國立臺灣大學
National Taiwan University

CHAPTER 4
GREEDY ALGORITHMS

Iris Hui-Ru Jiang
Fall 2017

Department of Electrical Engineering
National Taiwan University

Outline

- Content:
  - Interval scheduling: The greedy algorithm stays ahead
  - Scheduling to minimize lateness: An exchange argument
  - Shortest paths
  - The minimum spanning tree problem
  - Implementing Kruskal's algorithm: Union-find
- Reading:
  - Chapter 4

Greedy Algorithms

- An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion.

- It's easy to invent greedy algorithms for almost any problem.
  - Intuitive and fast
  - Usually not optimal

- It's challenging to prove greedy algorithms succeed in solving a nontrivial problem optimally.
  1. The greedy algorithm stays ahead.
  2. An exchange argument.

Interval Scheduling

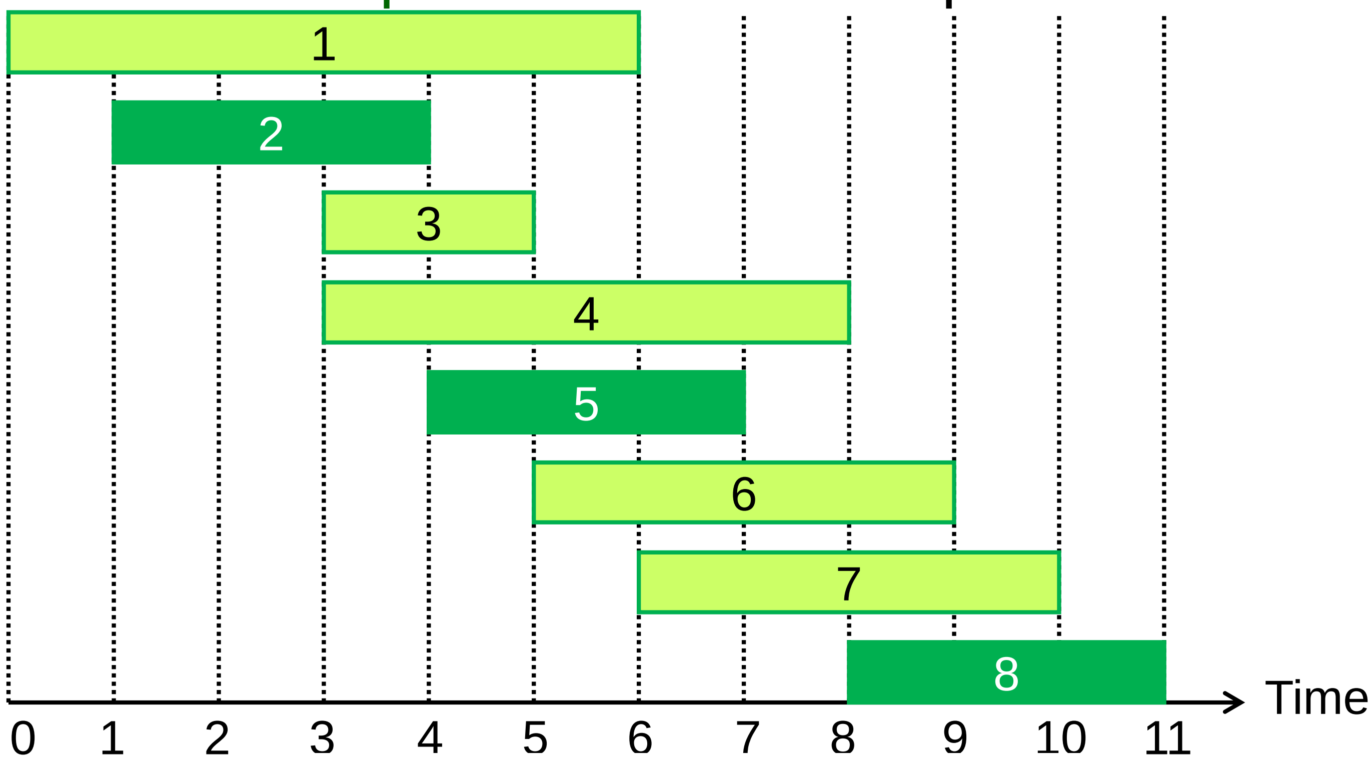
The greedy algorithm stays ahead

The Interval Scheduling Problem

- Given: Set of requests {1, 2, …, *n*}, *i*th request corresponds to an interval with start time *s*(*i*) and finish time *f*(*i*)
  - interval *i*: [*s*(*i*), *f*(*i*))
- Goal: Find a compatible subset of requests of maximum size

requests don't overlap

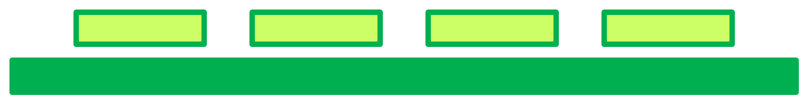
optimal



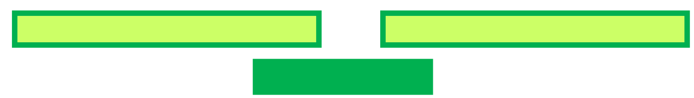
Maximum compatible subset {2, 5, 8}

Greedy Rule

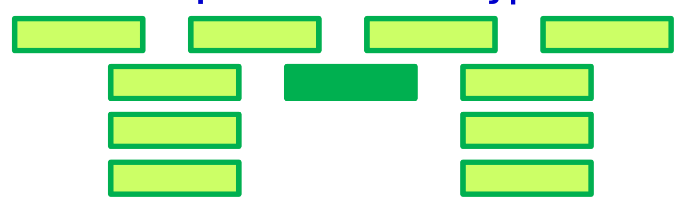
- Repeat
  - Use a simple rule to select a first request i_1
  - Once i_1 is selected, reject all requests incompatible with i_1.
- Until run out of requests
- Q: How to decide a greedy rule for a good algorithm?
- A:
  1. Earliest start time: min $s(i)$



  2. Shortest interval: min $\{f(i) - s(i)\}$



  3. Fewest conflicts: $\min_{i=1..n} |\{j: j \text{ is not compatible with } i\}|$



  4. Earliest finish time: min $f(i)$

The Greedy Algorithm

- The 4th greedy rule leads to the optimal solution.
  - We first accept the request that finish first
  - Natural idea: Free resource ASAP
- The greedy algorithm:

$\emptyset$: empty set = {}

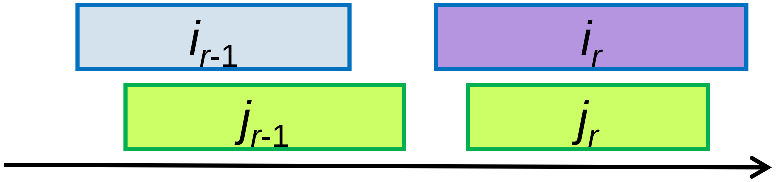
```
Interval-Scheduling(R)
// R: undetermined requests; A: accepted requests
1.  A = ∅;
2.  while (R is not empty) do
3.      choose a request i∈R with minimum f(i) // greedy rule
4.      A = A + {i}
5.      R = R – {i} – X, where X = {j: j∈R and j is not compatible with i}
6.  return A
```

  - Q: Feasible?
  - A: Yes! Line 5.
    - A is a compatible set of requests.
  - Q: Optimal? Efficient?

The Greedy Algorithm Stays Ahead

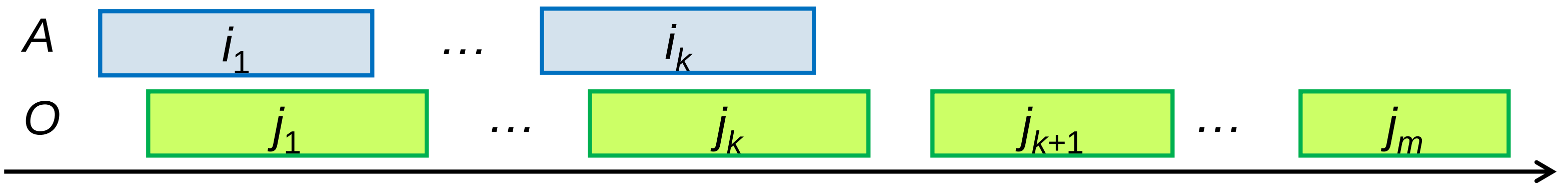
- Q: How to prove optimality?
  - Let O be an optimal solution. Prove $A = O$? or Prove $|A| = |O|$?
- The greedy algorithm stays ahead.
  - We will compare the partial solutions of A and O, and show that the greedy algorithm is doing better in a step-by-step fashion.
- Let A be the output of the greedy algorithm, $A = \{i_1, \ldots, i_k\}$, in the order they were added. Let O be the optimal solution, $O = \{j_1, \ldots, j_m\}$ in the ascending order of start (finish) times. For all indices $r \leq k$, we have $f(i_r) \leq f(j_r)$.
- Pf: Proof by induction!
  - Basis step: true for $r = 1$, $f(i_1) \leq f(j_1)$.
  - Inductive step: hypothesis: true for r-1.
    - $f(i_{r-1}) \leq f(j_{r-1})$
    - O is compatible, $f(j_{r-1}) \leq s(j_r)$
    - Hence, $f(i_{r-1}) \leq s(j_r)$; $j_r \in R$ after i_{r-1} is selected in line 5.
    - According to line 3, $f(i_r) \leq f(j_r)$.

Q: i_r finishes later?



The Greedy Algorithm Is Optimal

- The greedy algorithm returns an optimal set *A*.
- Pf: Proof by contradiction.
  - If *A* is not optimal, then an optimal set *O* must have more requests, i.e., $|O| = m > k = |A|$.
  - Since $f(i_k) \le f(j_k)$ and $m>k$, there is a request j_{k+1} in *O*.
  - $f(j_k) \le s(j_{k+1})$; $f(i_k) \le s(j_{k+1})$.
  - Hence, j_{k+1} is compatible with i_k. *R* should contain j_{k+1}.
  - However, the greedy algorithm stops with request i_k, and it is only supposed to stop when *R* is empty. $\rightarrow\leftarrow$

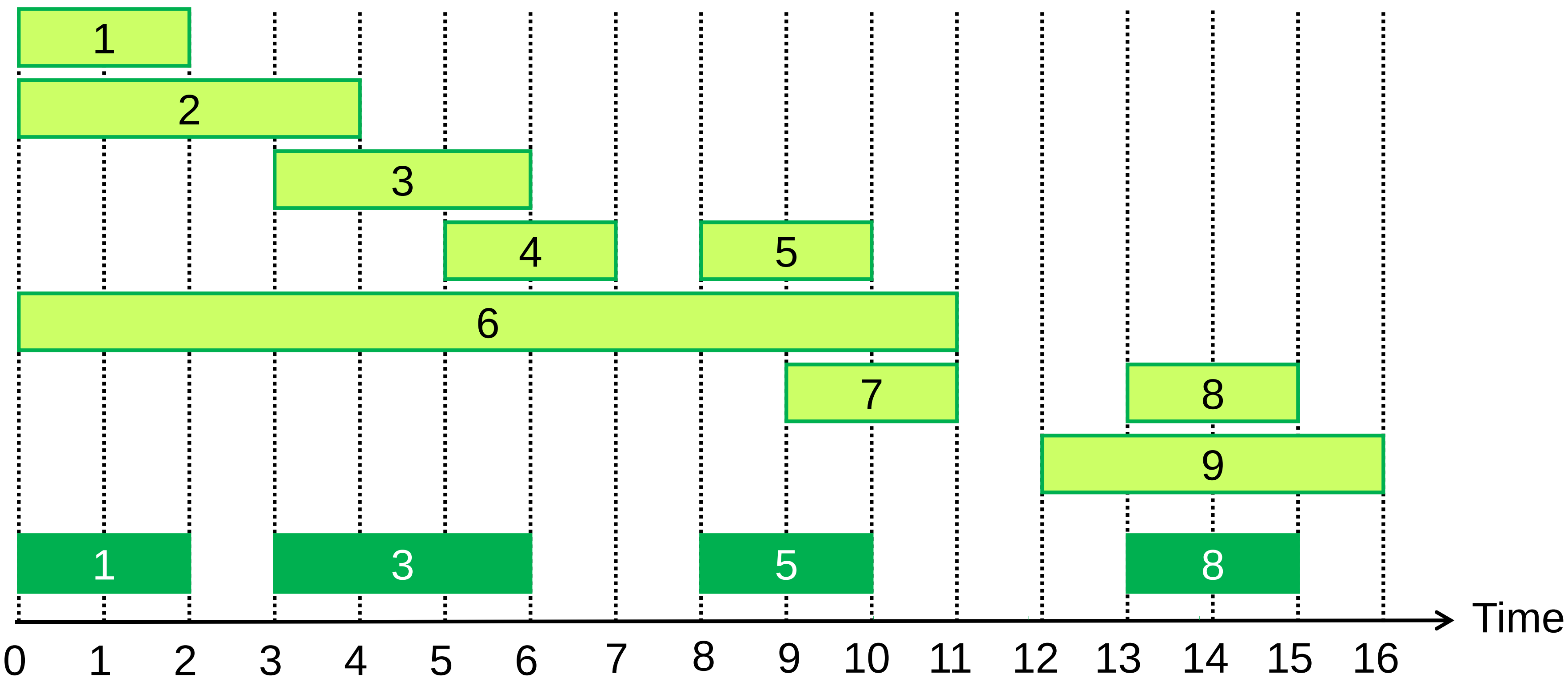


Implementation: The Greedy Algorithm

```
Interval-Scheduling(R)
// R: undetermined requests; A: accepted requests
1.  A = ∅
2.  while (R is not empty) do
3.      choose a request i∈R with minimum f(i) // greedy rule
4.      A = A + {i}
5.      R = R – {i} – X, where X = {j: j∈R and j is not compatible with i}
6.  return A
```

- Running time: From O(n^2) to O(n log n)
  - Initialization:
    - O(n log n): sort R in ascending order of $f(i)$
    - O(n): construct S, $S[i] = s(i)$
  - Lines 3 and 5:
    - O(n): scan R once
    - We always select the first interval in R
    - We do not delete all incompatible requests in line 5; we skip only those listed before the next selected interval.

