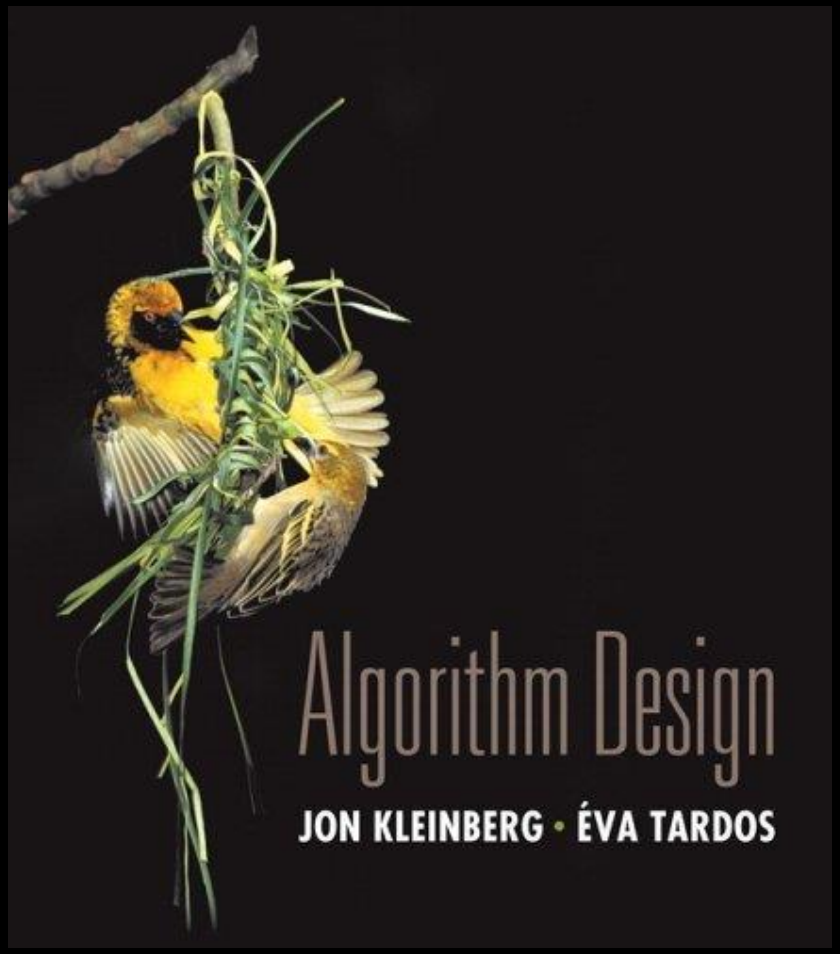


Chapter 4: Part I

Greedy Algorithms

CS 350 Winter 2018

A few more exercises from Ch. 3

Chapter 3: #3.6

Suppose that for a connected graph $G=(V,E)$, BFS and DFS, each rooted at u, produce the same tree, T. Prove that $G=T$ (i.e., G cannot contain any edges that don't belong to T).

Chapter 3: #3.6

Suppose that for a connected graph $G=(V,E)$, BFS and DFS, each rooted at u, produce the same tree, T. Prove that $G=T$ (i.e., G cannot contain any edges that don't belong to T).

Pf. (by contradiction) Where to start?

Chapter 3: #3.6

Suppose that for a connected graph G=(V,E), BFS and DFS, each rooted at u, produce the same tree, T. Prove that G=T (i.e., G cannot contain any edges that don't belong to T).

Pf. (by contradiction)

Assume not, and suppose that G has an edge e=(u,v) that doesn't not belong to T.

But then the distance between node u and v is at most 1 (why?). This leads to a contradiction…(where?).

Chapter 3: #3.7

Claim: Let G be a graph on n nodes, where n is an even number. If every node of G has degree at least n/2, then G is connected.

True or False? (How would we check?)

Chapter 3: #3.7

Claim: Let G be a graph on n nodes, where n is an even number. If every node of G has degree at least n/2, then G is connected.

True or False? (How would we check?) Try it out yourself.

Chapter 3: #3.7

Claim: Let G be a graph on n nodes, where n is an even number. If every node of G has degree at least n/2, then G is connected.

True or False? (How would we check?) Try it out yourself.

Pf. (by contradiction) Suppose not.

Let G have at least two connected components with the property that every node has degree at least n/2.

Now suppose v is a vertex in a component S with minimum size.

Then $|S| \leq n/2$. Now derive a contradiction!

4.0 Classic Greedy Problems

A Greedy Template

```
Greedy(input I)
 begin
   while (solution is not complete) do
            Select the best element x in the
                     remaining input I;
            Put x next in the output;
            Remove x from the remaining input;
   endwhile
End
```

(*) The notion of "best" has to be defined in each problem separately.
(*) Therefore, the essence of each greedy algorithm is the selection policy.
(*) Idea: At each step we myopically optimize some underlying criterion.
(*) One can often design many different greedy algorithms for the same problem, each one locally, incrementally optimizing some different measure (i.e. a heuristic) on its way to a solution.
(*) Some advantages: greedy algorithms can be extremely simple and intuitive and yet quite powerful – even optimal!

A Greedy Template

(2) Basic Methods for proving that a greedy algorithm produces an optimal solution:

(1) Show that it "stays ahead": meaning, we show that it does better than any other algorithm at each step.

(2) Use an "exchange argument": one considers any possible solution to the problem and gradually transforms it into the solution found by the greedy algorithm – without hurting its quality.

Some classic greedy problems: selection sort, knapsack problem, interval scheduling, optimal caching, MST (minimum spanning tree), shortest path problems, Huffman codes, clustering.

Selection Sort

Idea: The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list.

Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Selection Sort

Sorted sublist == ( )

Unsorted sublist == (11, 25, 12, 22, 64)

Least element in unsorted list == 11

Sorted sublist == (11)

Unsorted sublist == (25, 12, 22, 64)

Least element in unsorted list == 12

Sorted sublist == (11, 12)

Unsorted sublist == (25, 22, 64)

Least element in unsorted list == 22

Sorted sublist == (11, 12, 22)

Unsorted sublist == (25, 64)

Least element in unsorted list == 25

Sorted sublist == (11, 12, 22, 25)

Unsorted sublist == (64)

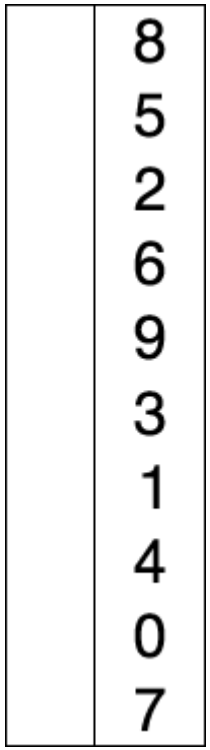
Least element in unsorted list == 64

Sorted sublist == (11, 12, 22, 25, 64)

Unsorted sublist == ( )

Selection Sort

```
/* a[0] to a[n-1] is the array to sort */
 2 int i,j;
 3 int n;
 4
 5 /* advance the position through the entire array */
 6 /*   (could do j < n-1 because single element is also min element) */
 7 for (j = 0; j < n-1; j++)
 8 {
 9     /* find the min element in the unsorted a[j .. n-1] */
10
11     /* assume the min is the first element */
12     int iMin = j;
13     /* test against elements after j to find the smallest */
14     for (i = j+1; i < n; i++)
15     {
16         /* if this element is less, then it is the new minimum */
17         if (a[i] < a[iMin])
18         {
19             /* found new minimum; remember its index */
20             iMin = i;
21         }
22     }
23
24     if (iMin != j)
25     {
26         swap(a[j], a[iMin]);
27     }
28 }
```



Selection Sort

Q: What is the run-time of SS?

Selecting the minimum requires scanning n elements (taking n-1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining n-1 elements and so on. Therefore, the total number of comparisons is:

Selection Sort

Q: What is the run-time of SS?

Selecting the minimum requires scanning n elements (taking n-1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining n-1 elements and so on. Therefore, the total number of comparisons is:

$$(n-1)+(n-2)+\ldots+1=$$

Selection Sort

Q: What is the run-time of SS?

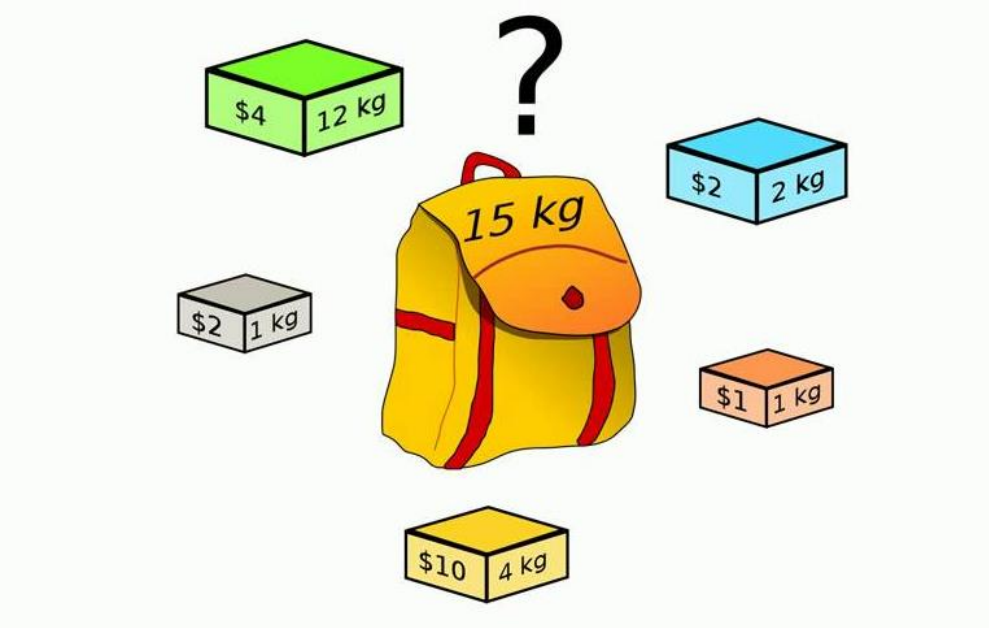
Selecting the minimum requires scanning n elements (taking n-1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining n-1 elements and so on. Therefore, the total number of comparisons is:

$$(n-1)+(n-2)+\ldots+1 = n\cdot n - \sum_{i=1}^{n-1} i = n^2 - \frac{n(n+1)}{2} = \frac{1}{2}\left(n^2-n\right) = O\left(n^2\right)$$

(not the best sorting algorithm available)

Knapsack Problem

The **knapsack problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.



Knapsack Problem

Problem: John wishes to take n items on a trip

- The weight of item *i* is w_i & items are all different (0/1 Knapsack Problem)
- The items are to be carried in a knapsack whose weight capacity is *c*
  - When sum of item weights ≤ c, all n items can be carried in the knapsack
  - When sum of item weights > c, some items must be left behind

Which items should be taken/left?

