is a spanning tree.
- **Terminology**:
  - The **subset $A \subseteq E$** is **viable** if **A is a subset of edges in some MST**.
  - The **edge $(u, v) \in E \backslash A$** is **safe** if **$A \cup \{(u, v)\}$ is viable**.
    - E\A means the edges of E that are not in A.

- A **generic MST algorithm** repeatedly does the following:
  - It **adds any safe edge** to the current spanning tree.

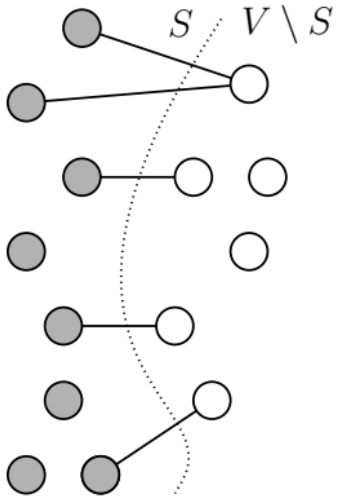
MST: Definitions

- A **cut $(S, V \setminus S)$** is a partition of the vertices into two disjoint subsets.
  - **Let $S \subset V$.**
  - **Let $A \subset E$.**
- An edge (u, v) **crosses** the cut if $u \in S$ and $v \notin S$.
- An edge is a **light edge** crossing a cut, if among all edges crossing the cut, it has the min weight.
- We say **a cut respects A** if no edge in A crosses the cut.

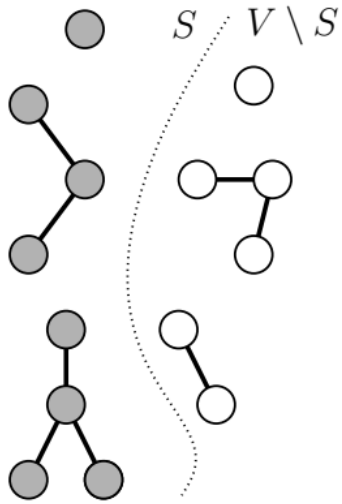
A cut $(S, V \setminus S)$



Edges crossing the cut



A subset respecting the cut



MST Lemma

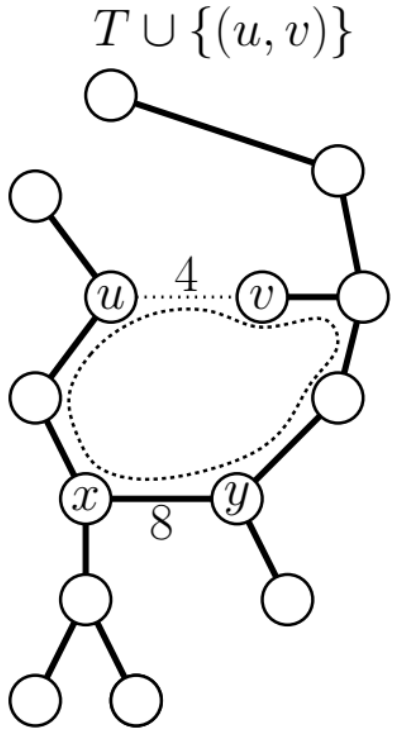
Let $G = (V, E)$ be a connected, undirected graph with weights on the edges. Let A be a viable subset of E, let $(S, V \setminus S)$ be any cut that respects A, and let (u, v) be a light edge crossing this cut. Then the edge (u, v) is safe for A.

Proof, by contradiction : (Assume that all the edge weights are distinct)

- Suppose that no MST, like T, contains the edge (u, v).
- Adding the edge (u, v) to T creates a cycle.
  - At least one other edge (x, y) of T crosses the cut.
- By removing (x, y), we construct a spanning tree T':

$$w(T') = w(T) - w(x, y) + w(u, v)$$

- Since (u, v) is lightest edge crossing the cut, $w(x, y) > w(u, v)$,
- Thus $w(T') < w(T)$, which contradicts the assumption that T was an MST.