The Interval Scheduling Problem



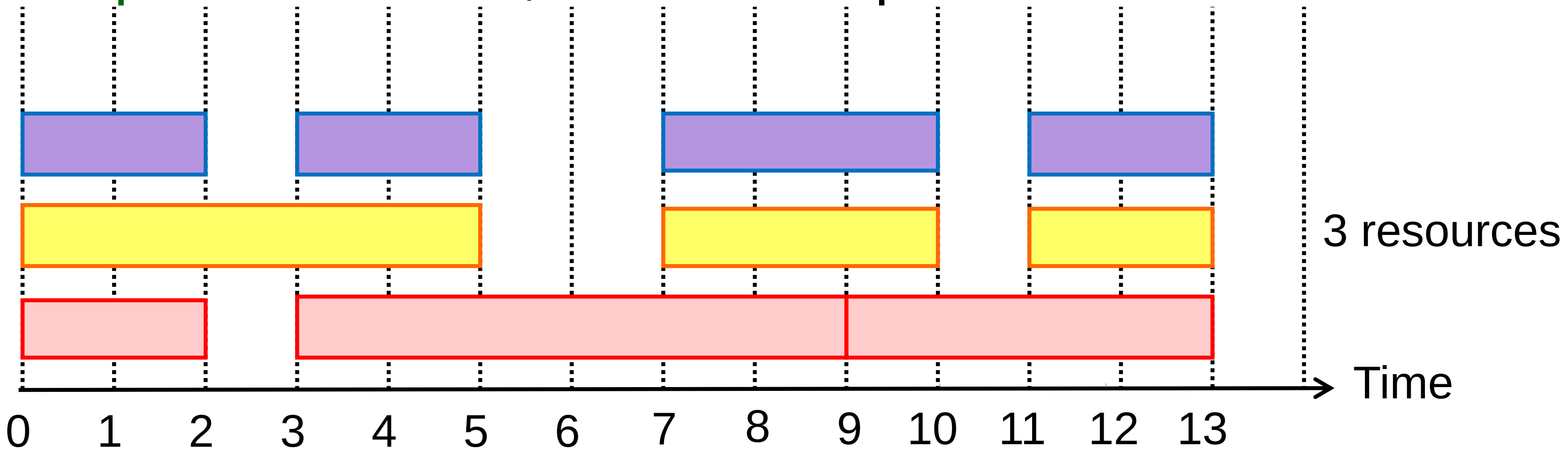
Maximum compatible subset {1, 3, 5, 8}

Interval Partitioning

Interval coloring

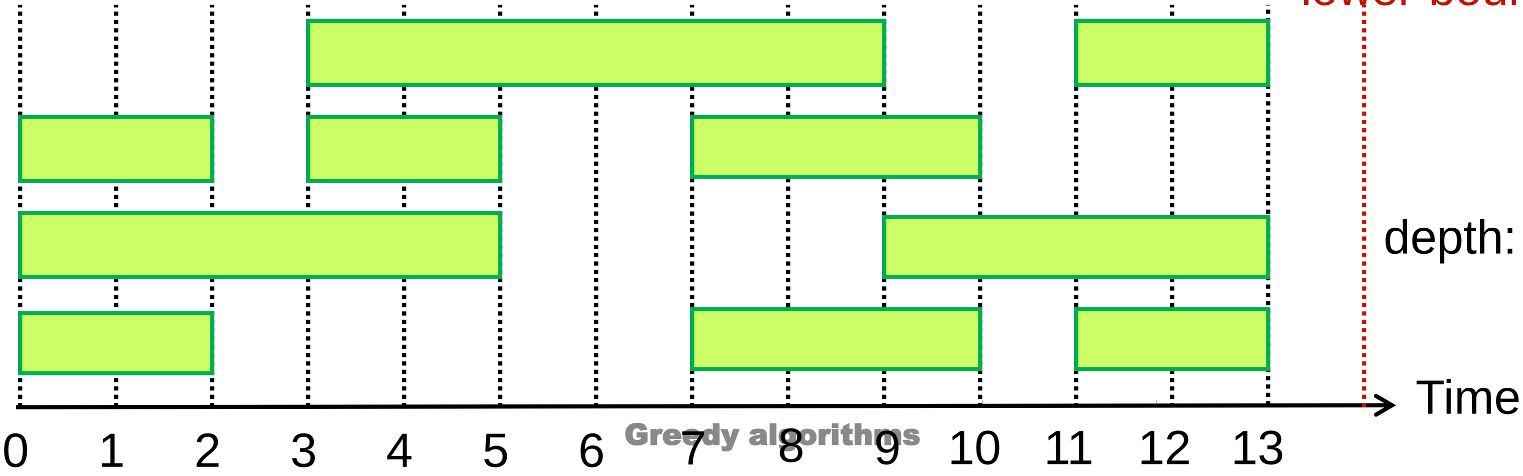
What if We Have Multiple Resources?

- The interval partitioning problem:
  - a.k.a. the interval coloring problem: one resource = one color
  - Use as few resources as possible
- Given: Set of requests {1, 2, …, n}, i^{th} request corresponds an interval with start time $s(i)$ and finish time $f(i)$
  - interval i: $[s(i), f(i))$
- Goal: Partition these requests into a minimum number of compatible subsets, each corresponds to one resource



How Many Resources Are Required?

- The depth of a set of intervals is the maximum number that pass over any single point on the time-line.
- In any instance of interval partitioning, the number of resources needed is at least the depth of the set of intervals.
- Pf:
  - Suppose a set of intervals has depth d, and let $I_1, \ldots, I_d$ all pass over a common point on the time-line.
  - Then each of these intervals must be scheduled on a different resource, so the whole instance needs at least d resources. lower bound



Can We Reach the Lower Bound?

- The depth *d* is the lower bound on the number of required resources.
- Q: Can we always use *d* resources to schedule all requests?
- A: Yes.

- Q: How to prove the optimality?
- A:

1. Find a bound that every possible solution must have at least a certain value
2. Show that the algorithm under consideration always achieves this bound

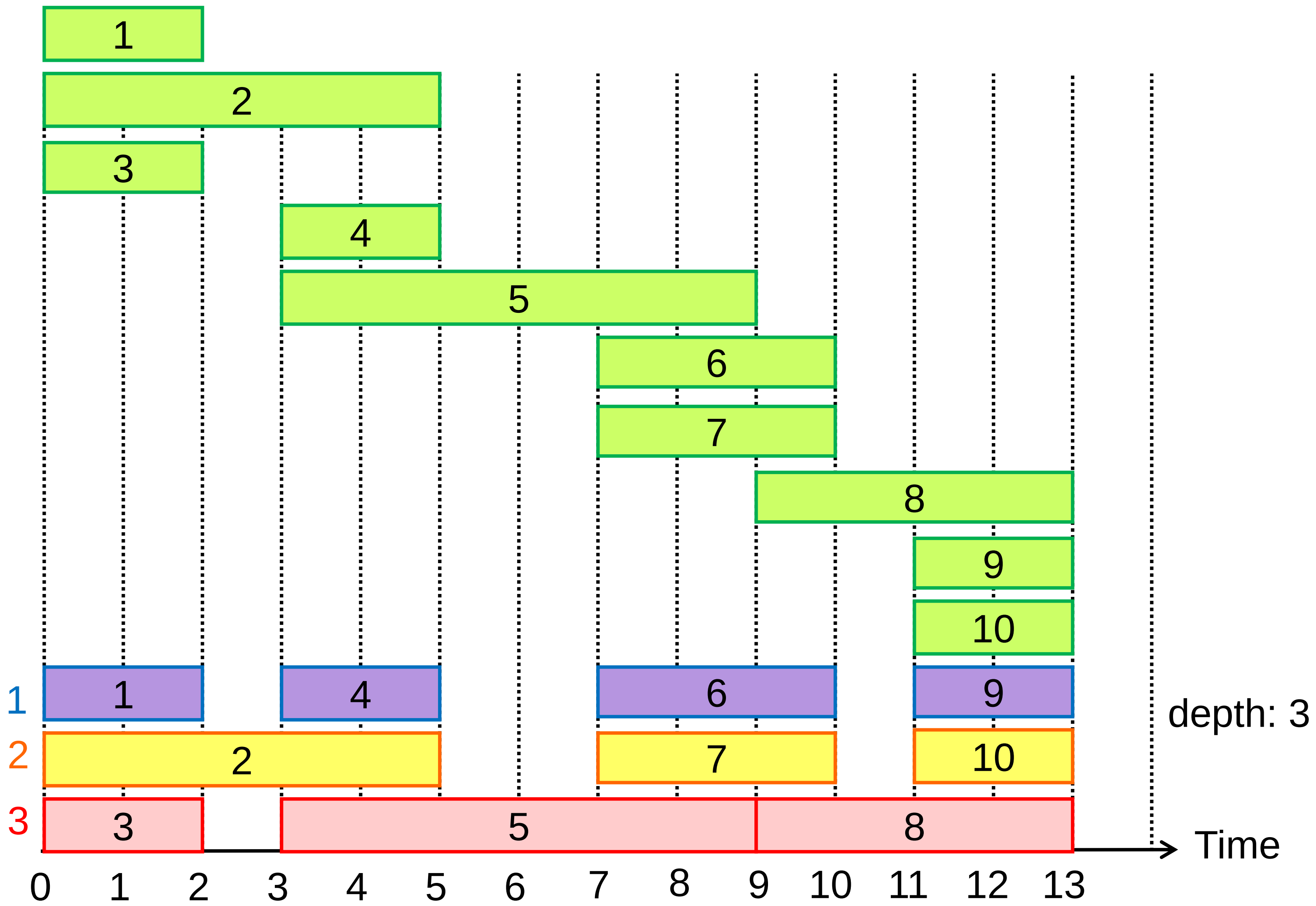
The Greedy Algorithm

- Assign a label to each interval. Possible labels: {1, 2, …, d}.
- Assign different labels for overlapping intervals.

Interval-Partitioning(R)
1. $\{I_1, \ldots, I_n\}$ = sort intervals in ascending order of their start times
2. **for** j **from** 1 **to** n **do**
3. exclude the labels of all assigned intervals that are not compatible with I_j
4. **if** (there is a nonexcluded label from {1, 2, …, d}) **then**
5. assign a nonexcluded label to I_j
6. **else** leave I_j unlabeled

- Implementation:
  - Lines 3--5: find a resource compatible with I_j, assign this label
    - Record the finish time of the last added interval for each label
    - Compatibility checking: $s(I_j) \geq f(\text{label}_i)$
      - Use priority queue to maintain labels

The Interval Partitioning Problem



Optimality (1/2)

- The greedy algorithm assigns every interval a label, and no two overlapping intervals receive the same label.
- Pf:
  1. No interval ends up unlabeled.
     - Suppose interval I_j overlaps t intervals earlier in the sorted list.
     - These t + 1 intervals pass over a common point, namely $s(I_j)$.
     - Hence, $t + 1 \le d$. Thus, $t \le d - 1$; at least one of the d labels is not excluded, and so there is a label that can be assigned to I_j.
       - i.e., line 6 never occurs!

Interval-Partitioning(R)
1. $\{I_1, \ldots, I_n\}$ = sort intervals in ascending order of their start times
2. **for** j **from** 1 **to** n **do**
3.   exclude the labels of all assigned intervals that are not compatible with I_j
4.   **if** (there is a nonexcluded label from $\{1, 2, \ldots, d\}$) **then**
5.   assign a nonexcluded label to I_j
6.   **else** leave I_j unlabeled

Optimality (2/2)

- Pf: (cont'd)
  2. No two overlapping intervals are assigned with the same label.
     - Consider any two intervals I_i and I_j that overlap, $i < j$.
     - When I_j is considered (in line 2), I_i is in the set of intervals whose labels are excluded (in line 3).
     - Hence, the algorithm will not assign the label used for I_i to I_j.
  - Since the algorithm uses d labels, we can conclude that the greedy algorithm always uses the minimum possible number of labels, i.e., it is optimal!