Knapsack Problem

John assigns a profit p_i to item i

All weights and profits are positive numbers

John wants to select a subset of the n items to take

- The weight of the subset should not exceed the capacity of the knapsack (constraint)
- Cannot select a fraction of an item (constraint)
- The profit of the subset is the sum of the profits of the selected items (optimization function)
- The profit of the selected subset should be maximum (optimization criterion)

Let $x_i = 1$ when item i is selected and $x_i = 0$ when item i is not selected

Because this is a 0/1 Knapsack Problem, you can choose the item or not choose it.

Apply greedy method:

Greedy attempt on capacity utilization

- Greedy criterion: select items in increasing order of weight
- When n = 2, c = 7, w = [3, 6], p = [2, 10], if only item 1 is selected ➔ profit of selection is 2 ➔ not best selection!

Greedy attempt on profit earned

- Greedy criterion: select items in decreasing order of profit
- When n = 3, c = 7, w = [7, 3, 2], p = [10, 8, 6], if only item 1 is selected ➔ profit of selection is 10 ➔ not best selection!

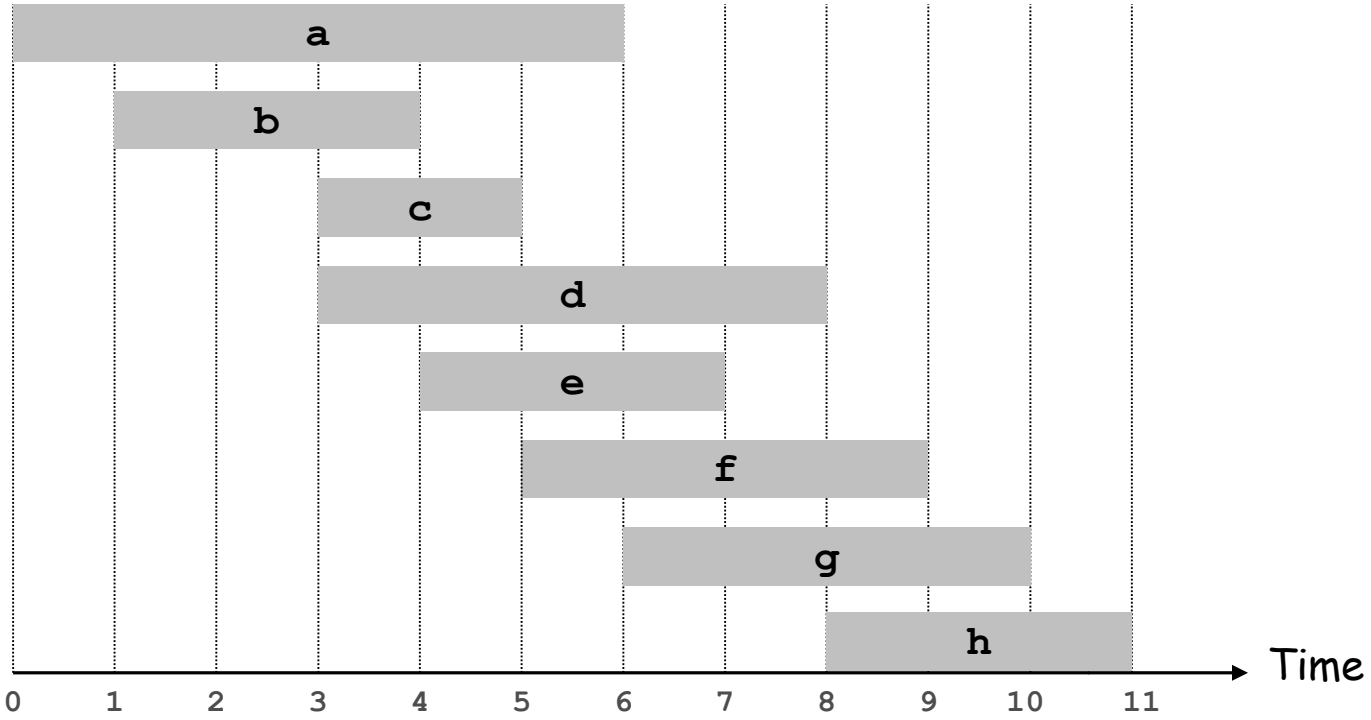
Moral: Greedy is not always best – other methods, including dynamic programming and genetic algorithms are better for this problem; knapsack problem is in general very difficult (NP-complete).

4.1.1 Interval Scheduling

Interval Scheduling

Interval scheduling.

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



Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some natural order.
Take each job provided it's compatible with the ones already taken.

- [Earliest start time] Consider jobs in ascending order of s_j.
- [Earliest finish time] Consider jobs in ascending order of f_j.
- [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.
- [Fewest conflicts] For each job j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some natural order.
Take each job provided it's compatible with the ones already taken.

Each of these approaches turn out to be sub-optimal!

counterexample for earliest start time

counterexample for shortest interval

counterexample for fewest conflicts

Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

A ← φ        (set of jobs selected)
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Implementation. O(n log n). (aside: O(n log n) is required due to sorting step, see: mergesort lecture later in course)

- Remember job j* that was added last to A.
- Job j is compatible with A if $s_j \geq f_{j^*}$.

Interval Scheduling: Greedy Algorithm

```
Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.


A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Claim: A is a compatible set of requests.

(trivial, why?)

Interval Scheduling: Greedy Algorithm

```
Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.


A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Claim: Solution is optimal.

Notation: let $i_1,\ldots,i_k$ be the set of requests in A in the order they were added to A; so $|A|=k$. Similarly, let $j_1,\ldots,j_m$ be the set of requests in O, an optimal set of intervals. Assume the requests in I are also ordered in the natural left-to-right order of the corresponding finish points.

Interval Scheduling: Greedy Algorithm

```
Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.


A ← ϕ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Claim: Solution is optimal!

Note: for all indices $r \leq k$, it follows that: $f(i_r) \leq f(j_r)$, where f() indicates the finish time of the interval (greediness guarantees -- can prove this with induction).

Interval Scheduling: Greedy Algorithm

```
Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.


A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Claim: Solution is optimal! $f(i_r) \le f(j_r)$

Pf. (By contradiction) If *A* is not optimal, then an optimal set *O* must have more requests, that is we must have $m > k$...