Kruskal's algorithm: Key points

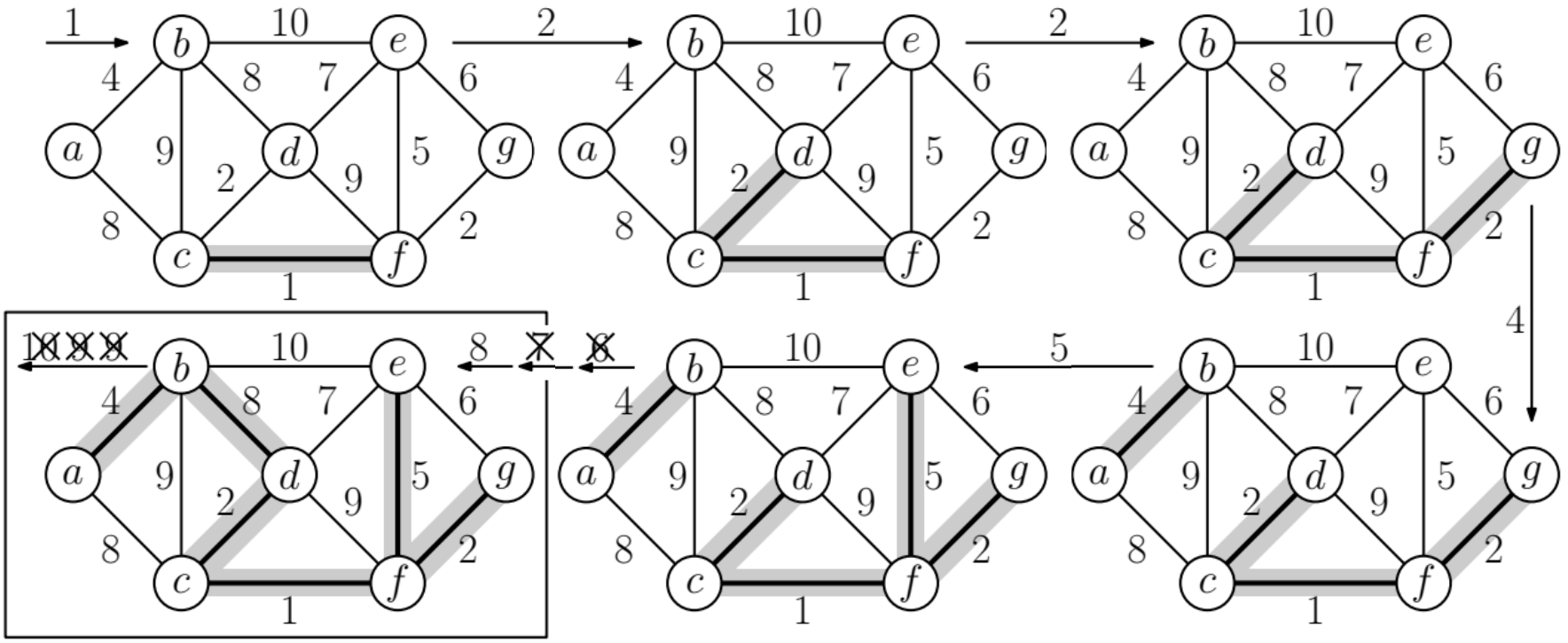
- It adds edges to A in increasing order of weight.
- Consider the edge (u, v) that the algorithm seeks to add next:
  - Suppose that this edge does not induce a cycle.
  - By the MST Lemma, $(\boldsymbol{u, v})$ **is safe**.
- **How to detect whether adding (u, v) will create a cycle in A?**
  - We use **disjoint set union-find data structure**:
    - **Create**$(\boldsymbol{u})$: Create a set containing a single item v.
    - **Find**$(\boldsymbol{u})$: Find the set that contains a given item u.
    - **Union**$(\boldsymbol{u, v})$: Merge the set containing u and the set containing v into a common set.
  - The union-find data structure can perform any sequence of up to n union and find operations in total time $O(n \cdot \alpha(n))$, where $\alpha(n)$ is the inverse Ackerman's function. Here, $\alpha(n) = O(\log n)$.

Kruskal's algorithm

```
KruskalMST(G=(V,E), w) {
    A = {}
    Place each vertex u in a set by itself
    Sort E in increasing order by weight w
    for each ((u, v) in this order) {
        if (find(u) != find(v)) {
            add (u, v) to A
            union(u, v)
        }
    }
    return A
}
```

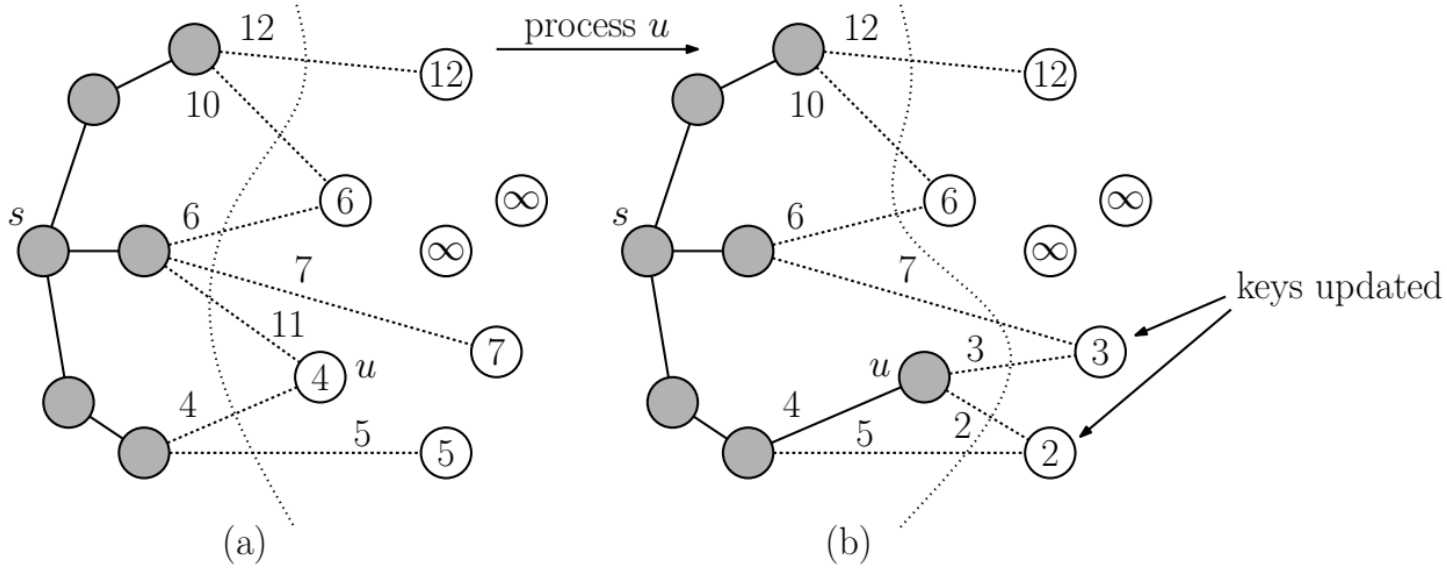
- **Running time** (as the graph is connected, $m \geq n - 1$)?
  - Sort the edges; $\Theta(m \log m) = \Theta(m \log n)$ time.
  - For-loop iterates m times, each involves $O(1)$ accesses to the Union-Find data structure, so total of $\Theta(m \log n)$.