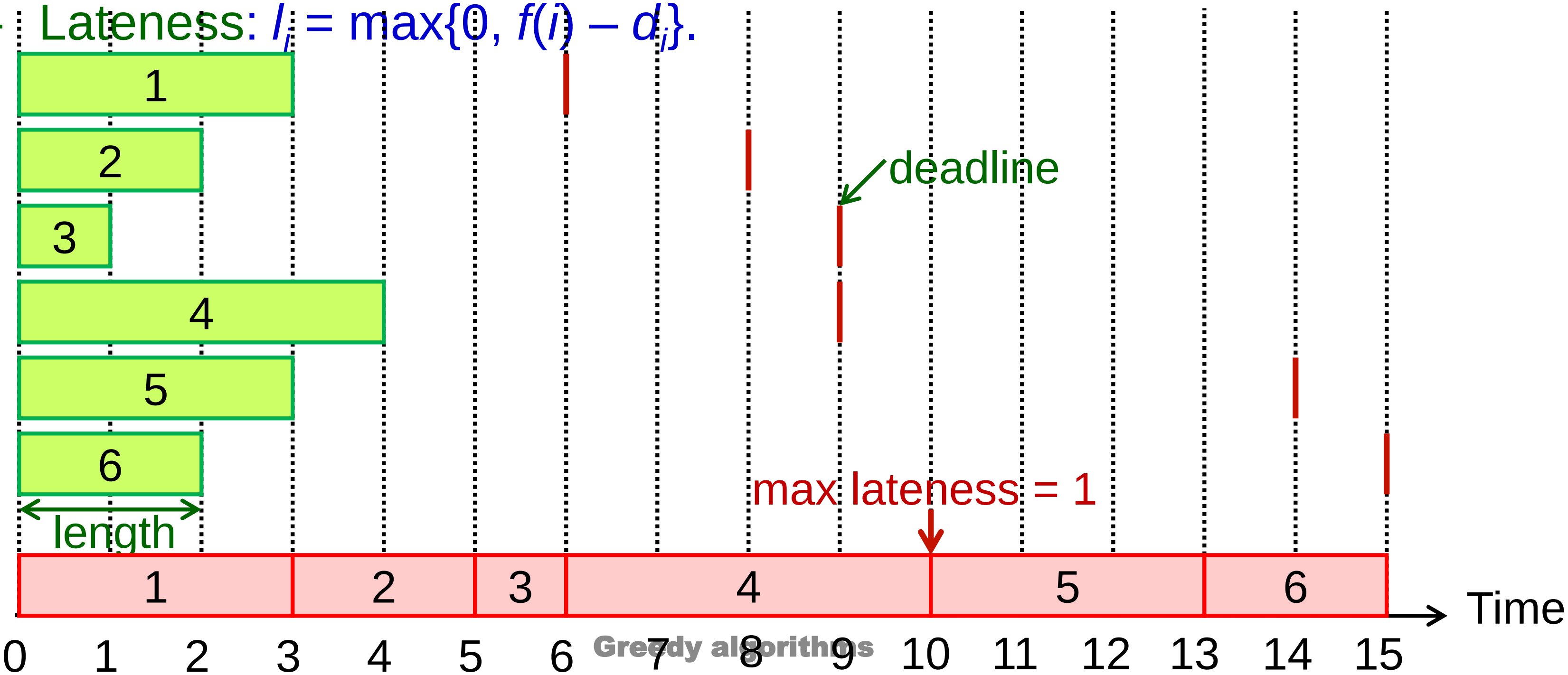
Interval-Partitioning(R)
1. $\{I_1, \ldots, I_n\}$ = sort intervals in ascending order of their start times
2. **for** j **from** 1 **to** n **do**
3.   exclude the labels of all assigned intervals that are not compatible with I_j
4.   **if** (there is a nonexcluded label from $\{1, 2, \ldots, d\}$) **then**
5.   assign a nonexcluded label to I_j
6.   **else** leave I_j unlabeled

Scheduling to Minimize Lateness

An exchange argument

What If Each Request Has a Deadline?

- Given: A single resource is available starting at time *s*. A set of requests {1, 2, …, *n*}, request *i* requires a contiguous interval of length t_i and has a deadline d_i.
- Goal: Schedule all requests without overlapping so as to minimize the maximum lateness.
  - Lateness: $l_i = \max\{0, f(i) - d_i\}$.



Greedy Rule

- Consider requests in some order.
  1. Shortest interval first: Process requests in ascending order of t_i

	1	2
t_i	1	10
d_i	100	10

  2. Smallest slack: Process requests in ascending order of $d_i - t_i$

	1	2
t_i	1	10
d_i	2	10

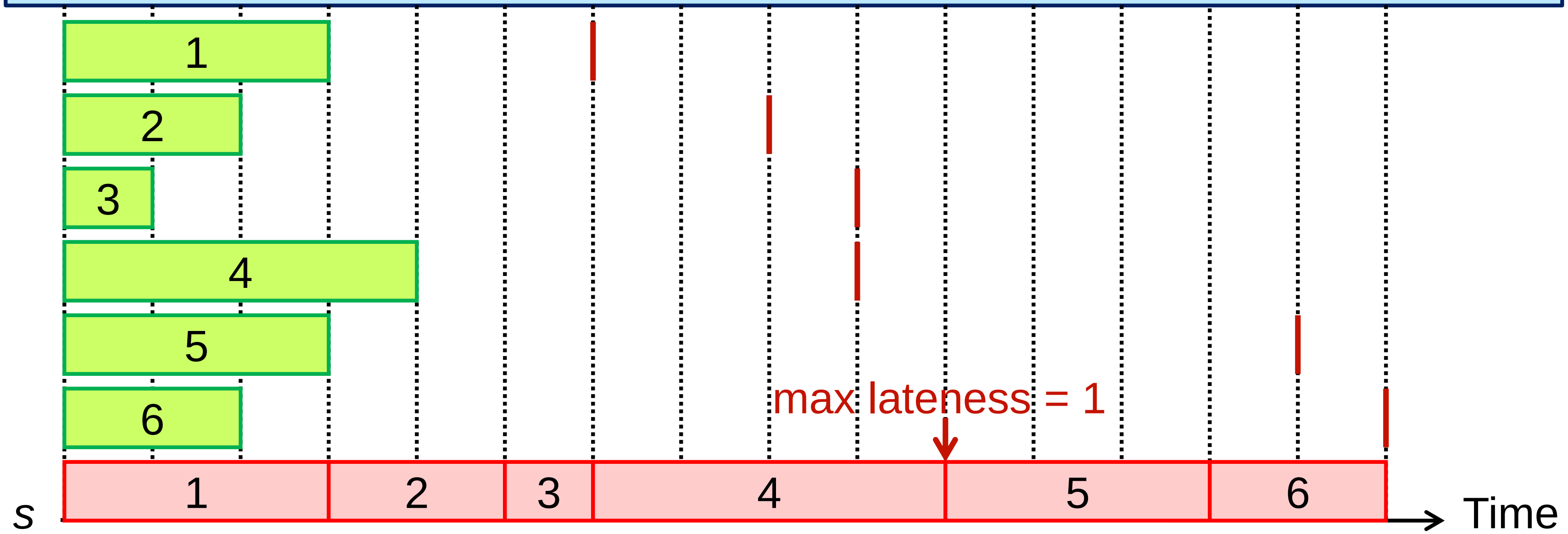
  3. Earliest deadline first: Process requests in ascending order of d_i

Jobs with earlier deadlines get completed earlier

Minimizing Lateness

- Greedy rule: Earliest deadline first!

Min-Lateness(R,s)
// f: the finishing time of the last scheduled request
1. $\{d_1, \ldots, d_n\}$ = sort requests in ascending order of their deadlines
2. $f = s$
3. **for** i **from** 1 **to** n **do**
4. assign request i to the time interval from $s(i) = f$ to $f(i) = f + t_i$
5. $f = f + t_i$
6. **return** the set of scheduled intervals $[s(i), f(i))$ for all i = 1..n

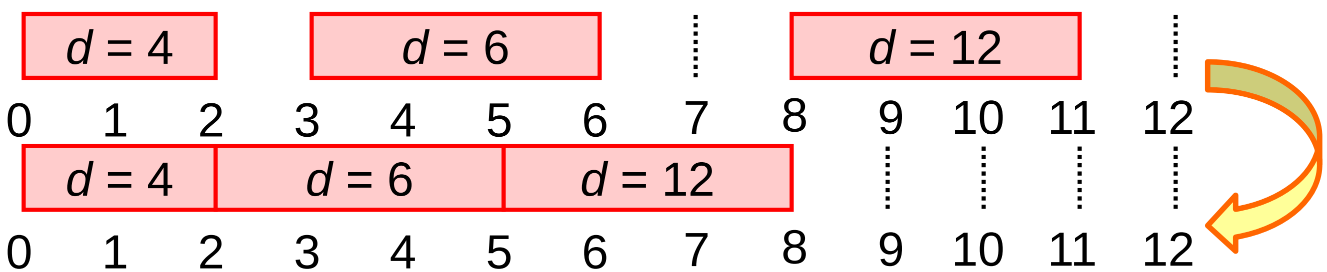


No Idle Time

- Observation: The greedy schedule has no idle time.
  - Line 4!

Min-Lateness(R,s)
// f: the finishing time of the last scheduled request
1. $\{d_1, \ldots, d_n\}$ = sort requests in ascending order of their deadlines
2. $f = s$
3. **for** i **from** 1 **to** n **do**
4. assign request i to the time interval from $s(i) = f$ to $f(i) = f + t_i$
5. $f = f + t_i$
6. **return** the set of scheduled intervals $[s(i), f(i))$ for all i = 1..n

- There is an optimal schedule with no idle time.

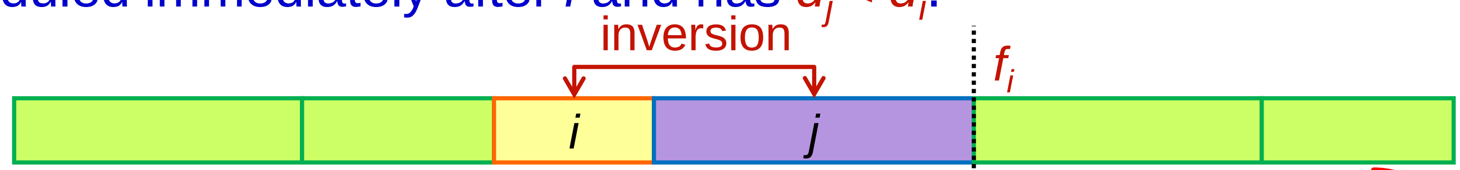


No Inversions

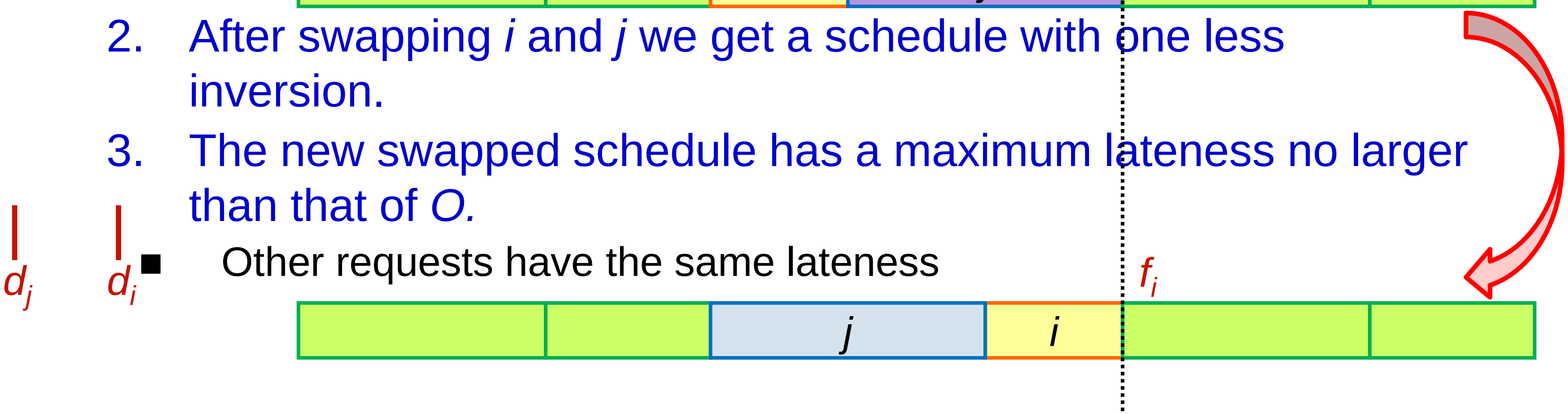
- Exchange argument: Gradually transform an optimal solution to the one found by the greedy algorithm without hurting its quality.
- An inversion in schedule S is a pair of requests i and j such that $s(i) < s(j)$ but $d_j < d_i$.
- All schedules without inversions and without idle time have the same maximum lateness.
- Pf:
  - If two different schedules have neither inversions nor idle time, then they can only differ in the order in which requests with identical deadlines are scheduled.
  - Consider such a deadline d. In both schedules, the jobs with deadline d are all scheduled consecutively (after all jobs with earlier deadlines and before all jobs with later deadlines).
  - Among them, the last one has the greatest lateness, and this lateness does not depend on the order of the requests.

Optimality

- There is an optimal schedule with no inversions and no idle time.
- Pf:
  - There is an optimal schedule *O* without idle time. (done!)
    1. If *O* has an inversion, there is a pair of jobs *i* and *j* such that *j* is scheduled immediately after *i* and has $d_j < d_i$.



    2. After swapping *i* and *j* we get a schedule with one less inversion.
    3. The new swapped schedule has a maximum lateness no larger than that of *O*.
       - Other requests have the same lateness



Optimality: Exchange Argument

- Theorem: The greedy schedule S is optimal.
- Pf: Proof by contradiction
  - Let O be an optimal schedule with inversions.
  - Assume O has no idle time.
  - If O has no inversions, then $S = O$. done!
  - If O 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.
    - This contradicts definition of O.