Interval Scheduling: Greedy Algorithm

```
Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.


A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Claim: Solution is optimal! (*) $f(i_r) \leq f(j_r)$

Pf. (By contradiction) If *A* is **not optimal**, then an optimal set *O* must have more requests, that is we must have $m > k$.

Applying (*) above, with $r = k$, we get that $f(i_k) \leq f(j_k)$.

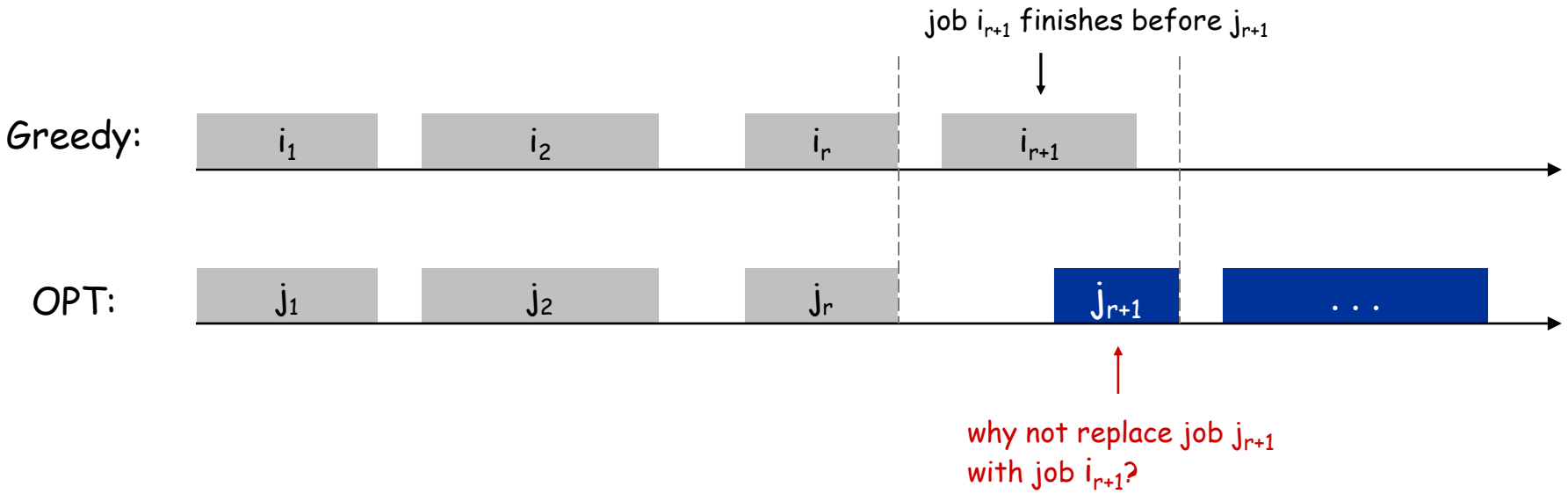
Since $m > k$, there is a request j_{k+1} in *O*. This request starts after request j_k ends, and hence after i_k ends.
So, after deleting all requests that are not compatible with requests: $i_1,\ldots,i_k$, the set of possible rests R still contains j_{k+1}. (contradiction)

Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal. (greedy algorithm "stays ahead" of an optimal solution)

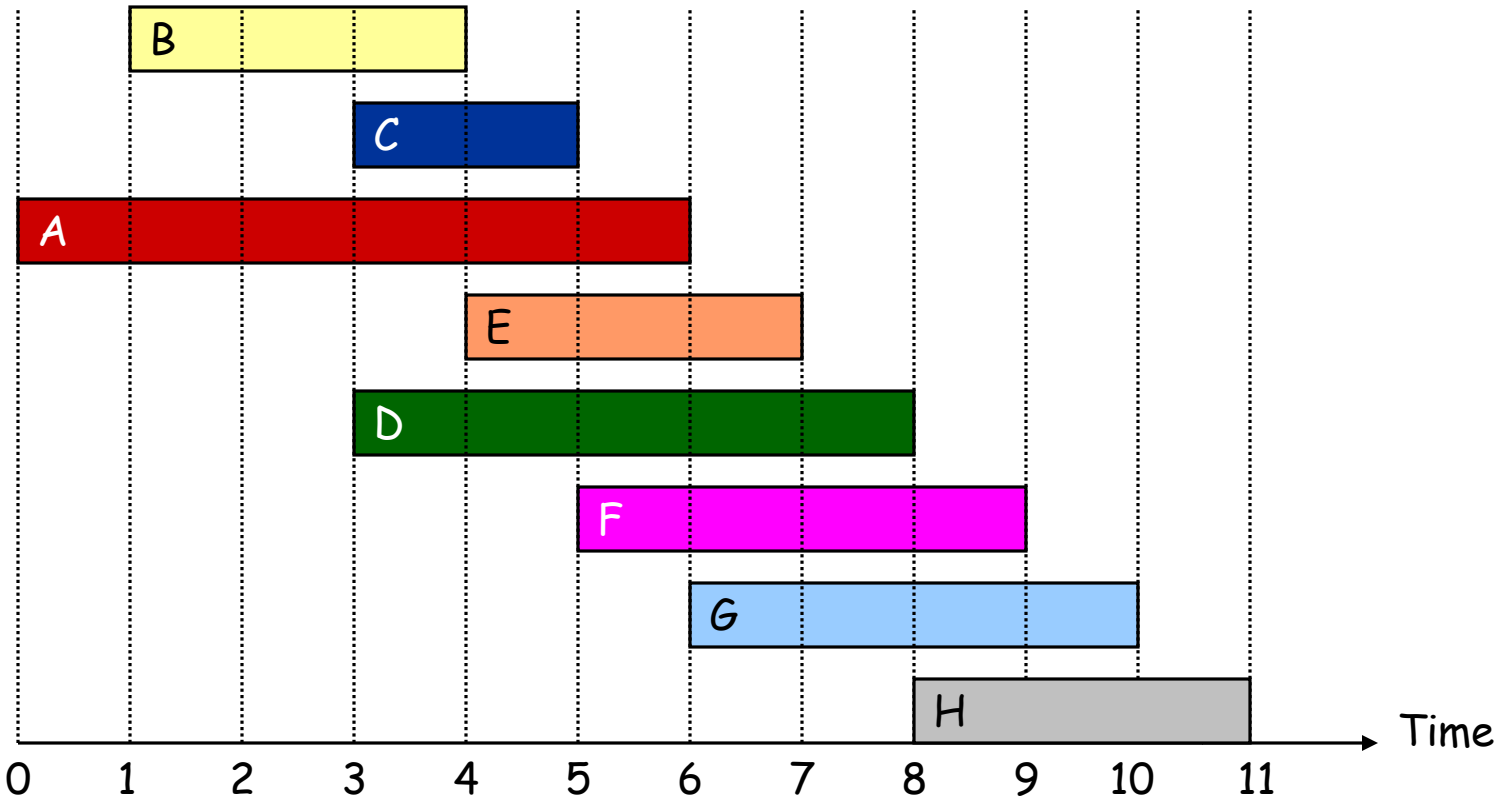
Pf. (by contradiction)

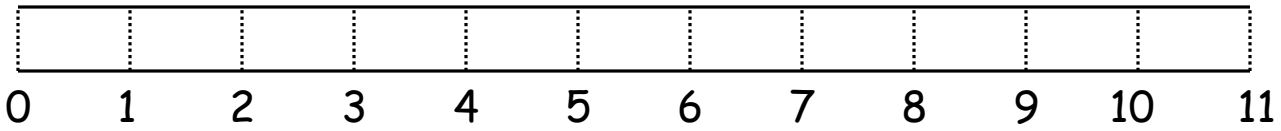
- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