Kruskal's algorithm: Example



Prim's algorithm: Key points

- Prim's algorithm looks like Dijkstra's algorithm.
- Its running time is essentially the same as Kruskal's algorithm.
- It builds the tree up by **adding leaves one at a time** to the current tree.



Prim's algorithm: Finding a safe edge

- Consider the set of vertices S currently part of the tree, and its complement $(V \backslash S)$.
- From the MST Lemma, it is safe to add the light edge.
- How to determine the light edge quickly?
- We use a **priority queue data structure:**
  - **Insert**$(\boldsymbol{u}, \boldsymbol{k})$: Insert u with the key value k in Q.
  - **Extract-min()**: Extract the item with the minimum key value in Q.
  - **Decrease-key**$(\boldsymbol{u}, \boldsymbol{k'})$: Decrease the value of u's key value to k'.
- All the above operations can be performed in time $O(\log n)$.

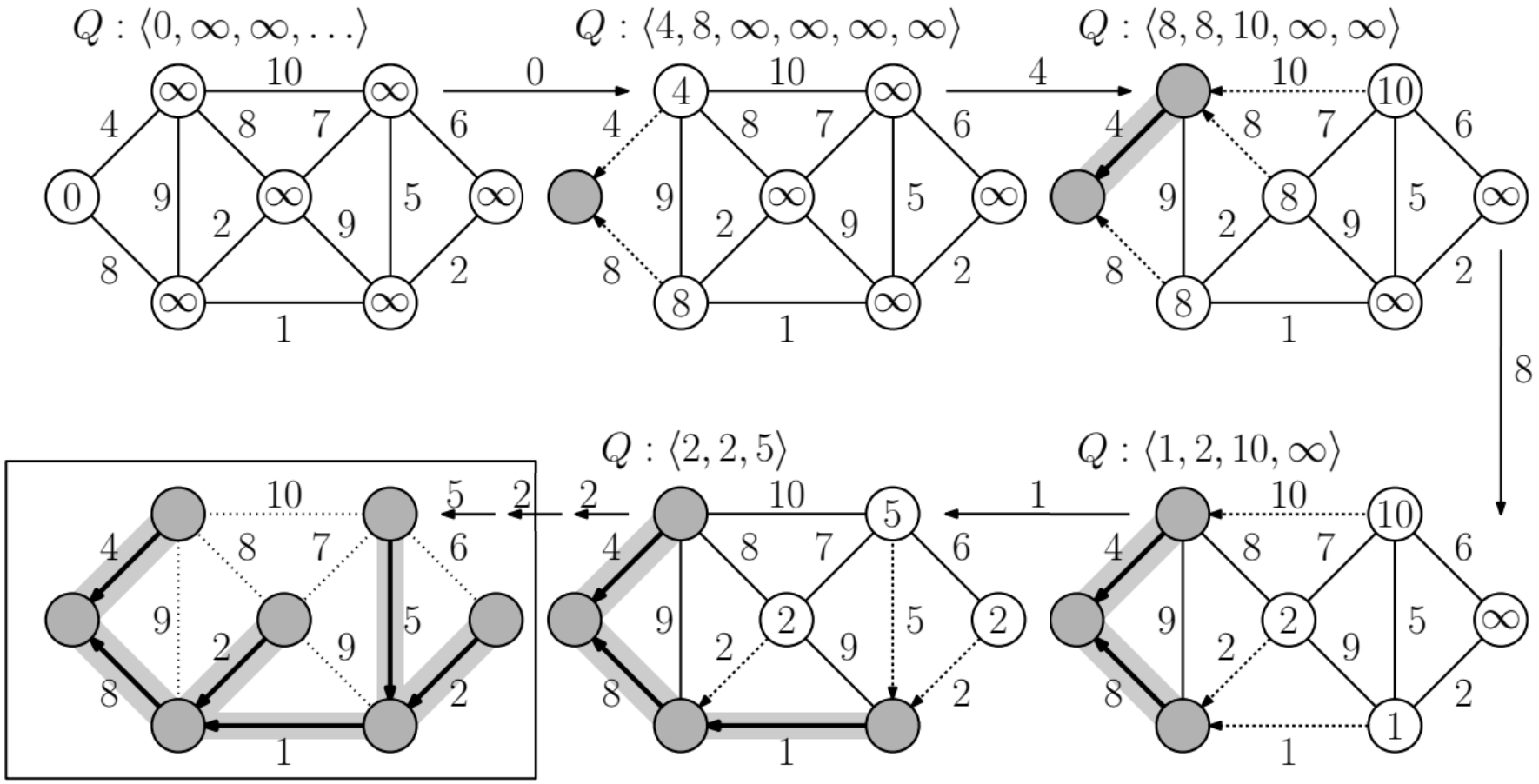
Prim's algorithm

```
PrimMST(G=(V,E), w, s) {
    for each (u in V) {
        key[u] = +infinity
        color[u] = undiscovered
    }
    key[s] = 0
    pred[s] = null
    add all vertices to priority queue Q
    while (Q is nonEmpty) {
        u = extract-min from Q
        for each (v in Adj[u]) {
            if ((color[v] == undiscovered) && (w(u,v) < key[v])) {
                key[v] = w(u,v)
                decrease key value of v to key[v]
                pred[v] = u
            }
        }
        color[u] = finished
    }
    [The pred pointers define the MST as an inverted tree rooted at s]
}
```