Summary: Greedy Analysis Strategies

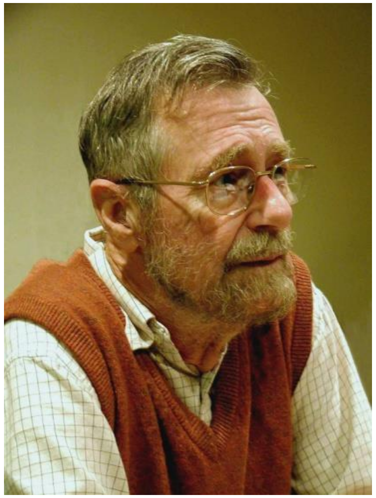
- An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion.
- It's challenging to prove greedy algorithms succeed in solving a nontrivial problem optimally.
  1. The greedy algorithm stays ahead: Show that after each step of the greedy algorithm, its partial solution is better than the optimal.
  2. An exchange argument: Gradually transform an optimal solution to the one found by the greedy algorithm without hurting its quality.

Shortest Paths

Edsger W. Dijkstra 1959

Edsger W. Dijkstra (1930—2002)

- 1972 Recipient of the ACM Turing Award



The question of whether computers can think is as relevant as the question of whether submarines can swim.

If you want more effective programmers,
you will discover that they should not waste their time debugging,
they should not introduce the bugs to start with.

Program testing can be a very effective way to show the presence of bugs,
but it is hopelessly inadequate for showing their absence.

-- Turing Award Lecture 1972, the humble programmer

http://www.cs.utexas.edu/users/EWD/obituary.html

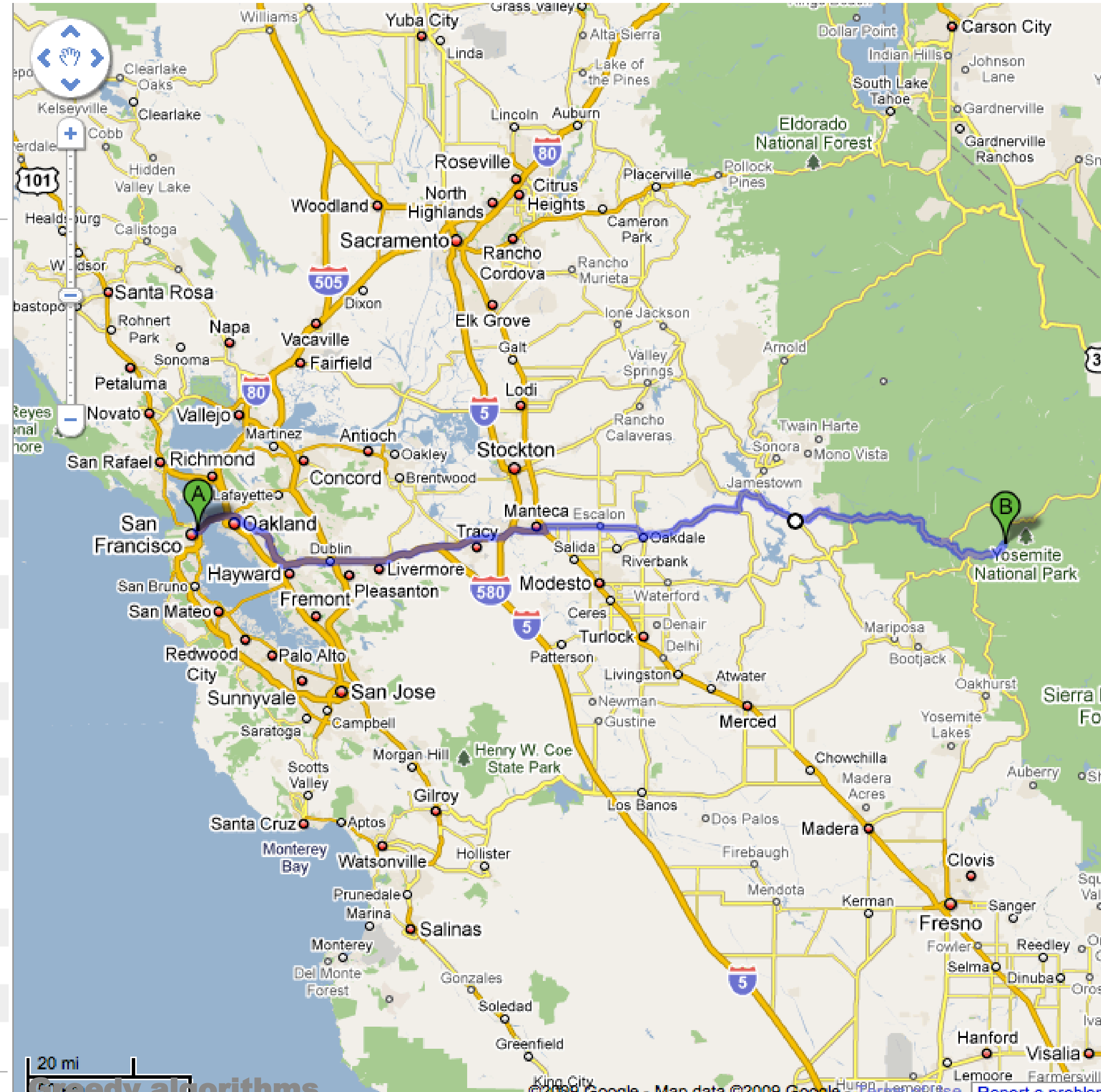
Google Map

Shortest path from
San Francisco Shopping Centre to
Yosemite National Park

A Westfield San Francisco Center
845 Market St, San Francisco, CA 94103-1923 - (415) 495-5656

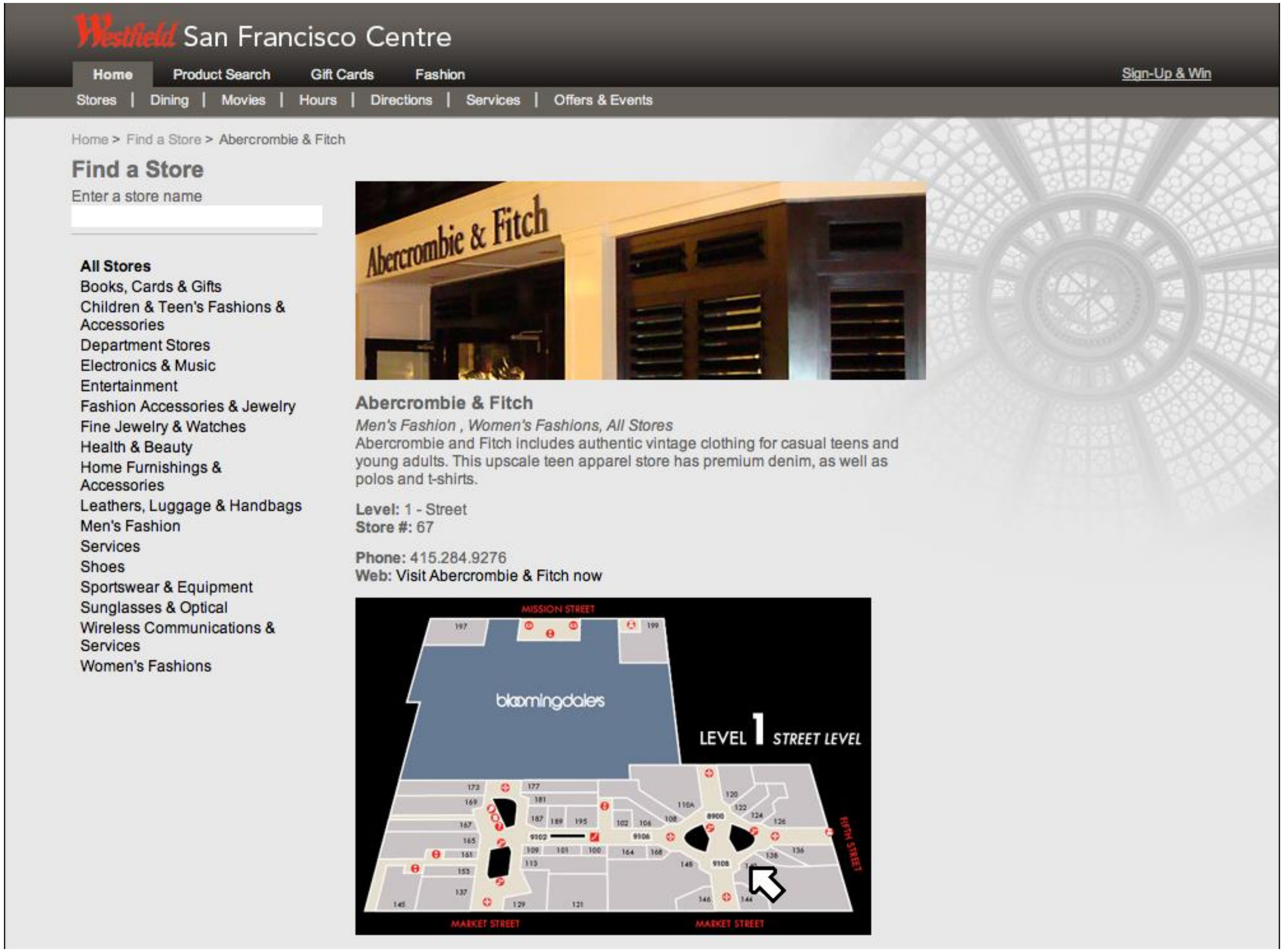
1.	Head **northeast** on **Mission St** toward **4th St** About 2 mins	go 0.6 mi total 0.6 mi
2.	Turn **right** at **1st St** About 2 mins	go 0.4 mi total 1.0 mi
3.	Slight **right** to merge onto **Bay Bridge/I-80 E** toward **Oakland** Continue to follow I-80 E About 6 mins	go 5.9 mi total 6.8 mi
4.	Take the exit onto **I-580 E** toward **CA-24/Hayward/Stockton** About 42 mins	go 46.2 mi total 53.0 mi
5.	Continue onto **I-205 E** About 14 mins	go 14.6 mi total 67.6 mi
6.	Merge onto **I-5 N** About 1 min	go 0.8 mi total 68.4 mi
7.	Take exit **461** to merge onto **CA-120 E** toward **Manteca/Sonora** About 6 mins	go 6.4 mi total 74.8 mi
8.	Take the exit onto **CA-120 E/CA-99 N** toward **Sacramento/Sonora N** About 2 mins	go 1.7 mi total 76.5 mi
9.	Take exit **242** for **CA-120 E/Yosemite Ave** toward **Sonora**	go 0.2 mi total 76.7 mi
10.	Turn **right** at **CA-120 E/Yosemite Ave** Continue to follow CA-120 E About 33 mins	go 19.8 mi total 96.5 mi
11.	Turn **left** at **CA-108 E/CA-120 E/E F St** Continue to follow CA-108 E/CA-120 E About 38 mins	go 25.0 mi total 121 mi
12.	Turn **right** at **CA-120 E** About 20 mins	go 12.6 mi total 134 mi
13.	Turn **right** at **Old Priest Grde** About 5 mins	go 1.8 mi total 136 mi
14.	Slight **left** at **CA-120 E** About 53 mins	go 34.8 mi total 171 mi
15.	Slight **right** at **Big Oak Rd** About 17 mins	go 9.4 mi total 180 mi
16.	Turn **left** at **CA-140 E/El Portal Rd** (signs for **Yosemite Valley/CA-41/Fresno**) About 2 mins	go 0.9 mi total 181 mi
17.	Take the 1st **right** onto **CA-140 E** About 8 mins	go 5.1 mi total 186 mi
18.	Turn **left** at **Sentinel Dr** About 2 mins	go 0.3 mi total 186 mi
19.	Continue onto **Village Dr** About 1 min	go 0.2 mi total 187 mi
20.	**Village Dr** turns slightly **right** and becomes **Ahwahnee Dr**	go 364 ft total 187 mi

B Yosemite National Park
9039 Village Dr, Yosemite Natl Pk, CA 95389 - (209) 372-0200



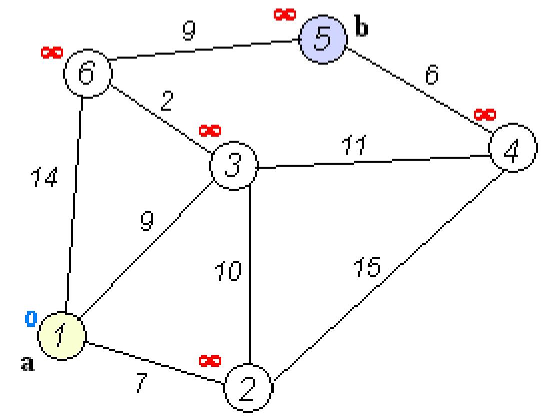
Floor Guide in a Shopping Mall

- Direct shoppers to their destinations in real-time



The Shortest Path Problem

- Given:
  - Directed graph $G = (V, E)$
    - Length l_e = length of edge $e = (u, v) \in E$
      - Distance; time; cost
      - $l_e \geq 0$
    - Q: what if undirected?
    - A: 1 undirected edge = 2 directed ones
  - Source s
- Goal:
  - Shortest path P_v from s to each other node $v \in V - \{s\}$
    - Length of path P: $l(P) = \sum_{e \in P} l_e$