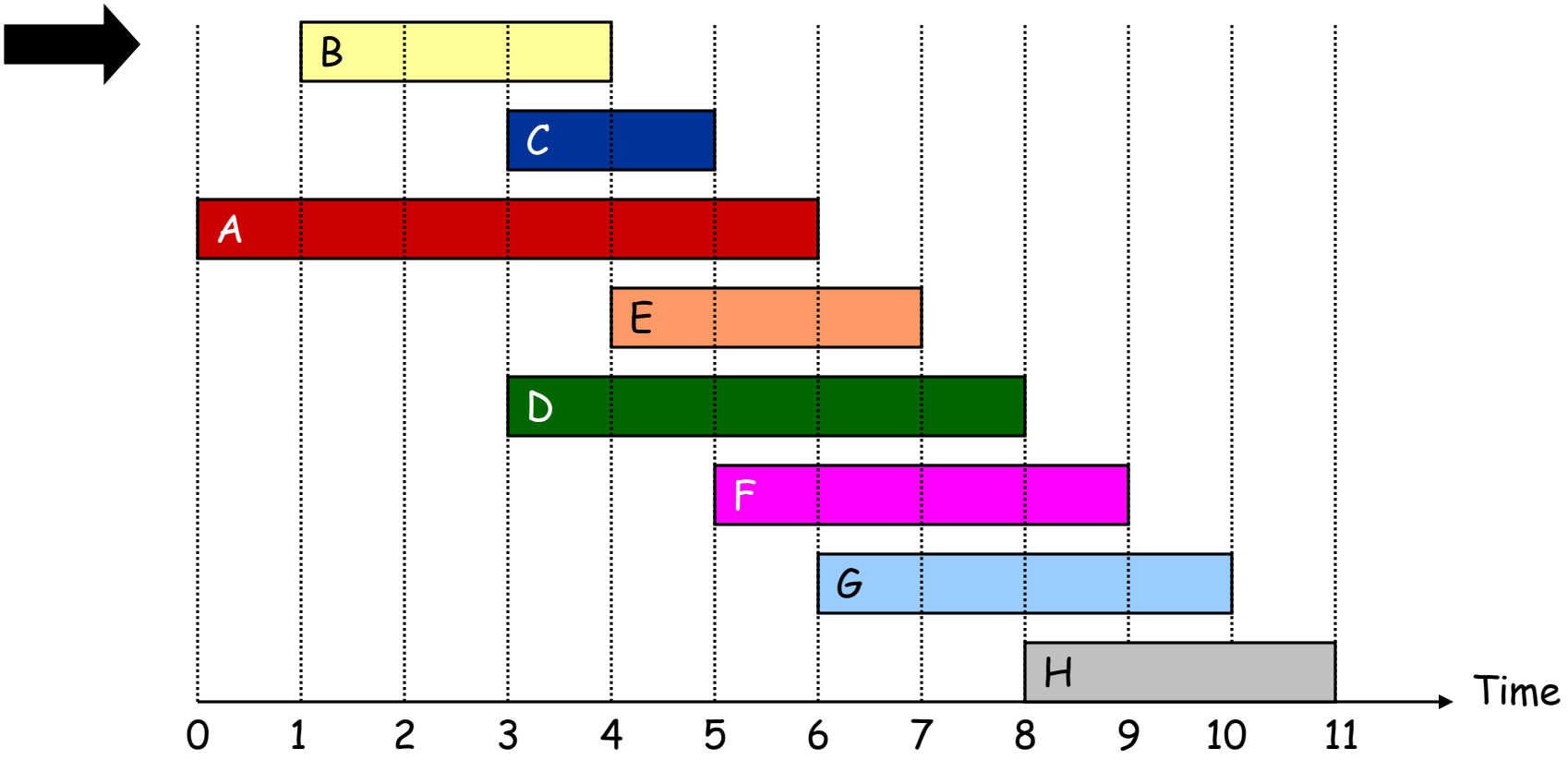
Interval Scheduling Demo

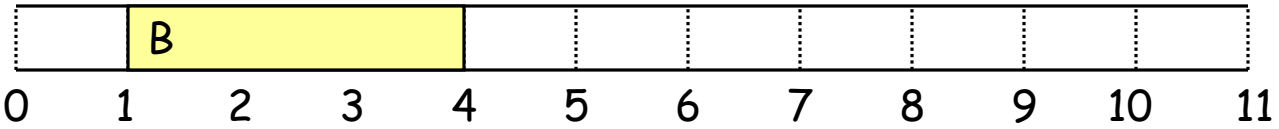
Interval Scheduling



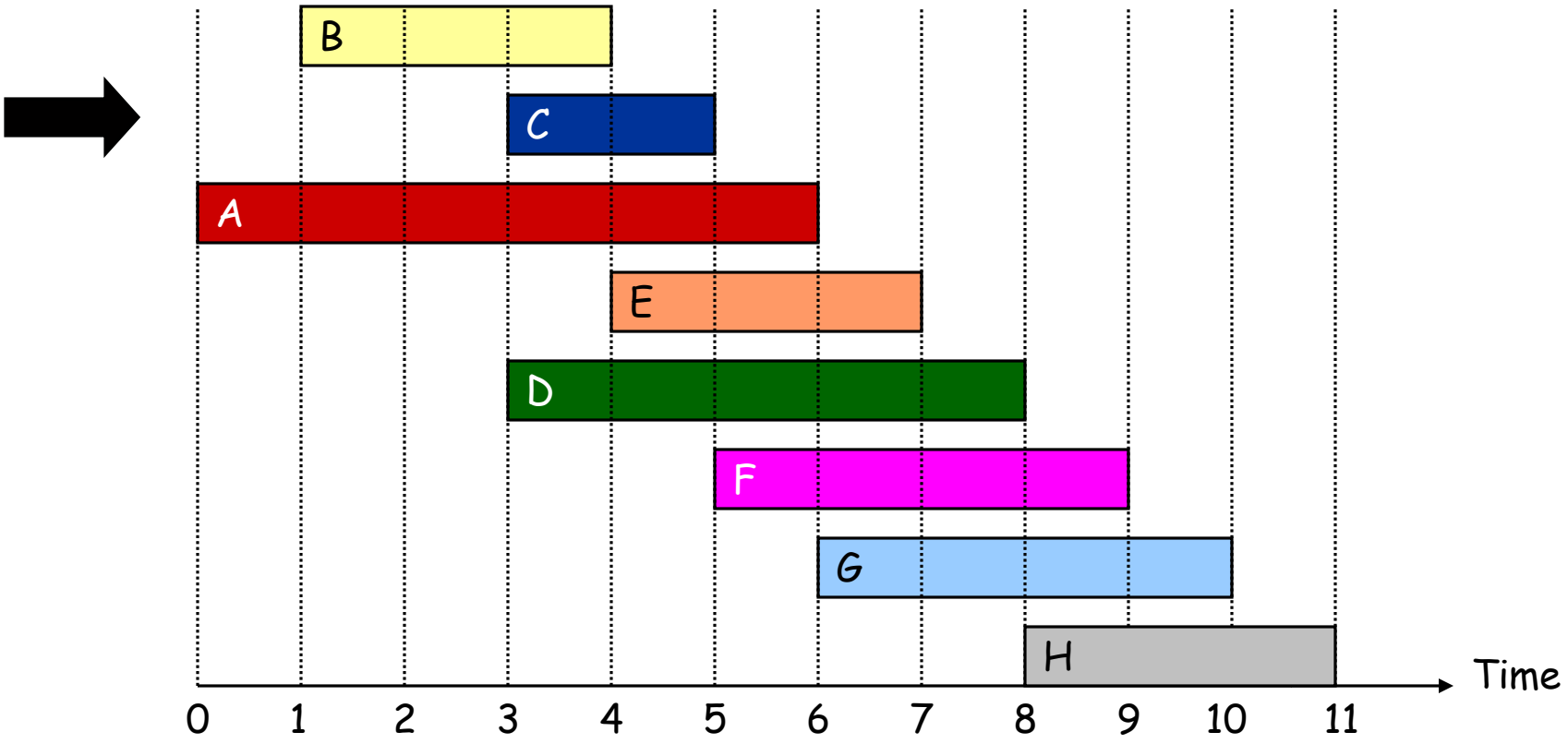


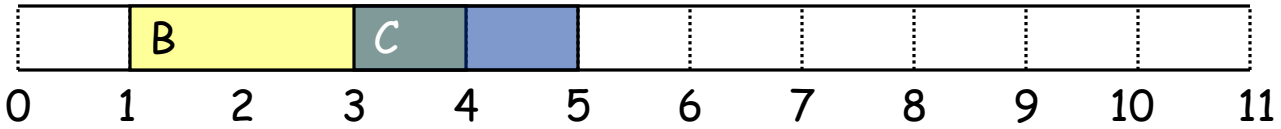
Interval Scheduling



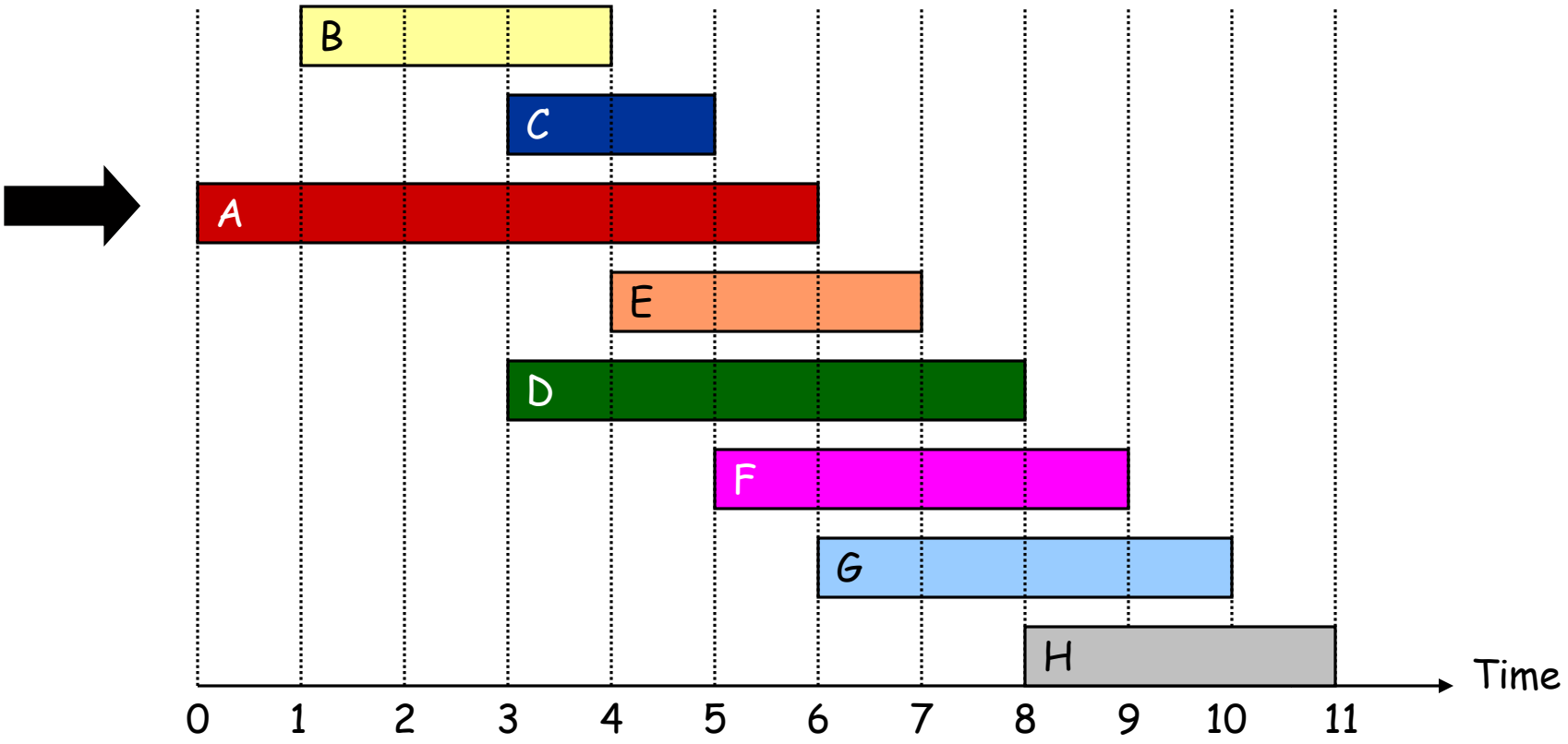


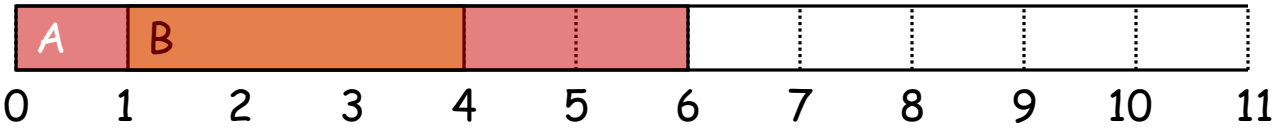
Interval Scheduling



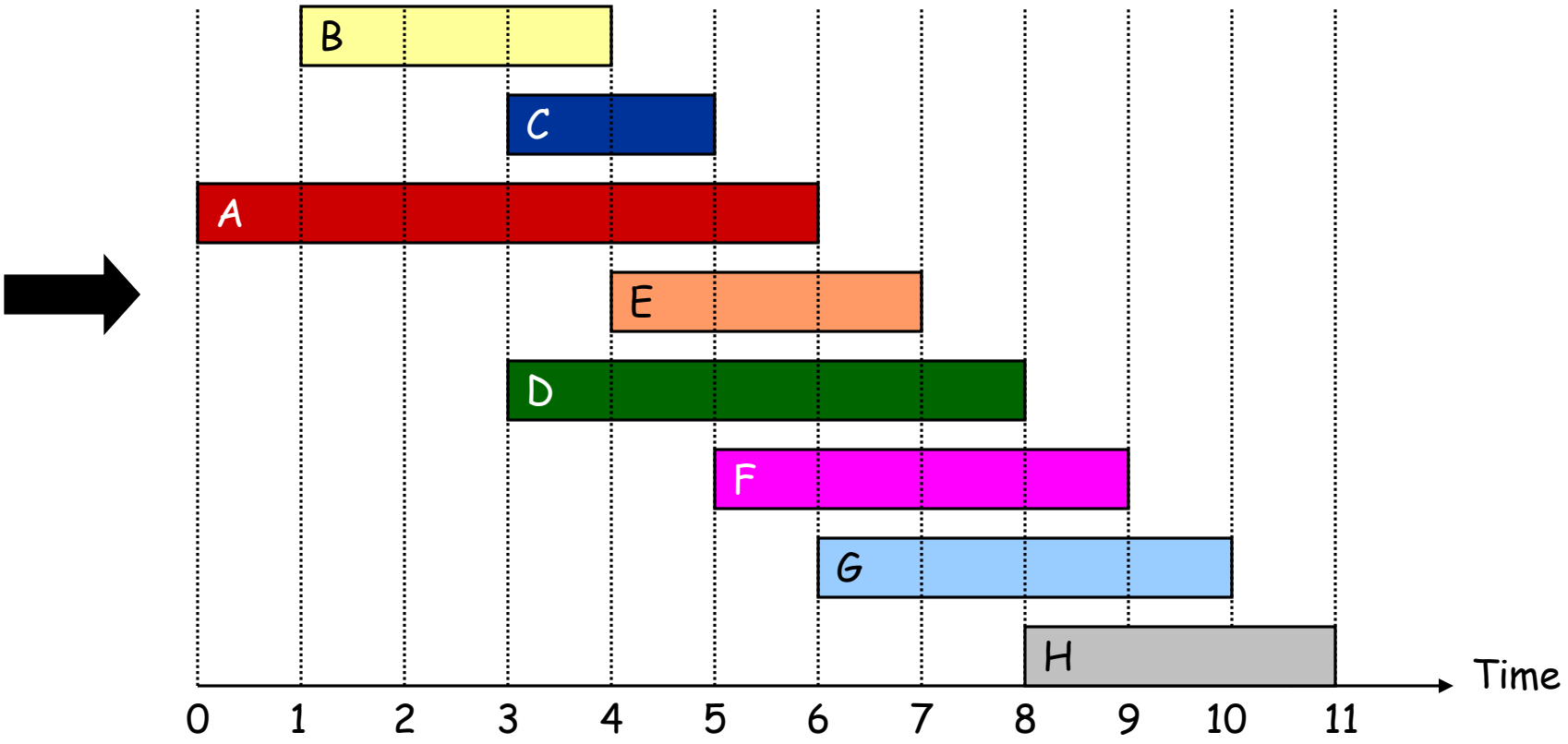


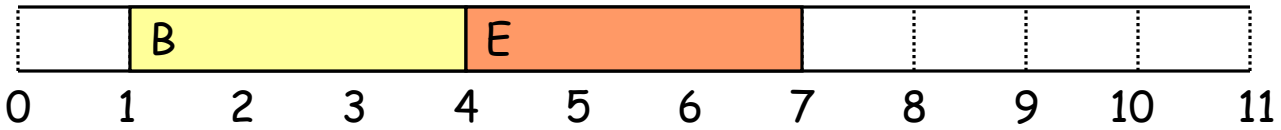
Interval Scheduling



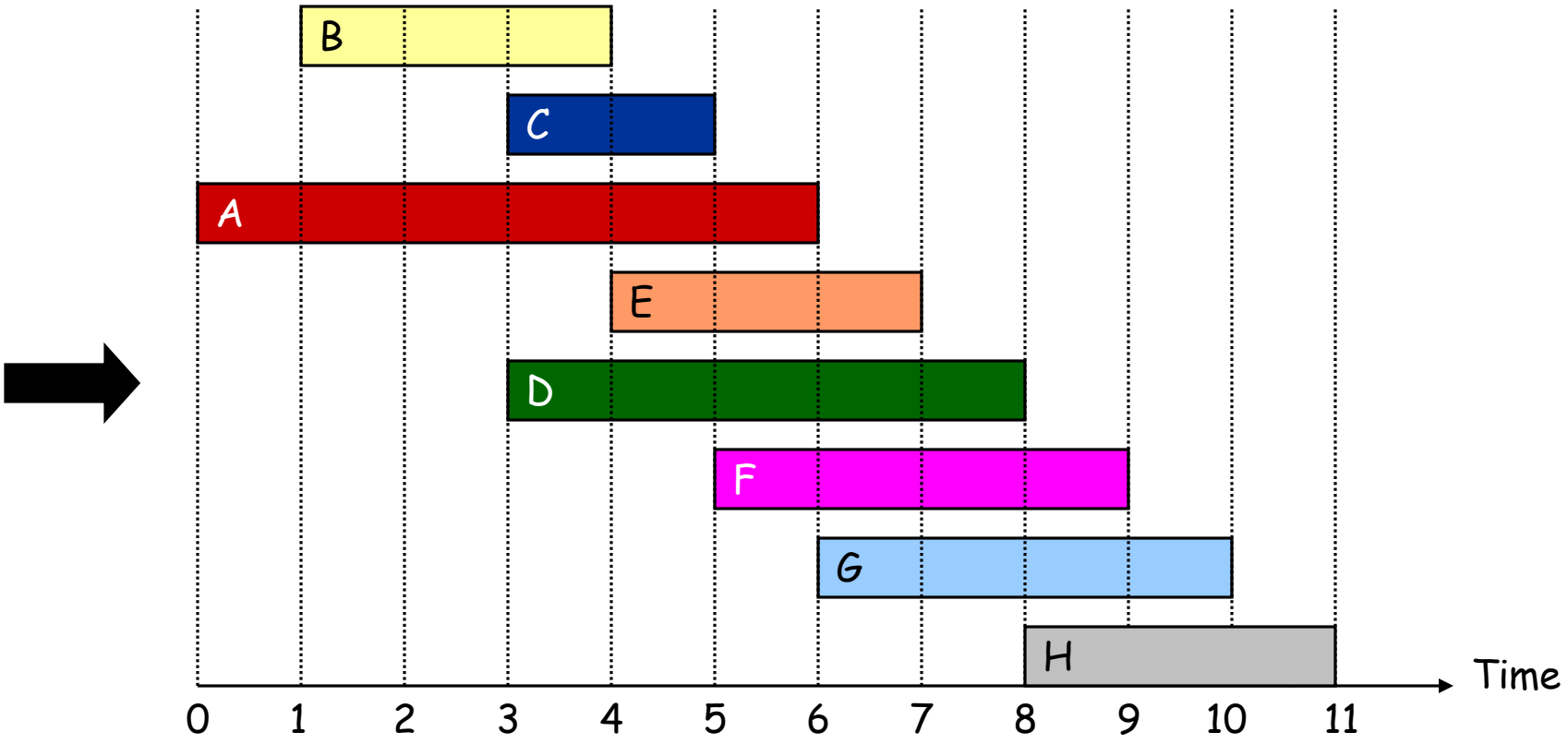


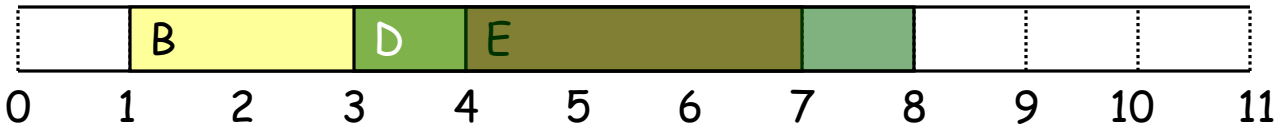
Interval Scheduling



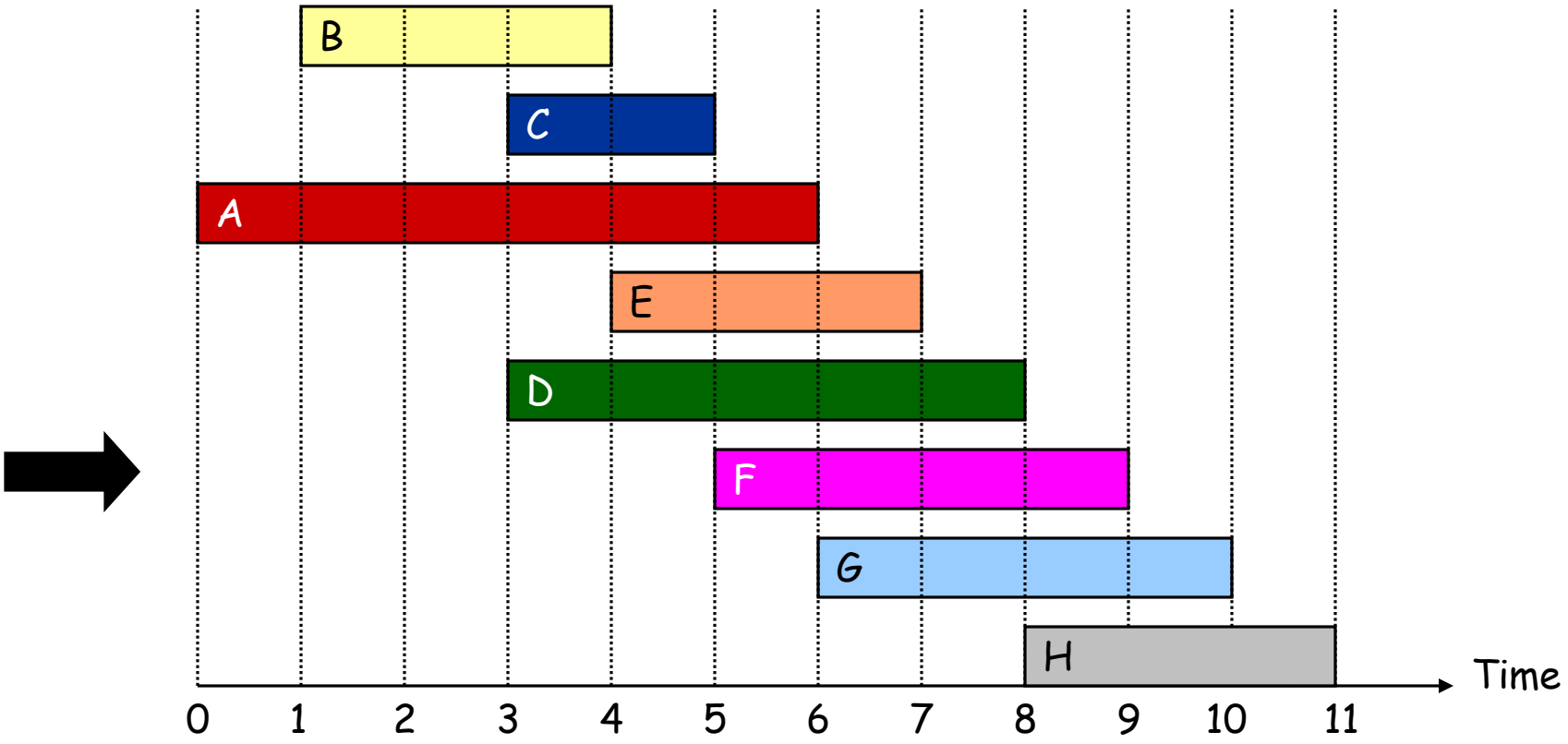


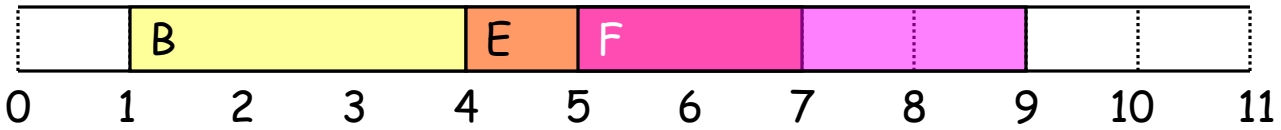
Interval Scheduling



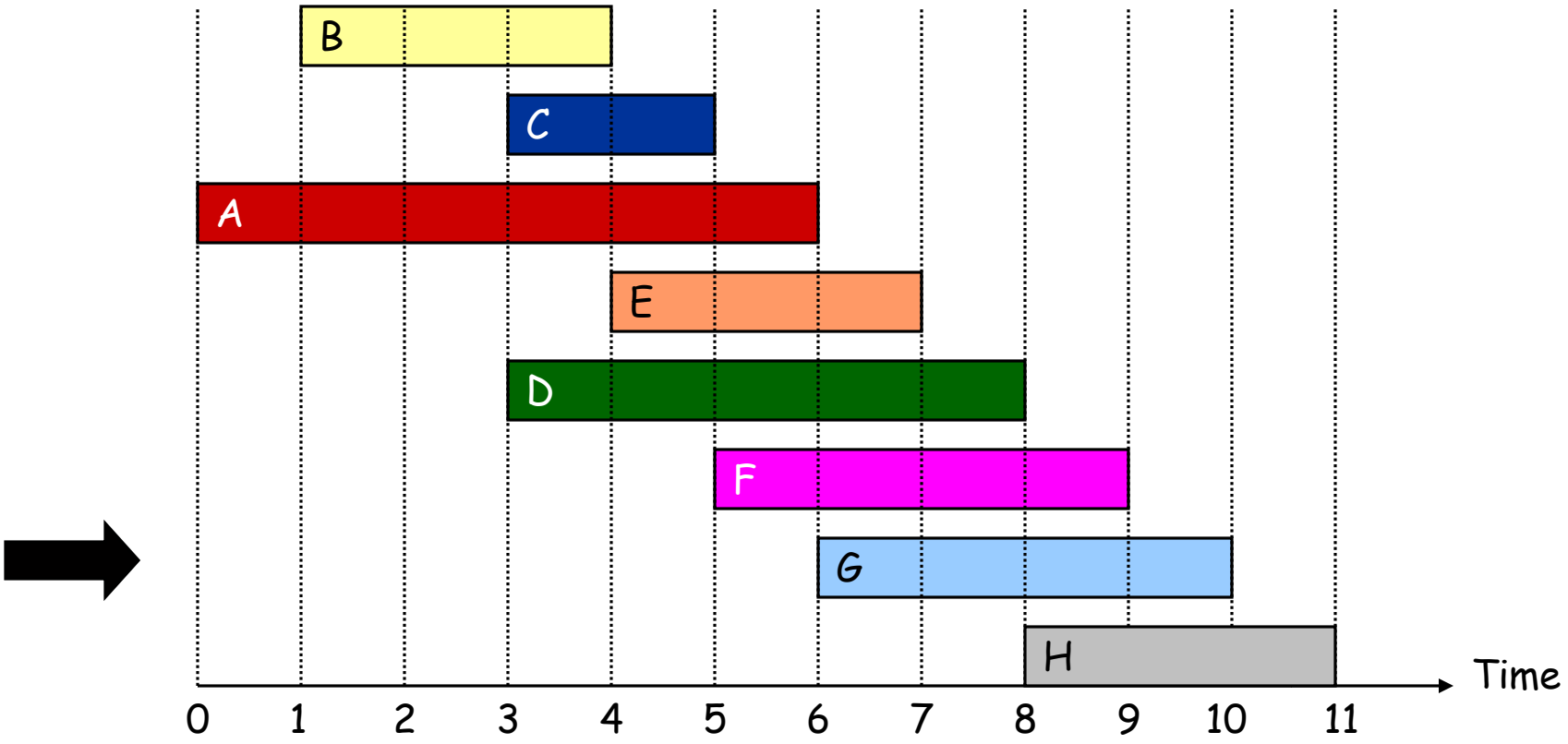


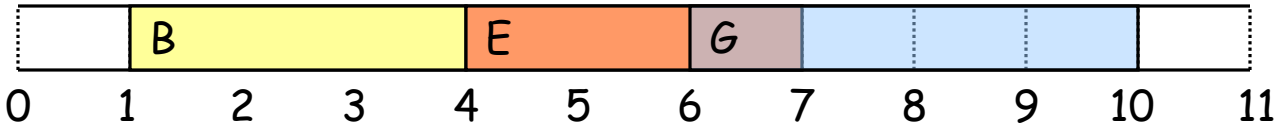
Interval Scheduling



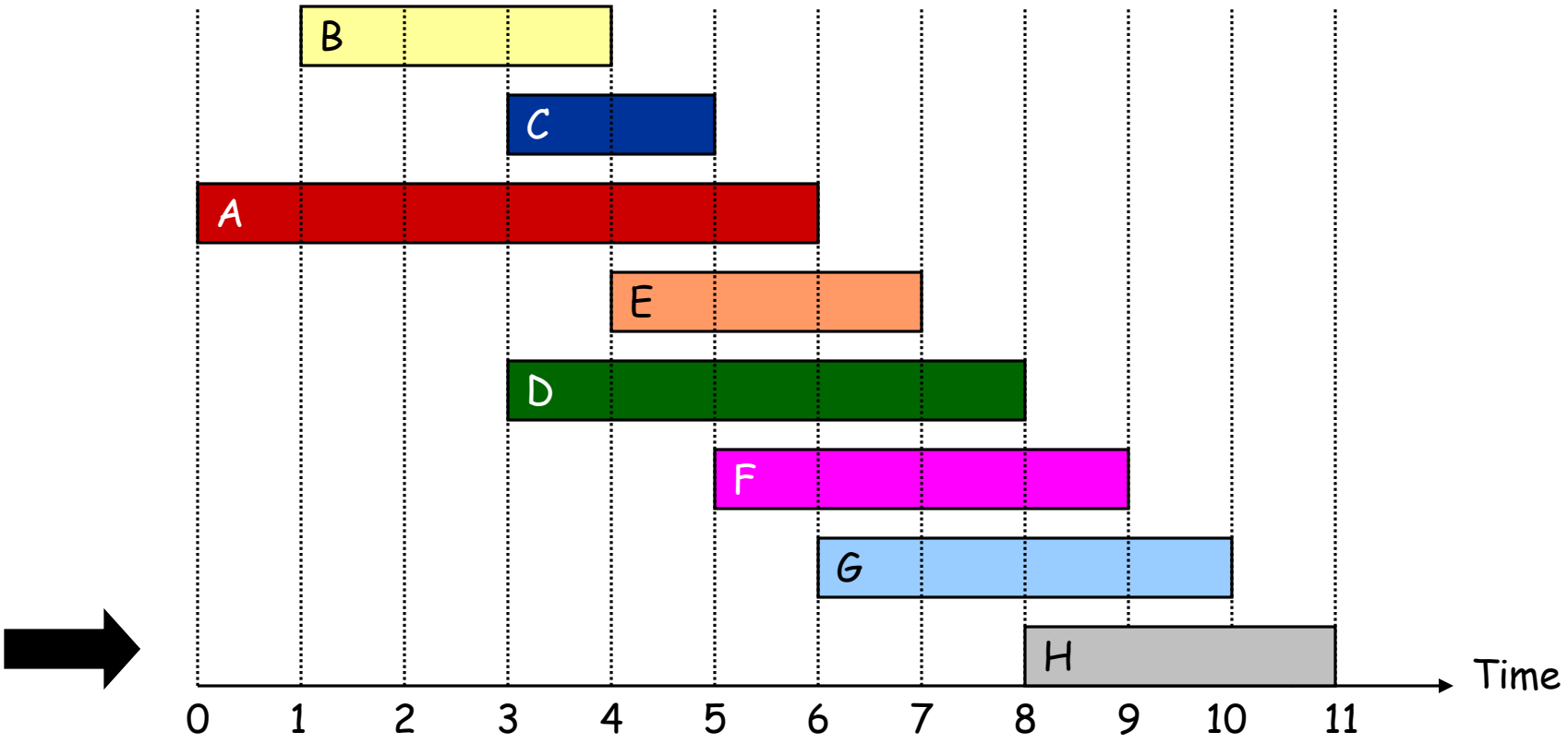


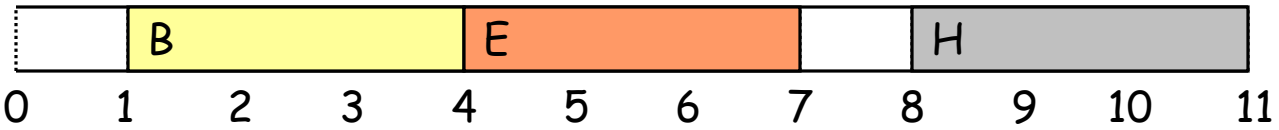
Interval Scheduling





Interval Scheduling





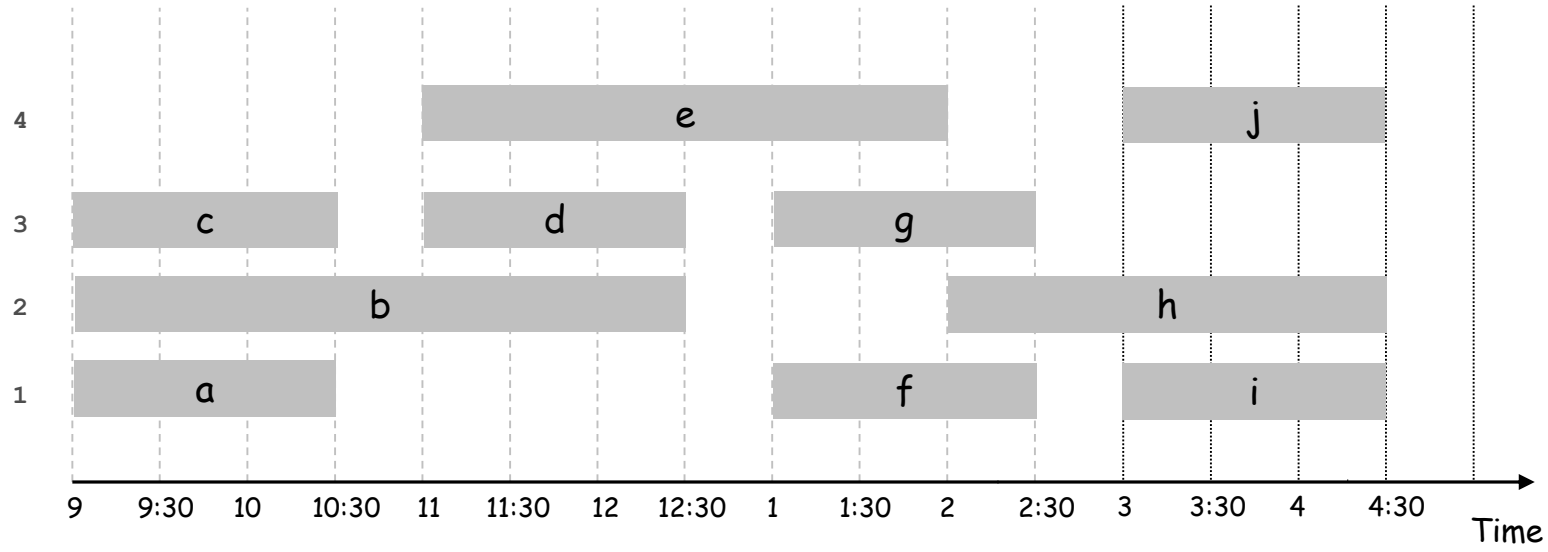
4.1.2 Interval Partitioning

Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses 4 classrooms to schedule 10 lectures.

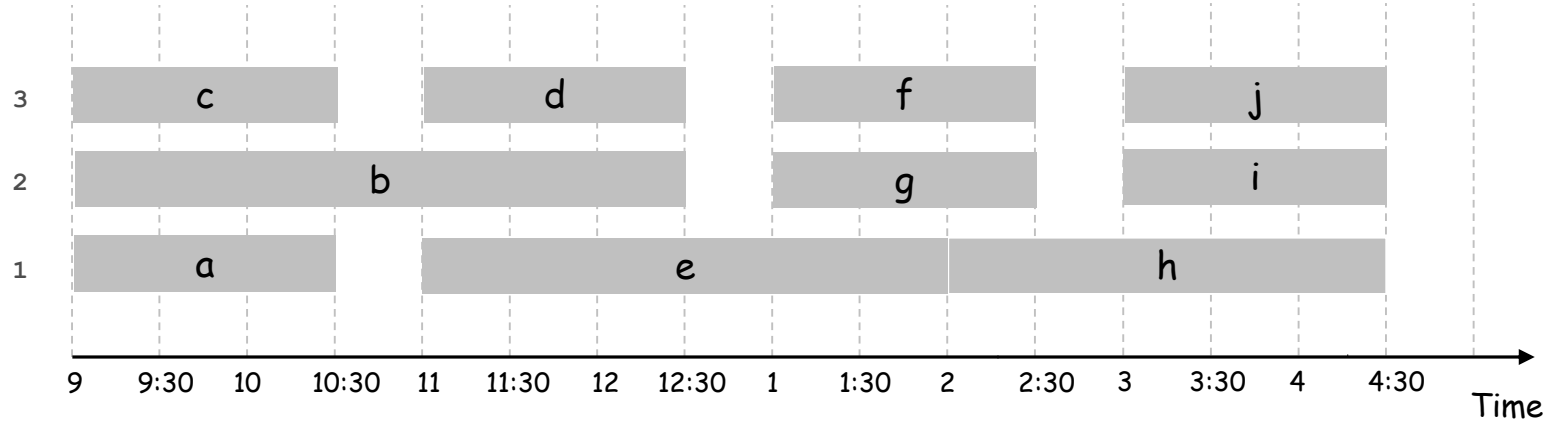


Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses only 3.



Interval Partitioning: Lower Bound on Optimal Solution

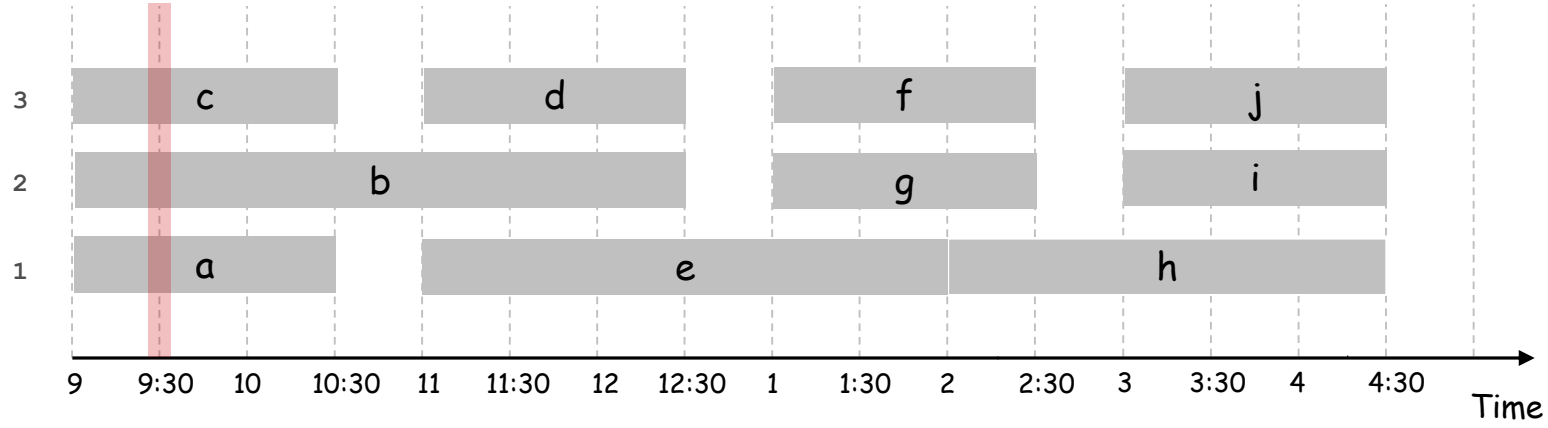
Def. The depth of a set of open intervals is the maximum number that contain any given time.

Key observation. Number of classrooms needed $\geq$ depth.

Ex: Depth of schedule below = 3 $\Rightarrow$ schedule below is optimal.

a, b, c all contain 9:30

Q. Does there always exist a schedule equal to depth of intervals?



Interval Partitioning: Greedy Algorithm

Greedy algorithm. Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that s_1 ≤ s_2 ≤ ... ≤ s_n.
d ← 0   ← number of allocated classrooms

for j = 1 to n {
   if (lecture j is compatible with some classroom k)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

Implementation. $O(n \log n)$.

- For each classroom k, maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.