Running time?

- It takes $O(\log n)$ to extract min from Q.
- For each incident edge, we spend $O(\log n)$ time for the neighboring vertex.
- Thus the time is $O(\log n + \text{degree}(u) \cdot \log n)$.
- Thus total time is $\Theta((n+m)\log n)$

Prim's algorithm: Example



Boruvka's algorithm: Key points

- Oldest algorithm, 1926 by the Czech mathematician Otakar Boruvka.
- It is the easiest to implement on a parallel computer.

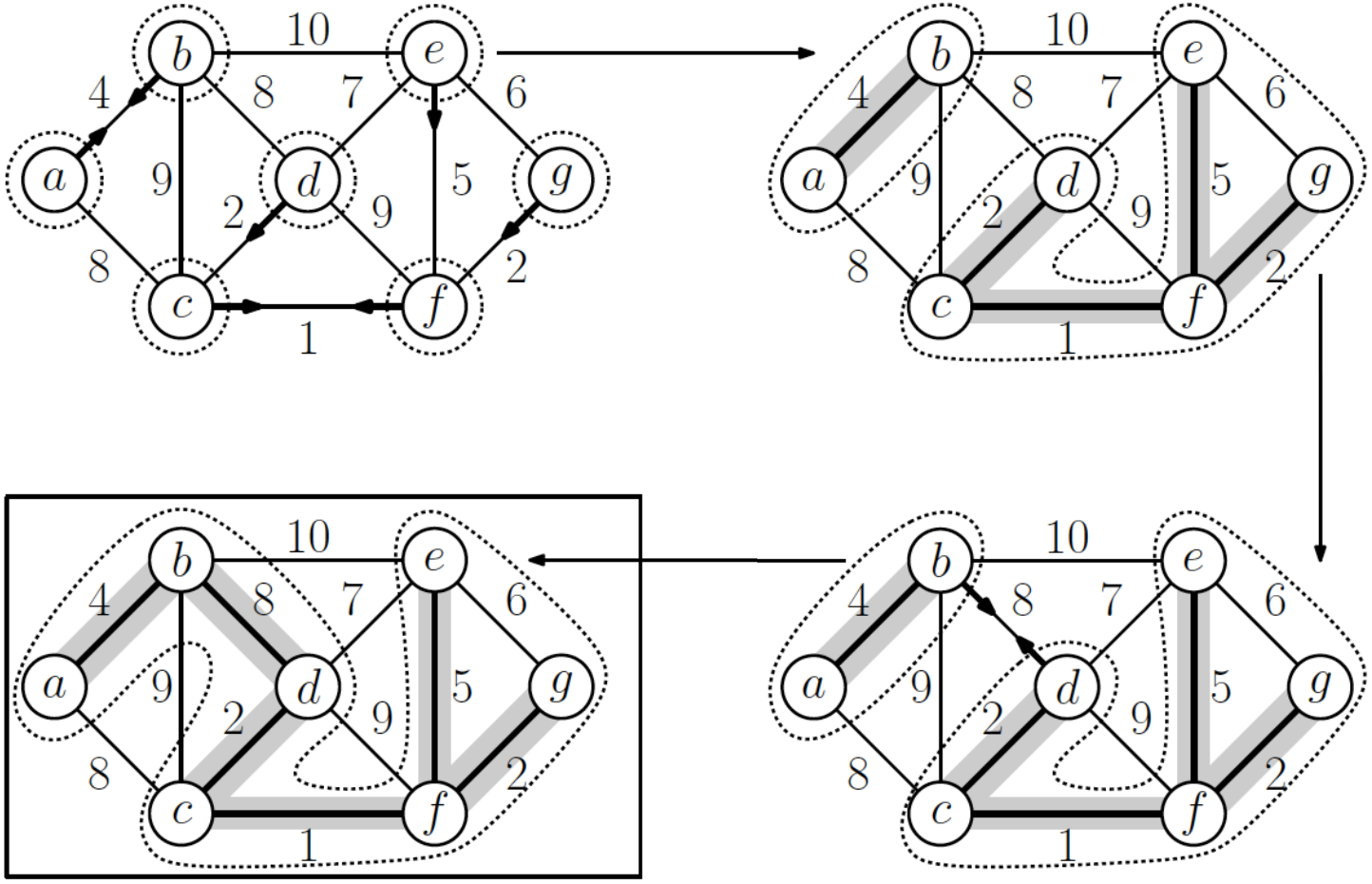
- Boruvka's algorithm adds a whole set of edges all at once to the MST.
  - In a single step, many components can be merged together into a single component.
  - Assume that edge weights are distinct.