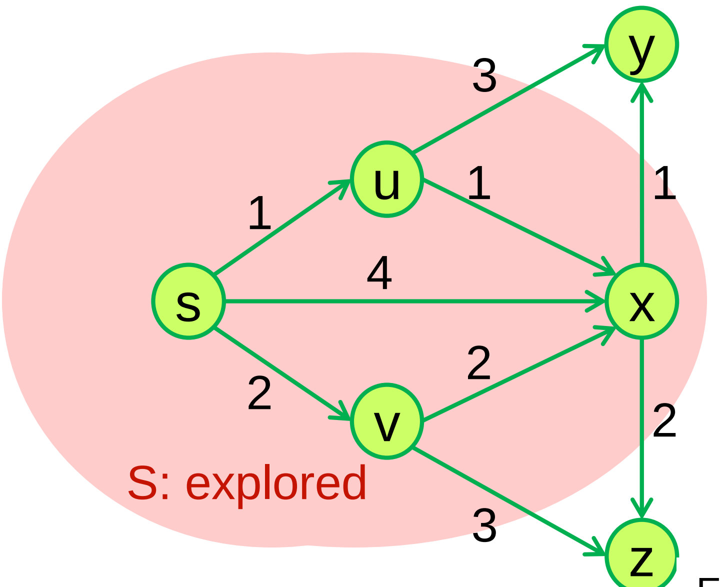
$l(a \to b) = l(1 \to 3 \to 6 \to 5)$
$= 9+2+9 = 20$

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

Dijkstra's Algorithm

Dijkstra(G,l)
// S: the set of explored nodes
// for each $u \in S$, we store a shortest path distance $d(u)$ from s to u
1. initialize $S = \{s\}$, $d(s) = 0$
2. **while** $S \neq V$ **do**
3. select a node $v \notin S$ with at least one edge from S for which
4. $d'(v) = \min_{e = (u, v):\, u \in S} (d(u)+l_e)$
5. add v to S and define $d(v) = d'(v)$

shortest path to some u in explored part, followed by a single edge (u, v)



$d'(x) = 2$; $d'(y) = 4$; $d'(z) = 5$

$d'(y) = 3$; $d'(z) = 4$

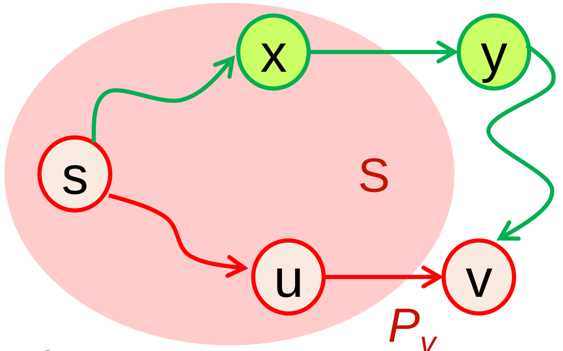
E. W. Dijkstra: *A note on two problems in connexion with graphs*. In *Numerische Mathematik*, 1 (1959), S. 269–271.

Correctness

The algorithm stays ahead!

Q: Why $l_e \geq 0$?

- Loop invariant: Consider the set S at any point in the algorithm's execution. For each node $u \in S$, $d(u)$ is the length of the shortest s-u path P_u.
- Pf: Proof by induction on $|S|$



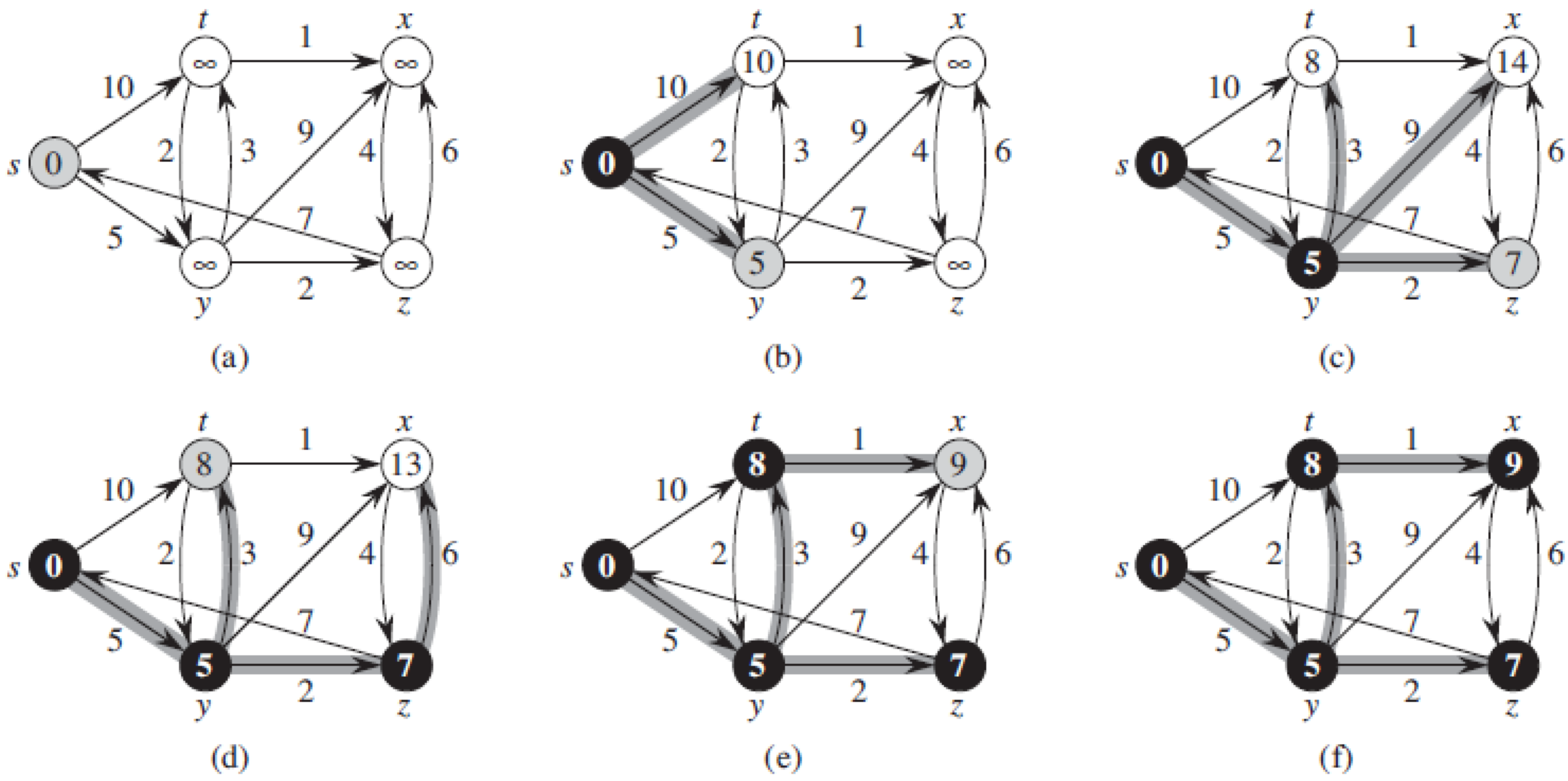
  - Basis step: trivial for $|S| = 1$.
  - Inductive step: hypothesis: true for $k \geq 1$.
    - Grow S by adding v;
      let (u, v) be the final edge on our s-v path P_v.
    - By induction hypothesis, P_u is the shortest s-u path.
    - Consider any other s-v path P; P must leave S somewhere; let y be the first node on P that is not in S, and $x \in S$ be the node just before y.
    - P cannot be shorter than P_v because it is already at least as long as P_v by the time it has left the set S.
    - At iteration k+1, $d(v) = d'(v) = d(u) + l_{e=(u, v)} \leq d(x) + l_{e'=(x, y)} \leq l(P)$

Implementation

- Q: How to do line 4 efficiently?
  $d'(v) = \min_{e = (u, v):\, u \in S} d(u) + l_e$
- A: Explicitly maintain $d'(v)$ in the view of each unexplored node v instead of S
  – Next node to explore = node with minimum $d'(v)$.
  – When exploring v, update $d'(w)$ for each outgoing (v, w), $w \notin S$.
- Q: How?
-

Operation	Dijkstra	Array	Binary heap	Fibonacci heap
Insert				
ExtractMin				
ChangeKey				
IsEmpty				
Total				

Example



Cormen, 3rd ed.

Recap Heaps: Priority Queues

Binary Tree Application

Priority Queue

- In a priority queue (PQ)
  - Each element has a priority (key)
  - Only the element with highest (or lowest) priority can be deleted
    - Max priority queue, or min priority queue
  - An element with arbitrary priority can be inserted into the queue at any time

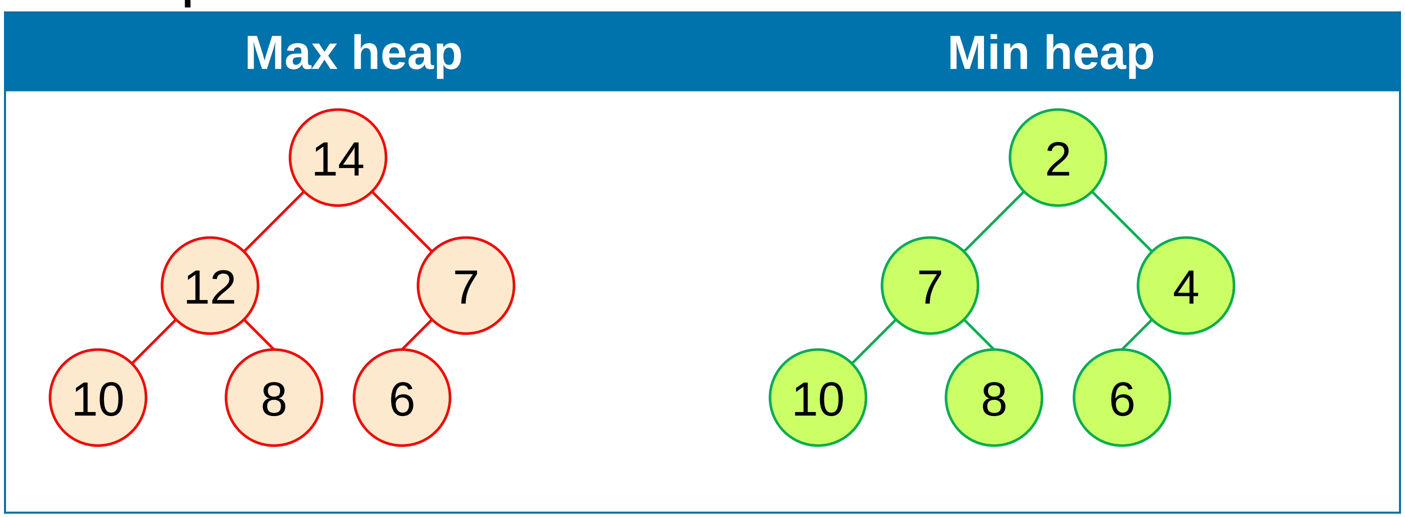
Operation	Binary heap	Fibonacci heap
FindMin	$\Theta(1)$	$\Theta(1)$
ExtractMin	$\Theta(\lg n)$	$O(\lg n)$
Insert	$\Theta(\lg n)$	$\Theta(1)$
ChangeKey	$\Theta(\lg n)$	$\Theta(1)$

The time complexities are worst-case time for binary heap, and amortized time complexity for Fibonacci heap

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein *Introduction to Algorithms, 2nd Edition. MIT Press and McGraw-Hill,* 2001.
Fredman M. L. & Tarjan R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* 34(3), pp. 596-615.

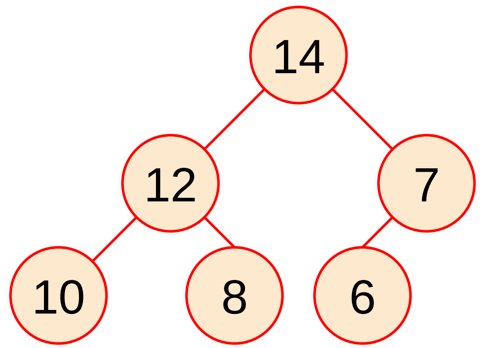
Heap

- Definition: A max (min) heap is
  - A max (min) tree: *key*[*parent*] >= (<=) *key*[*children*]
  - A complete binary tree
- Corollary: Who has the largest (smallest) key in a max (min) heap?
  - Root!
- Example



Class *MaxHeap*

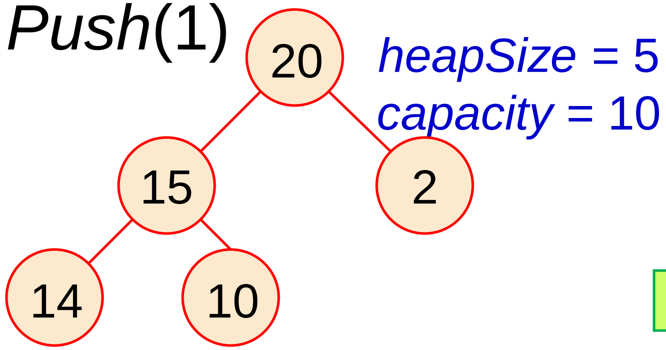
- Implementation?
  - Complete binary tree ⇒ array representation