Interval Partitioning: Greedy Analysis

Observation. Greedy algorithm never schedules two incompatible lectures in the same classroom. (so it produces a compatible assignment)

Theorem. Greedy algorithm is optimal.

Pf.

- Let d = number of classrooms that the greedy algorithm allocates.
- Classroom d is opened because we needed to schedule a job, say j, that is incompatible with all d-1 other classrooms.
- These d jobs each end after s_j.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j.
- Thus, we have d lectures overlapping at time $s_j + \varepsilon$.
- Key observation $\Rightarrow$ all schedules use $\geq$ d classrooms.

- In summary: The greedy algorithm above schedules every interval on a resource, using a number of resources (e.g. classroom) equal to the depth of the set of intervals. This is the optimal number of resources needed.

4.2 Scheduling to Minimize Lateness

Scheduling to Minimizing Lateness

Minimizing lateness problem.

- Single resource processes one job at a time.
- Job j requires t_j units of processing time and is due at time d_j.
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
- Lateness: $\ell_j = \max \{ 0, f_j - d_j \}$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

Ex:

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

Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

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

- [Earliest deadline first] Consider jobs in ascending order of deadline d_j.

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

Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

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

	1	2
t_j	1	10
d_j	100	10

counterexample

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

	1	2
t_j	1	10
d_j	2	10

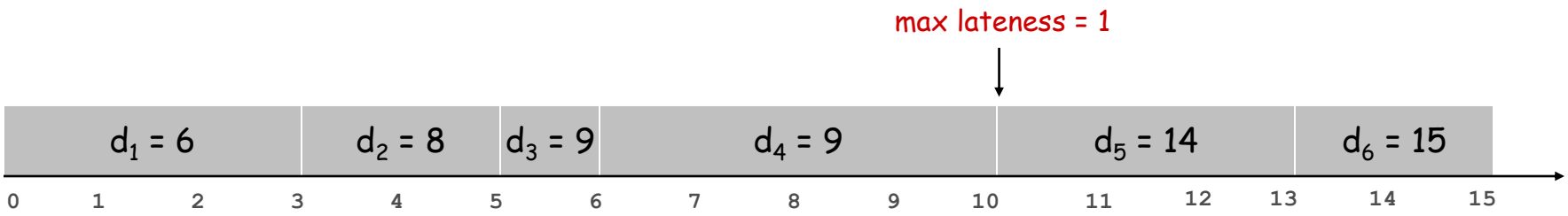
counterexample

Minimizing Lateness: Greedy Algorithm

Greedy algorithm. Earliest deadline first.

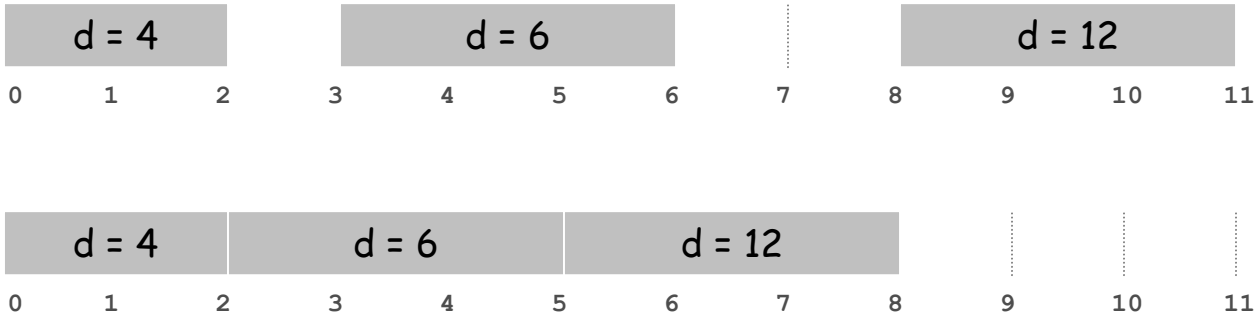
```
Sort n jobs by deadline so that d_1 ≤ d_2 ≤ … ≤ d_n