Boruvka's algorithm

```
BoruvkaMST(G=(V,E), w) {
    initialize each vertex to be its own component
    A = {}
    while (there are two or more components) {
        for (each component C) {
            find the lightest edge (u,v) with u in C and v not in C
            add {u,v} to A (unless it is already there)
        }
        apply DFS to graph (V, A), to compute the new components
    }
    return A
```

- **Running time:**
  - By DFS, each iteration can be performed in $O(n+m)$ time.
  - How many iterations are required in general?
    - After each step, we have at most half of components. Thus it is $O(\log n)$.

Boruvka's algorithm: Example



Huffman Coding: Introduction

- A method of **encoding data** efficiently.
- Standard codes, like *ASCII* or the *Unicode,* are represented by a ***fixed-length codeword*** of bits (8 or 16 bits per character).

Character	a	b	c	d
Fixed-Length Codeword	00	01	10	11

- A string "abacdaacac" would be encoded by **20 characters**:

a	b	a	c	d	a	a	c	a	c
00	01	00	10	11	00	00	10	00	10

➢ **What if you knew the probabilities of characters in advance?**

Variable-length code

- Scan the file and determine the exact frequencies of all the characters.

Character	a	b	c	d
Probability	0.60	0.05	0.30	0.05
Variable-Length Codeword	0	110	10	111

- **Frequently occurring characters are encoded using fewer bits.**
- The string "abacdaacac" could be encoded by **17 characters**:

a	b	a	c	d	a	a	c	a	c
0	110	0	10	111	0	0	10	0	10

- Therefore, for an input string of length n,
  - The **2-bit fixed-length code** uses $2n$ **bits**.
  - The #bits uses in the **variable-length code** is:

25% savings!

$$n(0.60 \cdot 1 + 0.05 \cdot 3 + 0.30 \cdot 2 + 0.05 \cdot 3) = n(0.60 + 0.15 + 0.60 + 0.15) = 1.5n$$

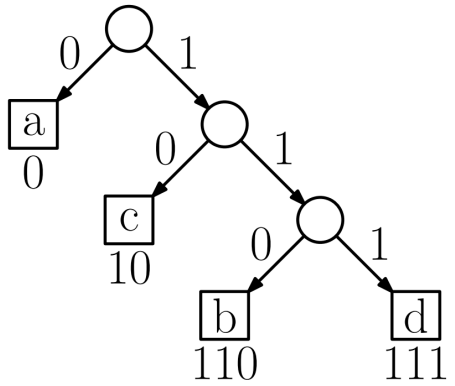
Prefix Codes

- **Once a string is encoded, can we decode the string?**
  - e.g. $a \rightarrow 001$ and $b \rightarrow 00101$, how do we decide if we see 00101 ...?
  - **The code should have the property that it can be uniquely decoded.**
- **If no codeword is a prefix of any other, as soon as we see a codeword as a prefix, we may decode without fear *of it matching some longer codeword.***

❑ **Prefix code: Map codewords to characters so that no codeword is a prefix of another.**

- A **tree-representation** of a prefix code:

Character	a	b	c	d
Codeword	0	110	10	111



- **Expected number of bits** for encoding a text of length n:

$$B(T) = n \sum_{x \in C} p(x) d_T(x)$$

❑ **We want to compute a prefix code T that minimizes the expected length *B(T)*.**

Huffman's algorithm: Key points

- Huffman coding was used for many years for file compression.
  - **Pack** was a Unix shell compression program based on Huffman coding.
  - Later **better compression methods** discovered:
    - **gzip** is based on a more sophisticated method called the Lempel-Ziv coding.
    - **bzip2** is based on combining the Burrows-Wheeler transformation with run-length encoding, and Huffman coding.
- We are **given** the **occurrence probabilities** for the characters.
- IDEA: Build the tree up from the leaf:
  - We merge x and y as the left and right children of a root node z.
  - z will have a probability equal to the sum of x and y's probabilities.