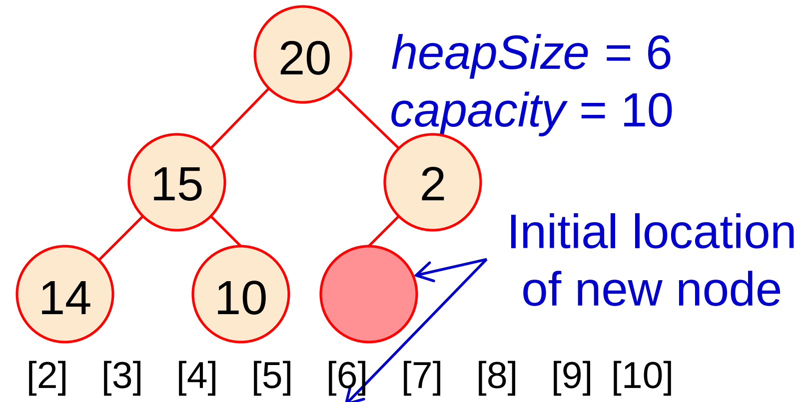
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
heap	-	14	12	7	10	8	6	-	-	-	-

Insertion into a Max Heap (1/3)

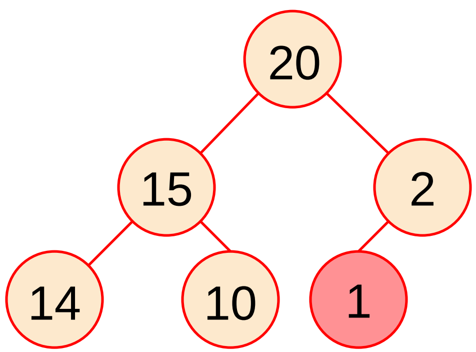
- Maintain heap property all the times
- *Push*(1)



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	20	15	2	14	10	-	-	-	-	-



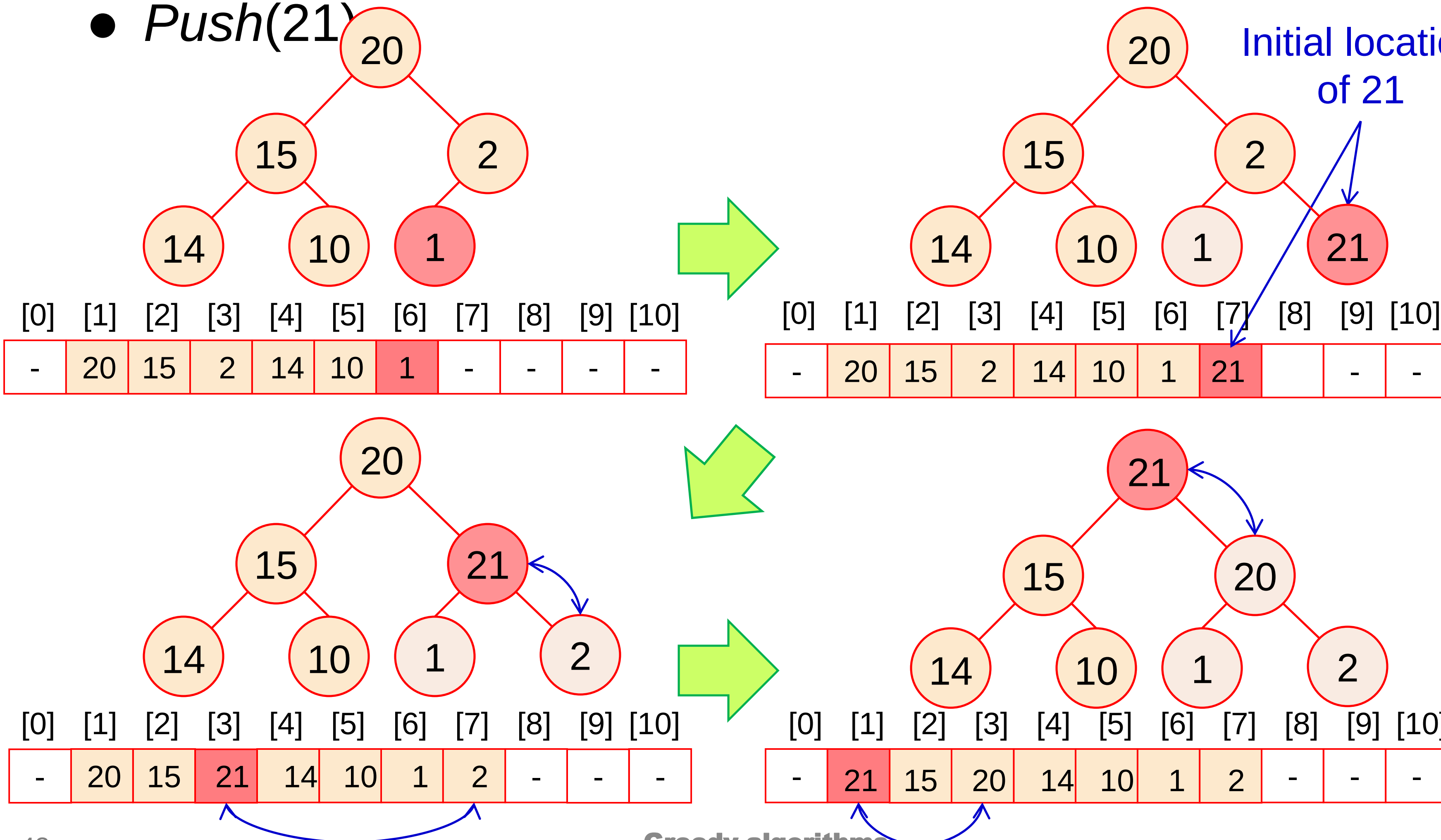
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	20	15	2	14	10		-	-	-	-



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	20	15	2	14	10	1	-	-	-	-

Insertion into a Max Heap (2/3)

- Maintain heap ⇒ bubble up if needed!
- *Push*(21)

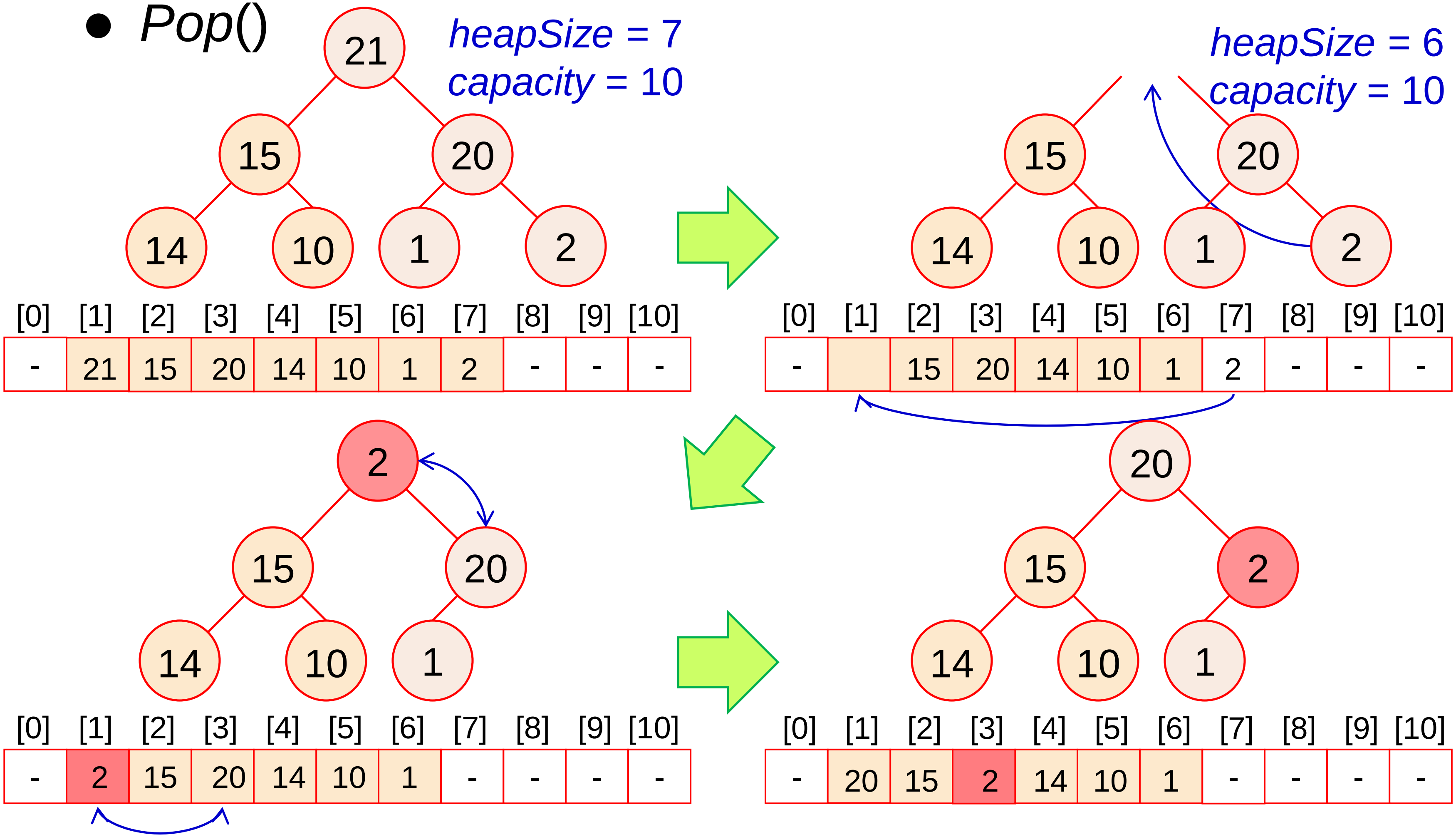


Insertion into a Max Heap (3/3)

- Time complexity?
  - How many times to bubble up in the worst case?
  - Tree height: $\Theta(\lg n)$

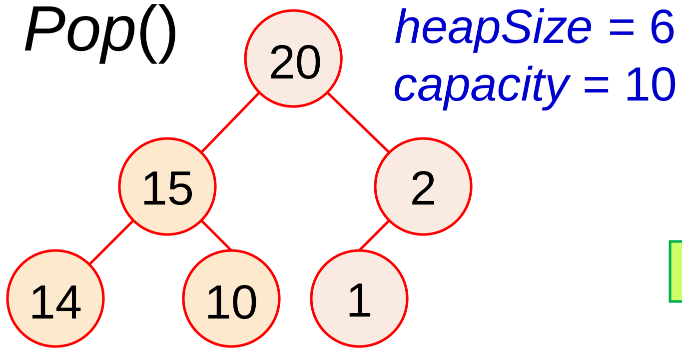
Deletion from a Max Heap (1/3)

- Maintain heap ⇒ trickle down if needed!
- *Pop*()

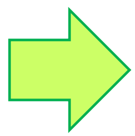


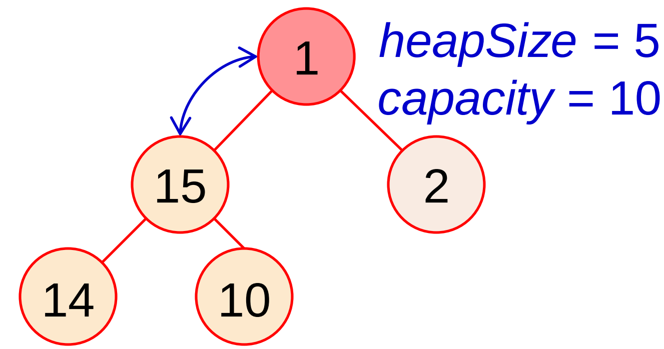
Deletion from a Max Heap (2/3)

- Maintain heap ⇒ trickle down if needed!
- *Pop*()

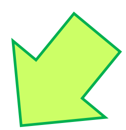


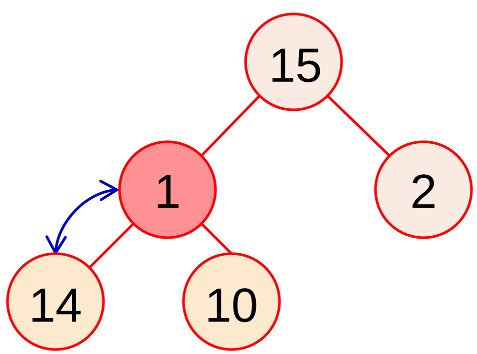
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	20	15	2	14	10	1	-	-	-	-



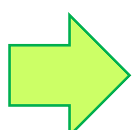


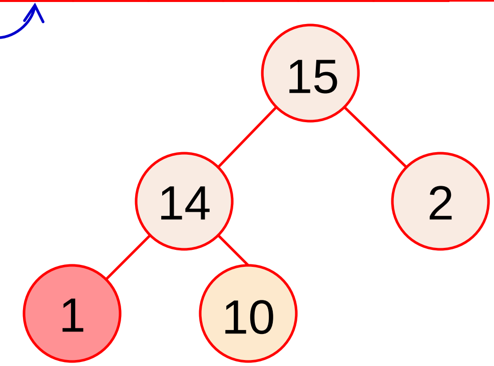
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	1	15	2	14	10	-	-	-	-	-





[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	15	1	2	14	10	-	-	-	-	-





[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	15	14	2	1	10	-	-	-	-	-

Deletion from a Max Heap (3/3)

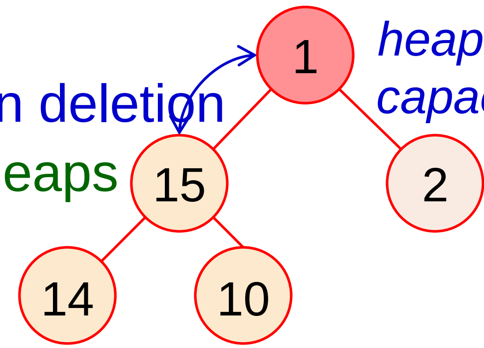
- Time complexity?
  - How many times to trickle down in the worst case? $\Theta(\lg n)$