Huffman's algorithm

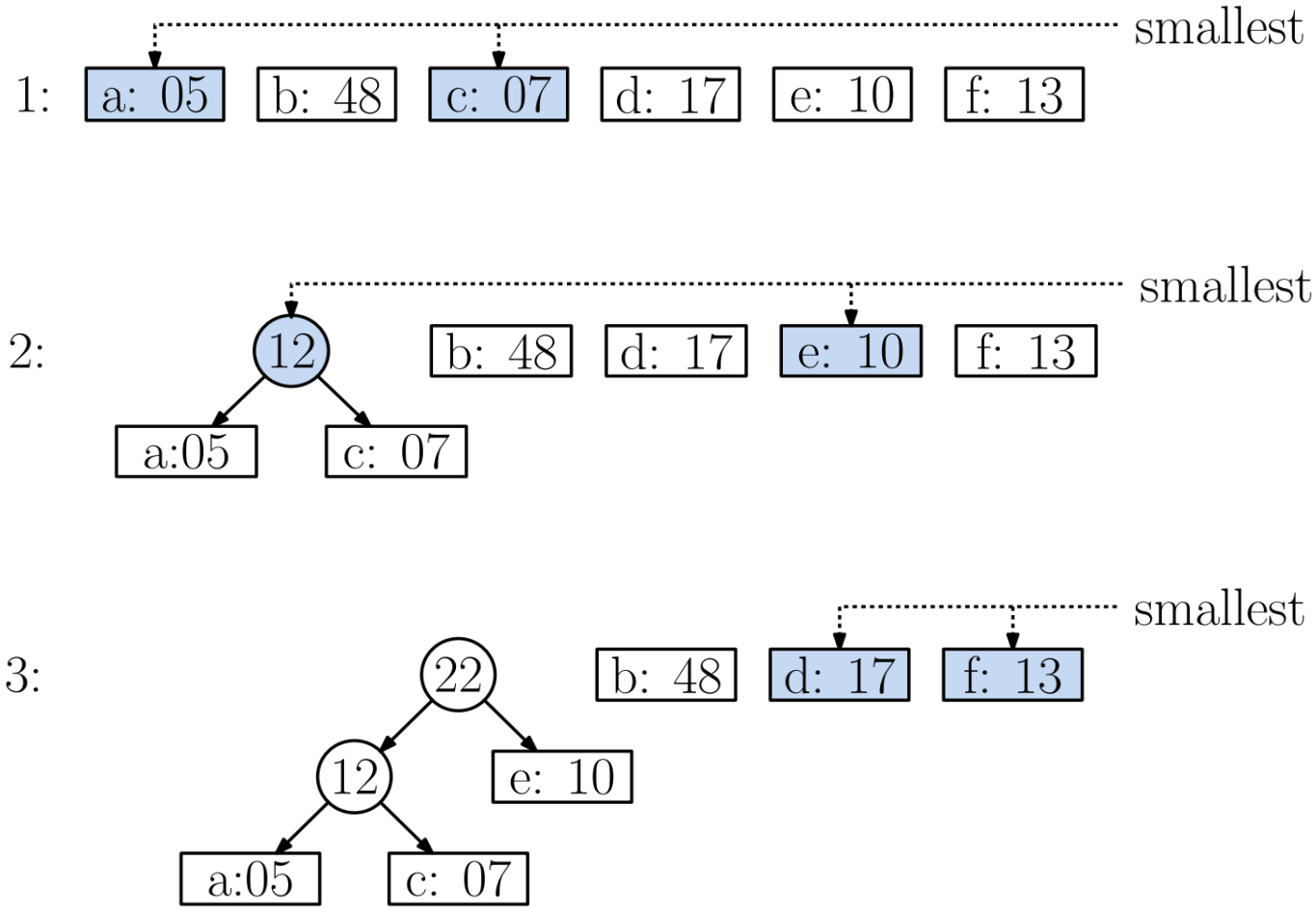
```
huffman(char C[], float prob[]) {
    for each (x in C) {
        add x to Q sorted by prob[x]
    }
    n = size of C
    for (i = 1 to n-1) {
        z = new internal tree node
        left[z]  = x = extract-min from Q
        right[z] = y = extract-min from Q
        prob[z]  = prob[x] + prob[y]
        insert z into Q
    }
    return the last element left in Q as the root
}
```

Running time:

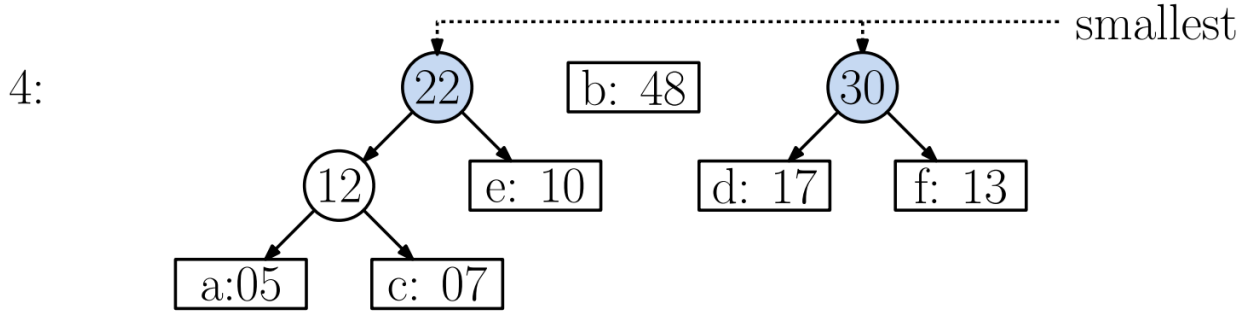
- Build a priority queue Q in $O(n)$ time.
- Extract the element with the smallest key in $O(\log n)$ time.
- Insert a new element in $O(\log n)$ time.

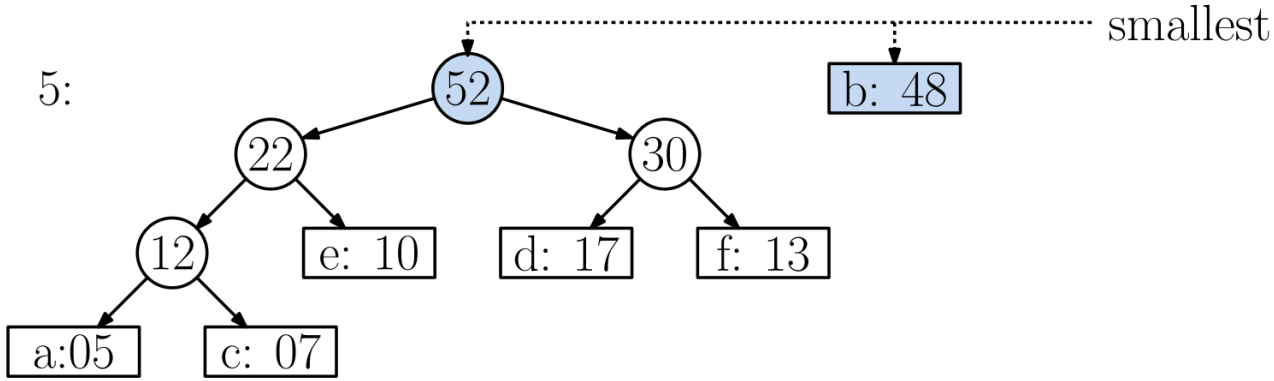
- Total Time: $O(n \log n)$

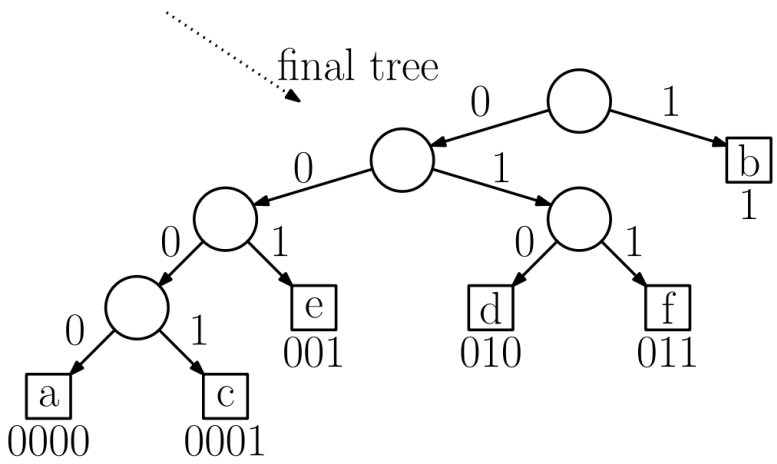
Huffman's algorithm: Example



Huffman's algorithm: Example







Huffman's algorithm: Correctness

- The cost of any encoding tree T is $B(T) = n \sum_{x \in C} p(x) d_T(x)$

➢ Convert an arbitrary solution to the greedy solution by swapping elements.

- Observe that the **Huffman tree is a *full binary tree***.

1. In any optimal code tree, the **two characters with the lowest probabilities** will be **siblings at the maximum depth** in the tree.

2. Merge these two characters into a single super character. Therefore, we will now have one less character in our alphabet. This will allow us to apply **induction** to the remaining $\boldsymbol{n-1}$ **characters.**

❑ **Claim:** Consider the two characters, $\boldsymbol{x}$ **and** $\boldsymbol{y}$ **with the smallest probabilities.** Then there is **an optimal code tree** in which these **two characters** are siblings **at the maximum depth in the tree.**

Huffman's algorithm: Correctness

Proof of Claim:

- **Let T be any optimal prefix tree**, and let b and c be two siblings of maximum depth.
- Without loss of generality, assume that $p(b) \leq p(c)$ and $p(x) \leq p(y)$.

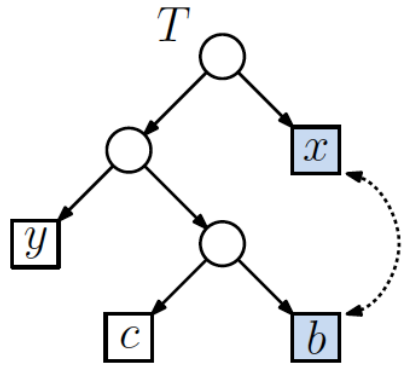
1. Since x and y have the two smallest probabilities, then $\boldsymbol{p(x) \leq p(b)}$ **and** $\boldsymbol{p(y) \leq p(c)}$.
2. As b and c are at the deepest level, we know $\boldsymbol{d_T(b) \geq d_T(x)}$ **and** $\boldsymbol{d_T(c) \geq d_T(y)}$.

➢ From 1-2, we have $p(b) - p(x) \geq 0$ and $d_T(b) - d_T\ (x) \geq 0$; their product is nonnegative.

❑ Switch the positions of x and b in the tree, resulting in a new tree T′.

- How the cost changes as we go from T to T′:

$$B(T') = B(T) - p(x)d_T(x) + p(x)d_T(b) - p(b)d_T(b) + p(b)d_T(x)$$
$$= B(T) + p(x)\big(d_T(b) - d_T(x)\big) - p(b)\big(d_T(b) - d_T(x)\big)$$
$$= B(T) - \big(p(b) - p(x)\big)\big(d_T(b) - d_T(x)\big) \leq B(T)$$



➢ ***Thus the cost does not increase. We can switch y with c to obtain a new tree T''.***

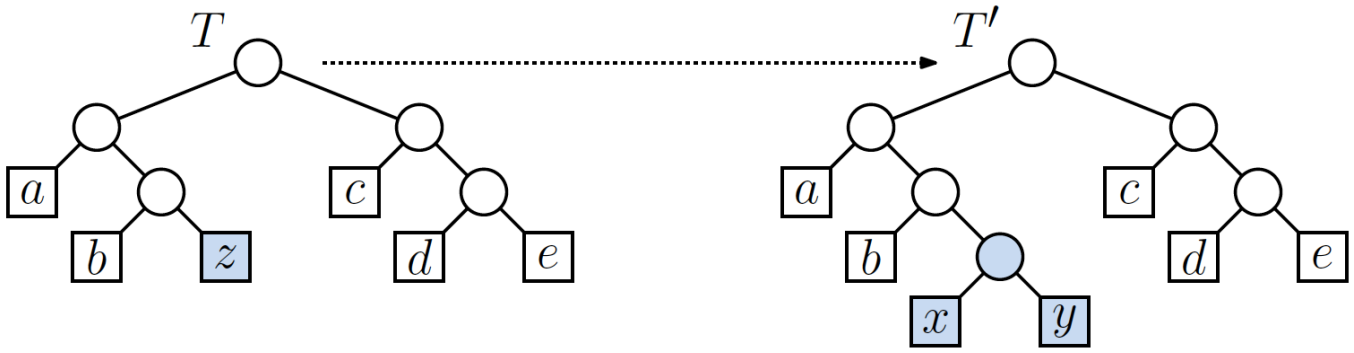
Huffman's algorithm: Correctness

Lemma: Huffman's algorithm produces an optimal prefix code tree.

Proof (by induction on n, the number of characters):

- For the basis case, n = 1, the tree consists of a single leaf node, which is obviously optimal.
- Assume that when given n-1 characters, the algorithm produces an optimal tree. We want to show this is true when the alphabet has exactly n characters.
- Let x and y be the two characters with the lowest probabilities.
- Remove x and y from the alphabet, and replace them with z whose probability is $p(z) = p(x) + p(y)$.
- Consider any prefix code tree T made with this new set of $n - 1$ characters.
  - We know that z will appear as a leaf node somewhere in this tree.
- Convert T into a prefix code tree T' for the original set of characters
  - by replacing the leaf node for z with an internal node,
  - whose left child (of z) is x and whose right child (of z) is y.

Huffman's algorithm: Correctness



Let T′ denote the resulting tree. The new cost is (x & y have now been added at depth $d(z)+1$):

$$\begin{aligned} B(T') &= B(T) - p(z)d(z) + p(x)(d(z)+1) + p(y)(d(z)+1) \\ &= B(T) - (p(x)+p(y))d(z) + (p(x)+p(y))(d(z)+1) \\ &= B(T) + (p(x)+p(y))(d(z)+1-d(z)) \\ &= B(T) + (p(x)+p(y)) \end{aligned}$$

- **In order to minimize $B(T')$, we should build T to minimize $B(T)$;** T involves n-1 characters and **Huffman's algorithm can produce such optimal tree.**

➢ **Thus the final tree is optimal.**