Max Heapify

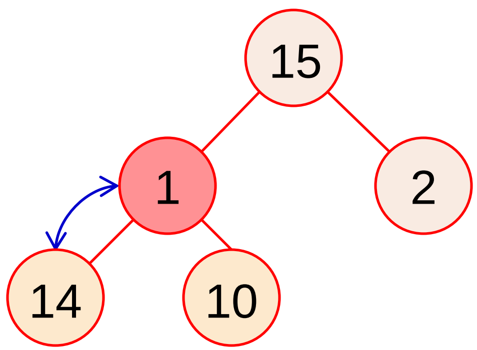
- Max (min) heapify = maintain the max (min) heap property
  - What we do to trickle down the root in deletion
  - Assume *i*’s left & right subtrees are heaps
    - But *key*[*i*] may be < (>) *key*[*children*]
  - Heapify *i* = trickle down *key*[*i*]

    ⇒ the tree rooted at *i* is a heap

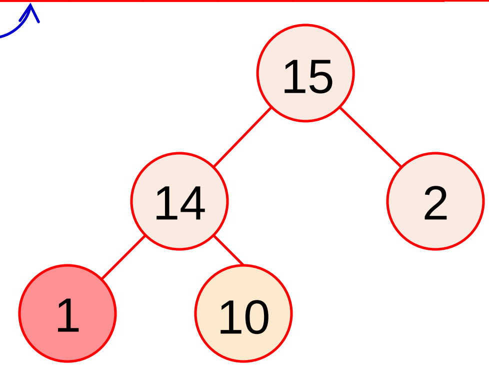
heapSize = 5
capacity = 10



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	1	15	2	14	10	-	-	-	-	-



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	15	1	2	14	10	-	-	-	-	-

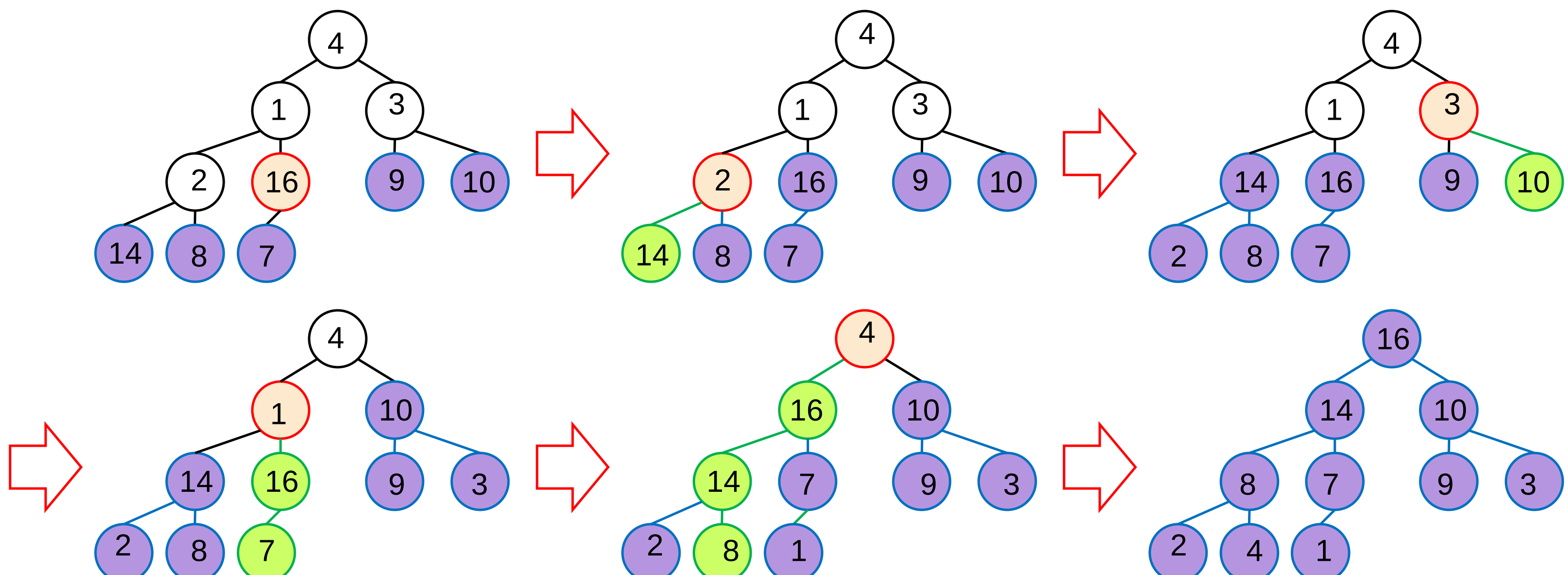


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	15	14	2	1	10	-	-	-	-	-

How to Build a Max Heap?

- How to convert any complete binary tree to a max heap?
- Intuition: Max heapify in a bottom-up manner
  - Induction basis: Leaves are already heaps
  - Inductive steps: Start at parents of leaves, work upward till root
  - Time complexity: $O(n \lg n)$

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
a	-	4	1	3	2	16	9	10	14	8	7

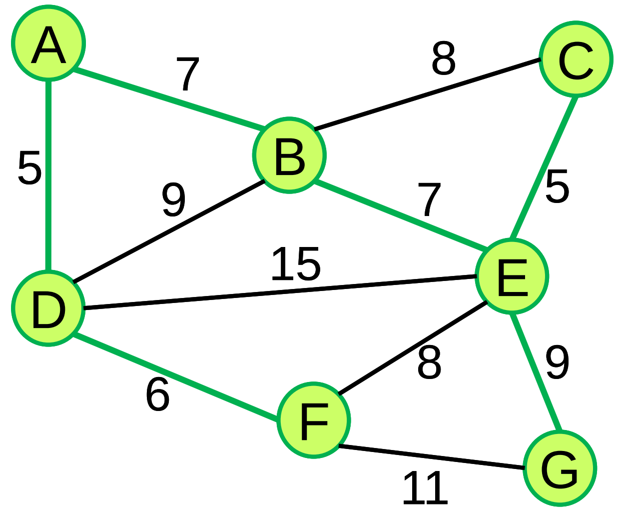


Minimum Spanning Trees

Robert C. Prim 1957 (Dijkstra 1959)
Joseph B. Kruskal 1956
Reverse-delete

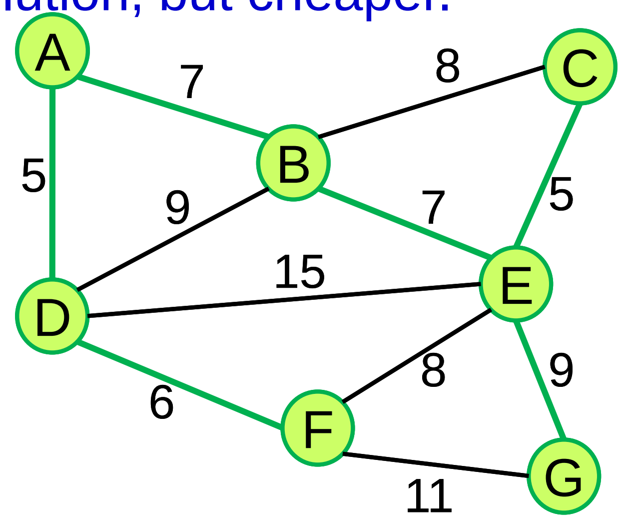
Minimum Spanning Graphs

- Q: How can a cable TV company lay cable to a new neighborhood, of course, as cheaply as possible?
- A: Curiously and fortunately, this problem is a case where many greedy algorithms optimally solve.
- Given
  - Undirected graph $G = (V, E)$
    - Nonnegative cost c_e for each edge $e = (u, v) \in V$
      - $c_e \geq 0$
- Goal
  - Find a subset of edges $T \subseteq E$ so that
    - The subgraph (V, T) is connected
    - Total cost $\Sigma_{e \in V}\, c_e$ is minimized

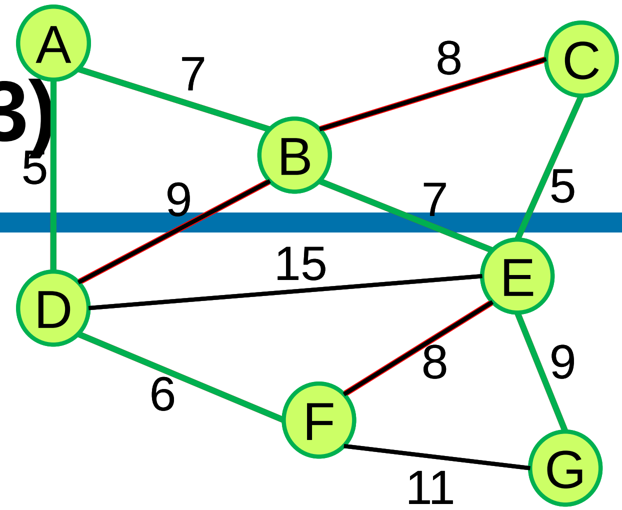


Minimum Spanning ????

- Q: Let T be a minimum-cost solution. What should (V, T) be?
- A:
  - By definition, (V, T) must be connected.
  - We show that it also contains no cycles.
  - Suppose it contained a cycle C, and let e be any edge on C.
  - We claim that $(V, T - \{e\})$ is still connected
  - Any path previously used e can now go path $C - \{e\}$ instead.
  - It follows that $(V, T - \{e\})$ is also a valid solution, but cheaper.
  - Hence, (V, T) is a tree.
- Goal
  - Find a subset of edges $T \subseteq E$ so that
    - (V, T) is a tree,
    - Total cost $\Sigma_{e \in V}\, c_e$ is minimized.



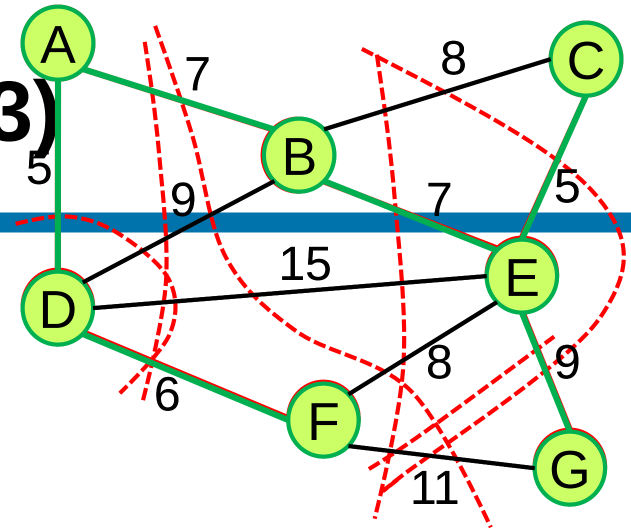
Greedy Algorithms (1/3)



- Q: What will you do?

- All three greedy algorithms produce an MST.
- Kruskal's algorithm:
  - Start with $T = \{\}$.
  - Consider edges in ascending order of cost.
  - Insert edge e in T as long as it does not create a cycle; otherwise, discard e and continue.

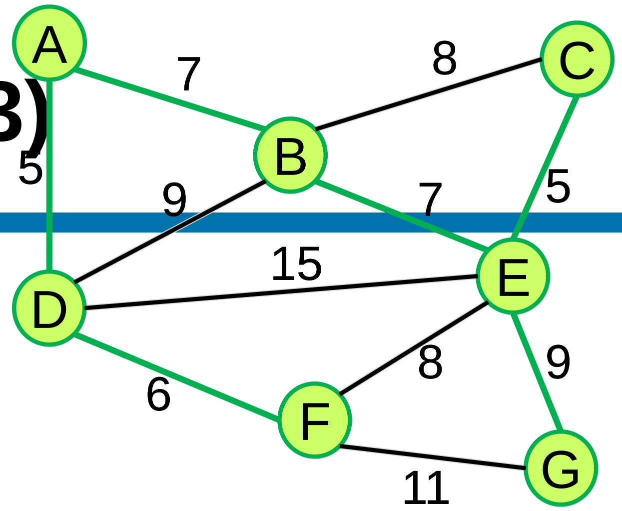
Greedy Algorithms (2/3)



- Q: What will you do?

- All three greedy algorithms produce an MST.
- Prim's algorithm: (c.f. Dijkstra's algorithm)
  - Start with a root node *s*.
  - Greedily grow a tree *T* from *s* outward.
  - At each step, add the cheapest edge *e* to the partial tree *T* that has exactly one endpoint in *T*.

Greedy Algorithms (3/3)



- Q: What will you do?

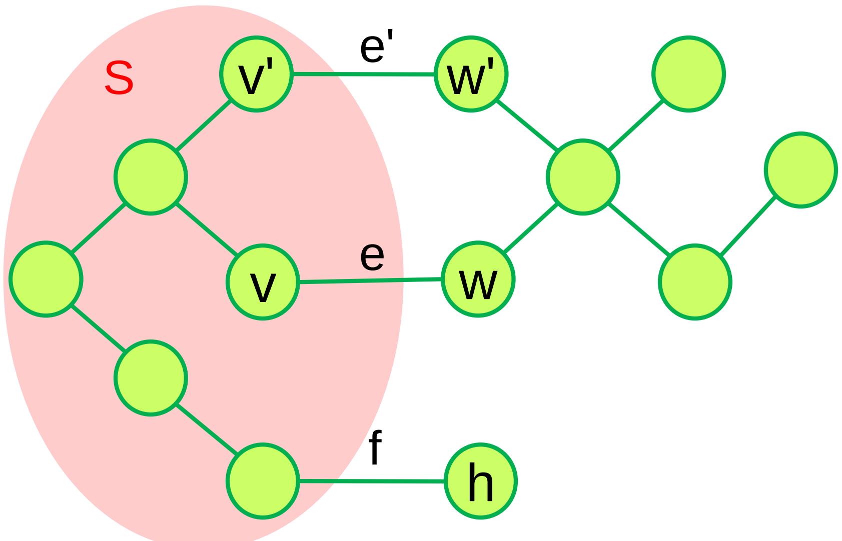
- All three greedy algorithms produce an MST.
- Reverse-delete algorithm: (reverse of Kruskal's algorithm)
  – Start with *T* = *E*.
  – Consider edges in descending order of cost.
  – Delete edge *e* from *T* unless doing so would disconnect *T*.

Cut Property (1/3)

- Simplifying assumption: All edge costs c_e are distinct.
- Q: When is it safe to include an edge in the MST?
- Cut Property: Let S be any subset of nodes, and let $e = (v, w)$ be the minimum cost edge with one end in S and the other in $V–S$. Then every MST contains e.
- Pf: Exchange argument!
  - Let T be a spanning tree that does not contain e. We need to show that T does not have the minimum possible cost.
  - Since T is a spanning tree, it must contain an edge f with one end in S and the other in $V–S$.
  - Since e is the cheapest edge with this property, we have $c_e < c_f$.
  - Hence, $T – \{f\} + \{e\}$ is a spanning tree that is cheaper than T.
- Q: What's wrong with this proof?
- A: Take care about the definition!

Cut Property (2/3)

- Q: What’s wrong with this proof?
- A: Take care about the definition!
  - Spanning tree: connected & acyclic



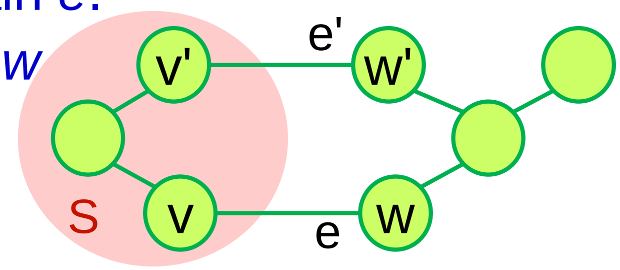
Cut Property (3/3)

Optimality of Kruskal's & Prim's algorithms!

- Cut Property: Let S be any subset of nodes, and let $e = (v, w)$ be the minimum cost edge with one end in S and the other in V-S. Then every MST contains e.
- Pf: Exchange argument!
  - Let T be a spanning tree that does not contain e.
  - T is a spanning tree; $\exists$ path $P \in T$ from v to w
  - Let $e' = (v', w')$ on P, $v' \in S$ and $w' \in V-S$.
  - $T' = T - \{e'\} + \{e\}$ is a spanning tree
    - (V, T') must be connected:

      (V, T) is connected, any path in (V, T) using e' can be rerouted in (V, T') by $v' \rightarrow v$, (v, w), $w \rightarrow w'$.
    - (V, T') must be acyclic:

      The only cycle in $(V, T' + \{e'\})$ is $e + P$, it isn't in (V, T')
  - Since $c_e < c_{e'}$, T' is cheaper than T.



Cycle Property

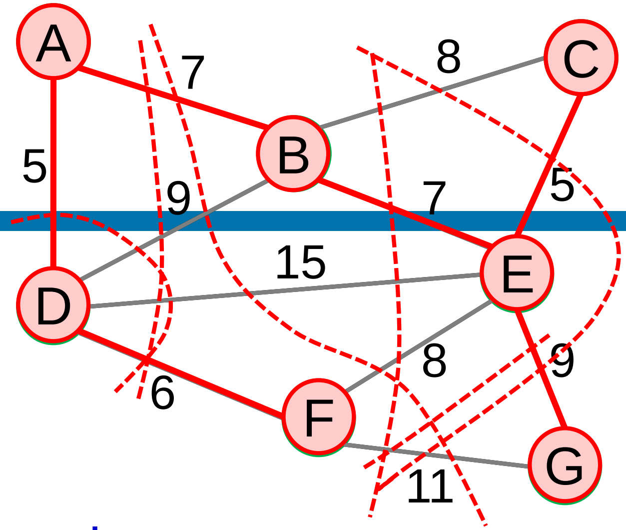
Optimality of Reverse-delete algorithm!

- Q: When is it safe to exclude an edge out?
- Cycle Property: Let *C* be any cycle in *G*, and let $e = (v, w)$ be the maximum cost edge in *C*. Then *e* does not belong to any MST.
- Pf: Exchange argument! (Similar to Cut Property)

Implementing MST Algorithms

Priority queue
Union-find

Prim's Example



- R. C. Prim, 1957
- Procedure:
  - Start with a root node *s*.
  - Greedily grow a tree *T* from *s* outward.
  - At each step, add the cheapest edge *e* to the partial tree *T* that has exactly one endpoint in *T*.

Prim's Algorithm

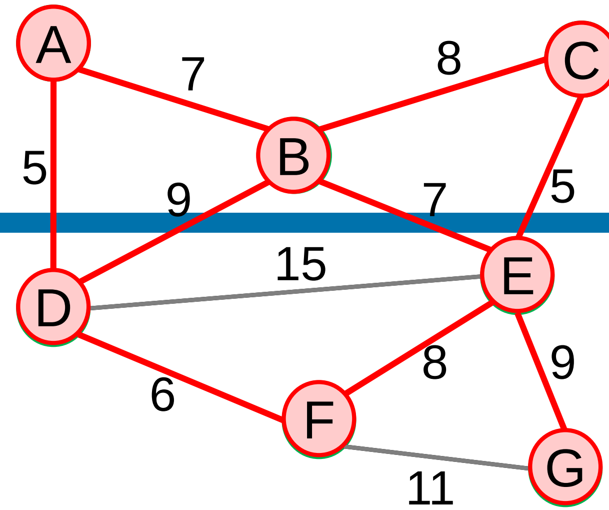
Dijkstra(G,l)
// S: the set of explored nodes
// for each $u \in S$, we store a shortest path distance $d(u)$ from s to u
1. initialize $S = \{s\}$, $d(s) = 0$
2. **while** $S \neq V$ **do**
3. select a node $v \notin S$ with at least one edge from S for which
4. $d'(v) = \min_{e = (u, v):\, u \in S} d(u) + l_e$
5. add v to S and define $d(v) = d'(v)$

- Q: How to change Dijkstra's algorithm to Prim's?
- Q: How to implement?

R. C. Prim: *Shortest connection networks and some generalizations*. In *Bell System Technical Journal*, 36 (1957), pp. 1389–1401.

E. W. Dijkstra: *A note on two problems in connexion with graphs*. In *Numerische Mathematik*, 1 (1959), S. pp. 269–271.

Kruskal's Algorithm



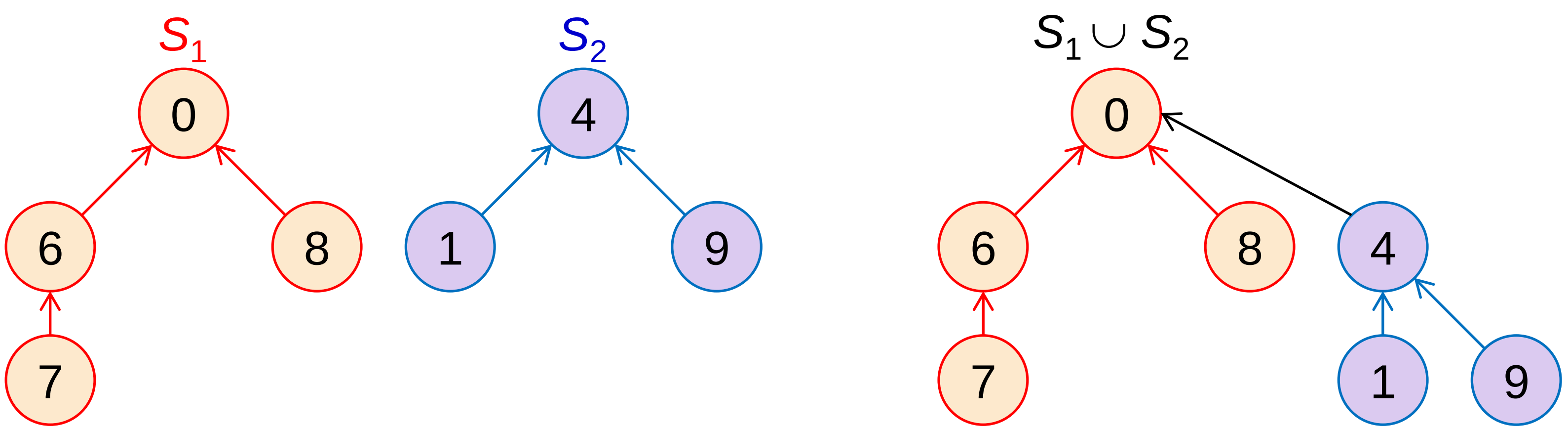
- J. B. Kruskal, 1956
- Procedure:
  - Start with $T = \{\}$.
  - Consider edges in ascending order of cost.
  - Insert edge e in T as long as it does not create a cycle; otherwise, discard e and continue.

Kruskal(G,c)
1. $\{e_1, e_2, \ldots, e_m\}$ = sort edges in ascending order of their costs
2. $T = \{\}$
3. **for each** $e_i=(u, v)$ **do**
4. **if** (u and v are not connected by edges in T) **then** // different subtrees
5. $T = T + \{e_i\}$ // merge these two corresponding subtrees

J. B. Kruskal: *On the shortest spanning subtree of a graph and the traveling salesman problem*. In *Proceedings of the American Mathematical Society*, 7(1) (Feb, 1956), pp. 48–50.

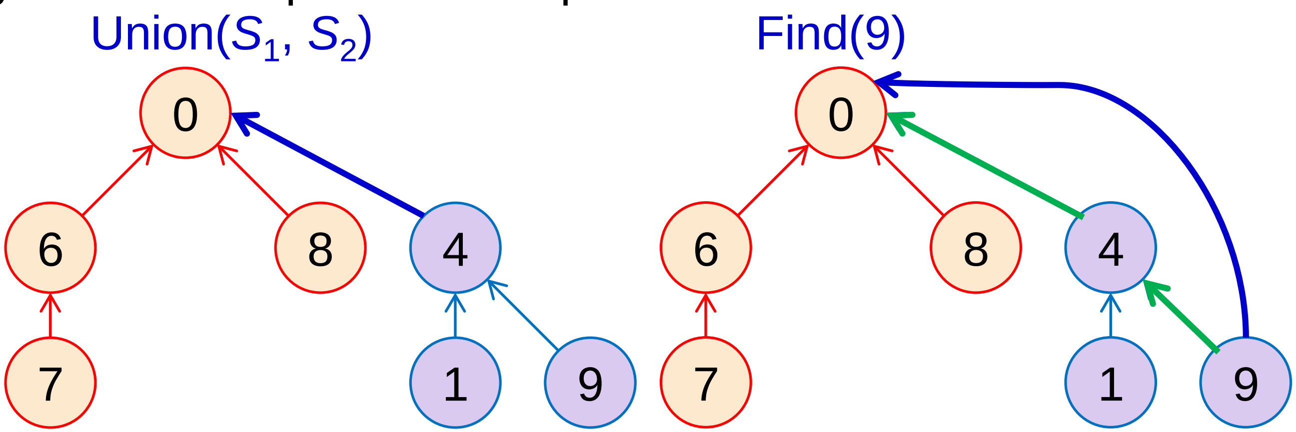
The Union-Find Data Structure (1/2)

- Union-find data structure represents disjoint sets
  - Disjoint sets: elements are disjoint
  - Each set has a representative
  - Operations:
    - MakeUnionFind(S): initialize a set for each element in S
    - Find(u): return the representative of the set containing u
    - Union(A, B): merge sets A and B



The Union-Find Data Structure (2/2)

- Implementation: disjoint-set forest
  - Representative is the root; link: from children to parent
  - Union: attach the smaller to the larger one (union by rank)
  - Find: trace back to root and redirect the link (path compression)
- Running time: union by rank + path compression
  - The amortized running time per operation is $O(\alpha(n))$, $\alpha(n) < 5$!!
    - Average running time of a sequence of n operations



B. A. Galler & M. J. Fischer. An improved equivalence algorithm. *Comm. of the ACM,* 7(5), (May 1964), pp. 301–303.

R. E. Tarjan & J. van Leeuwen. Worst-case analysis of set union algorithms. Journal of the ACM, 31(2), pp. 245–281, 1984.

Implementing Kruskal's Algorithm

```
Kruskal(G,c)
1.  {e_1, e_2, …, e_m} = sort edges in ascending order of their costs
2.  T = {}
3.  for each e_i =(u, v) do
4.      if (u and v are not connected by edges in T) then // different subtrees
5.          T = T + {e_i} // merge these two corresponding subtrees
```

- Use the union-find data structure
  - Maintain a disjoint set for each connected component (subtree)

  - Line 1: sort edge costs
  - Line 4: "Find" twice for each edge (total m edges in G)
  - Line 5: "Union" possibly once for each edge (total n-1 edges in T)

  - Comparison sort + simple disjoint set: $O(m \log n)$
  - Linear sort + union-find: $O(m\, \alpha(m, n))$