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + t_j]
   s_j ← t, f_j ← t + t_j
   t ← t + t_j
output intervals [s_j, f_j]
```



Minimizing Lateness: No Idle Time

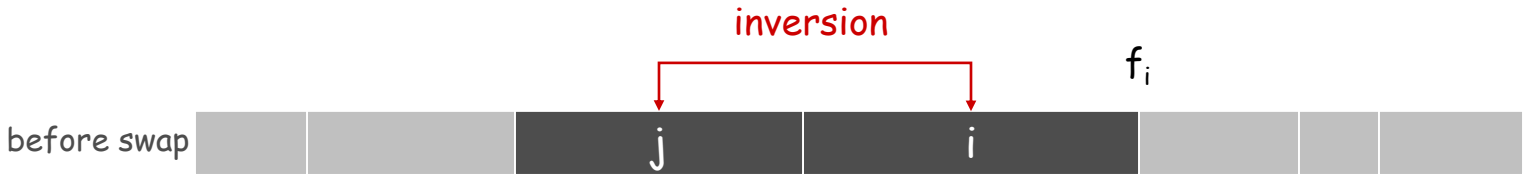
Observation. There exists an optimal schedule with no idle time. (why?)



Observation. The greedy schedule has no idle time. (why?)

Minimizing Lateness: Inversions

Def. Given a schedule S, an inversion is a pair of jobs i and j such that: i < j but j scheduled before i.



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

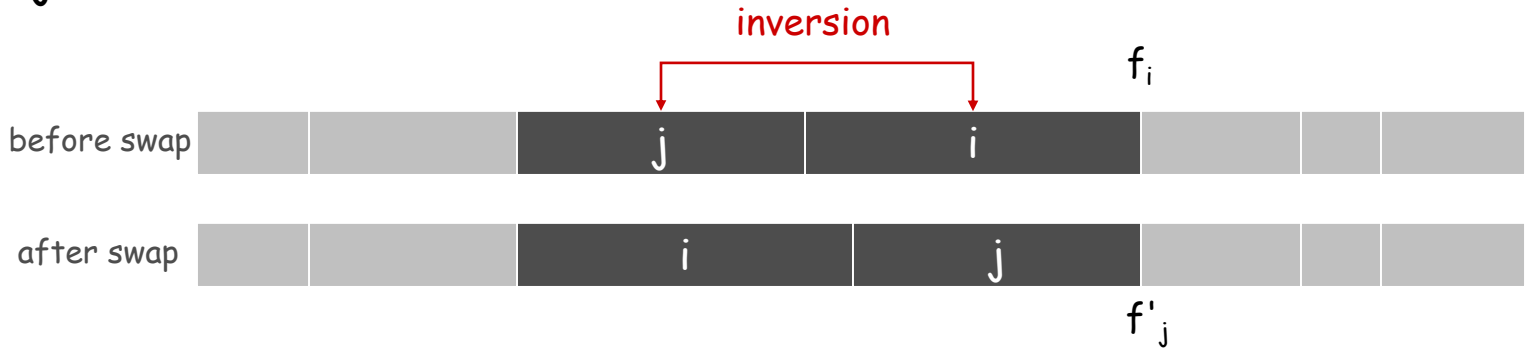
Observation. Greedy schedule has no inversions. (trivial)

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

(proof: Consider an inversion in which a job a is scheduled before a job b, and $d_a > d_b$. If we advance in the scheduled order of jobs from a to b one at a time, there has to come a point at which the deadline we see decreases for the first time (these are consecutive jobs forming an inversion).

Minimizing Lateness: Inversions

Def. Given a schedule S, an inversion is a pair of jobs i and j such that: i < j but j scheduled before i.



Claim. Swapping two consecutive, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

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

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

$$
\begin{aligned}
\ell'_j &= f'_j - d_j && \text{(definition)} \\
&= f_i - d_j && (j \text{ finishes at time } f_i) \\
&\leq f_i - d_i && (i < j) \\
&\leq \ell_i && \text{(definition)}
\end{aligned}
$$

Minimizing Lateness: Analysis of Greedy Algorithm

Theorem. Greedy schedule S is optimal.

Pf. Define S* to be an optimal schedule that has the fewest number of inversions, and let's see what happens.

- Can assume S* has no idle time.
- If S* has no inversions, then S = S* (since greedy algorithm will produce this schedule).
- If S* has an inversion, let i-j be an adjacent inversion. (guaranteed adjacent by previous result)
  - swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions (by prior result)
  - this contradicts definition of S*

In summary: S* cannot have any inversions, and so S*=S, thus the greedy schedule is optimal.

Greedy Analysis Strategies

Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

Structural. Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

Other greedy algorithms. Kruskal, Prim, Dijkstra, Huffman, …

4.3 Optimal Caching

Caching

(*) Caching is in general the process of storing a small amount of data in a fast memory so as to reduce the amount of time spent interacting with a slow memory.

For caching to be as effective as possible, it should generally be the case that when you go to access a piece of data, it is already in the cache.

To achieve this, a cache maintenance algorithm determines what to keep in the cache and what to "evict" from the cache as new data is brought in.

Optimal Offline Caching

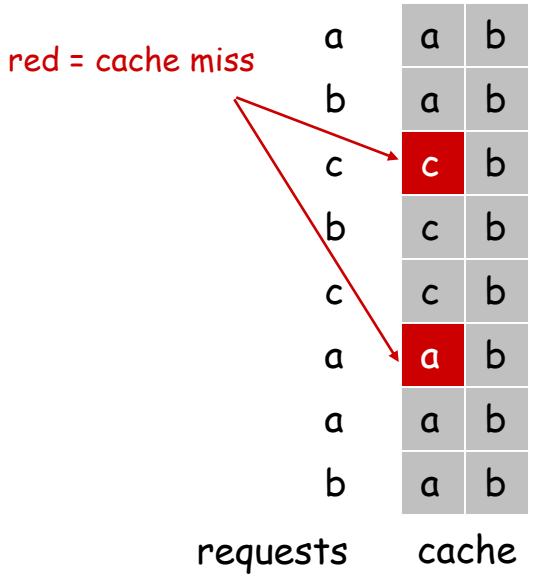
Caching.

- Cache with capacity to store k items.
- Sequence of m item requests $d_1, d_2, \ldots, d_m$.
- Cache hit: item already in cache when requested.
- Cache miss: item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.

Goal. Eviction schedule that minimizes number of cache misses.

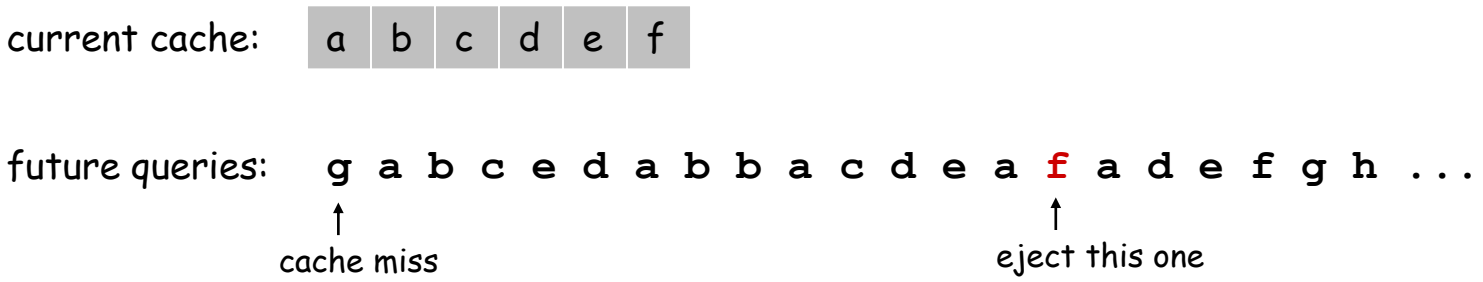
Ex: k = 2, initial cache = a b,
requests: a, b, c, b, c, a, a, b.

Optimal eviction schedule: 2 cache misses.



Optimal Offline Caching: Farthest-In-Future

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



Theorem. [Bellady, 1960s] FF is optimal eviction schedule.

Pf. Algorithm and theorem are intuitive; proof is subtle.

Reduced Eviction Schedules

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

Intuition. Can transform an unreduced schedule into a reduced one with no more cache misses.

a	a	b	c
a	a	x	c
c	a	d	c
d	a	d	b
a	a	c	b
b	a	x	b
c	a	c	b
a	a	b	c
a	a	b	c

an unreduced schedule

a	a	b	c
a	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
b	a	d	b
c	a	c	b
a	a	c	b
a	a	c	b

a reduced schedule

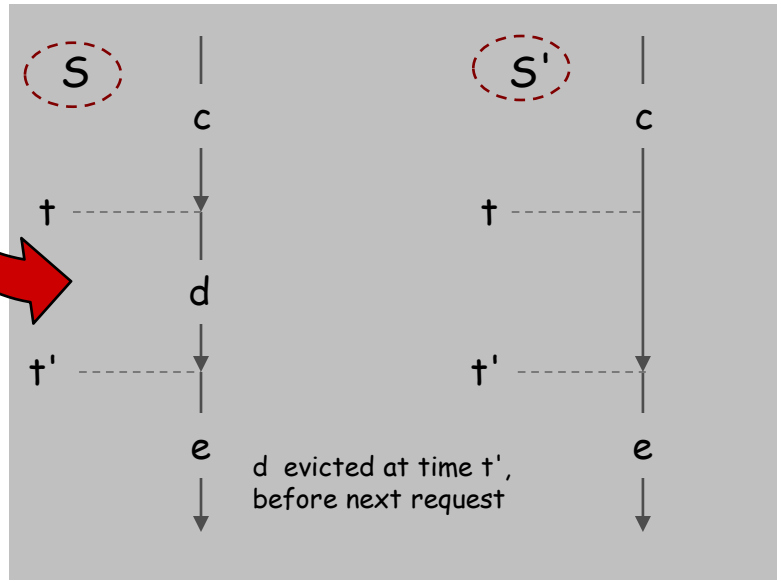
Reduced Eviction Schedules

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

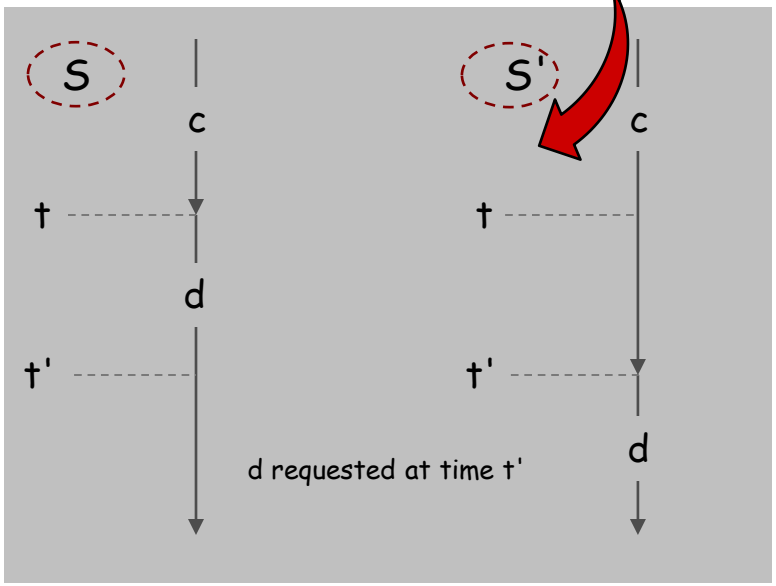
Pf. (by induction on number of unreduced items)

doesn't enter cache at requested time

- Suppose S brings d into the cache at time t, without a request.
- Let c be the item S evicts when it brings d into the cache.
- Case 1: d evicted at time t', before next request for d.
- Case 2: d requested at time t' before d is evicted. ▪



Case 1



Case 2

Farthest-In-Future: Analysis (Exchange Argument)

Theorem. FF is optimal eviction algorithm.

Pf. (by induction on number or requests j)

> Invariant: There exists an optimal reduced schedule S that makes the same eviction schedule as S_{FF} through the first j+1 requests.

Let S be reduced schedule that satisfies invariant through j requests.

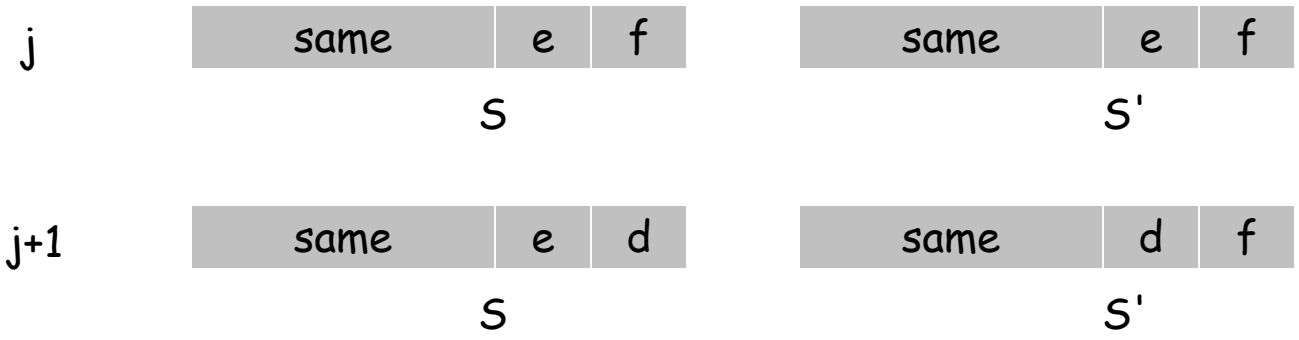
We produce S' that satisfies invariant after j+1 requests.

- Consider $(j+1)^{st}$ request $d = d_{j+1}$.
- Since S and S_{FF} have agreed up until now, they have the same cache contents before request j+1.
- Case 1: (d is already in the cache). S' = S satisfies invariant.
- Case 2: (d is not in the cache and S and S_{FF} evict the same element). S' = S satisfies invariant.

Farthest-In-Future: Analysis

Pf. (continued)

- Case 3: (d is not in the cache; S_{FF} evicts e; S evicts f ≠ e).
  - begin construction of S' from S by evicting e instead of f

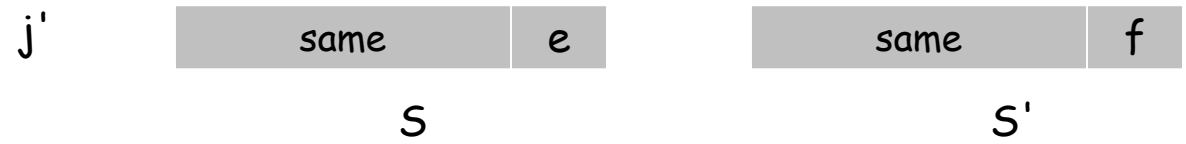


  - now S' agrees with S_{FF} on first j+1 requests; we show that having element f in cache is no worse than having element e

Farthest-In-Future: Analysis

Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.

must involve e or f (or both)



- Case 3a: g = e. Can't happen with Farthest-In-Future since there must be a request for f before e.

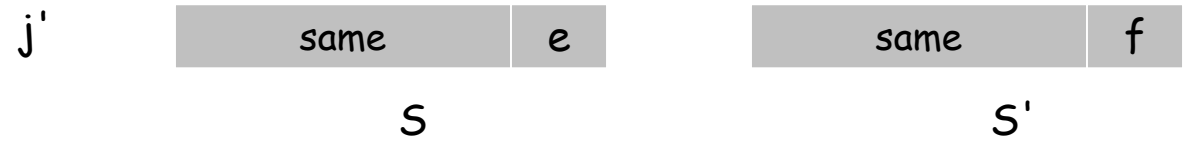
- Case 3b: g = f. Element f can't be in cache of S, so let e' be the element that S evicts.
  - if e' = e, S' accesses f from cache; now S and S' have same cache
  - if e' ≠ e, S' evicts e' and brings e into the cache; now S and S' have the same cache

Note: S' is no longer reduced, but can be transformed into a reduced schedule that agrees with S_{FF} through step j+1

Farthest-In-Future: Analysis

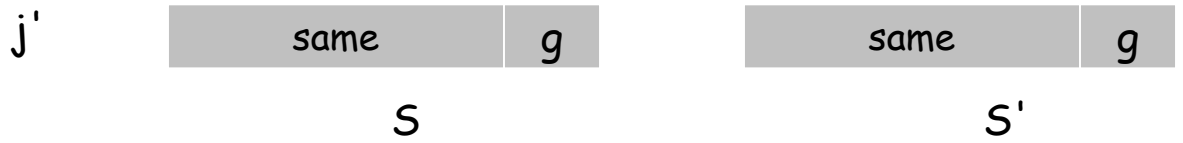
Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.

must involve e or f (or both)



otherwise S' would take the same action

- Case 3c: $g \neq e$, f. S must evict e.
  Make S' evict f; now S and S' have the same cache. ▪



Caching Perspective

Online vs. offline algorithms.

- Offline: full sequence of requests is known a priori.
- Online (reality): requests are not known in advance.
- Caching is among most fundamental online problems in CS.

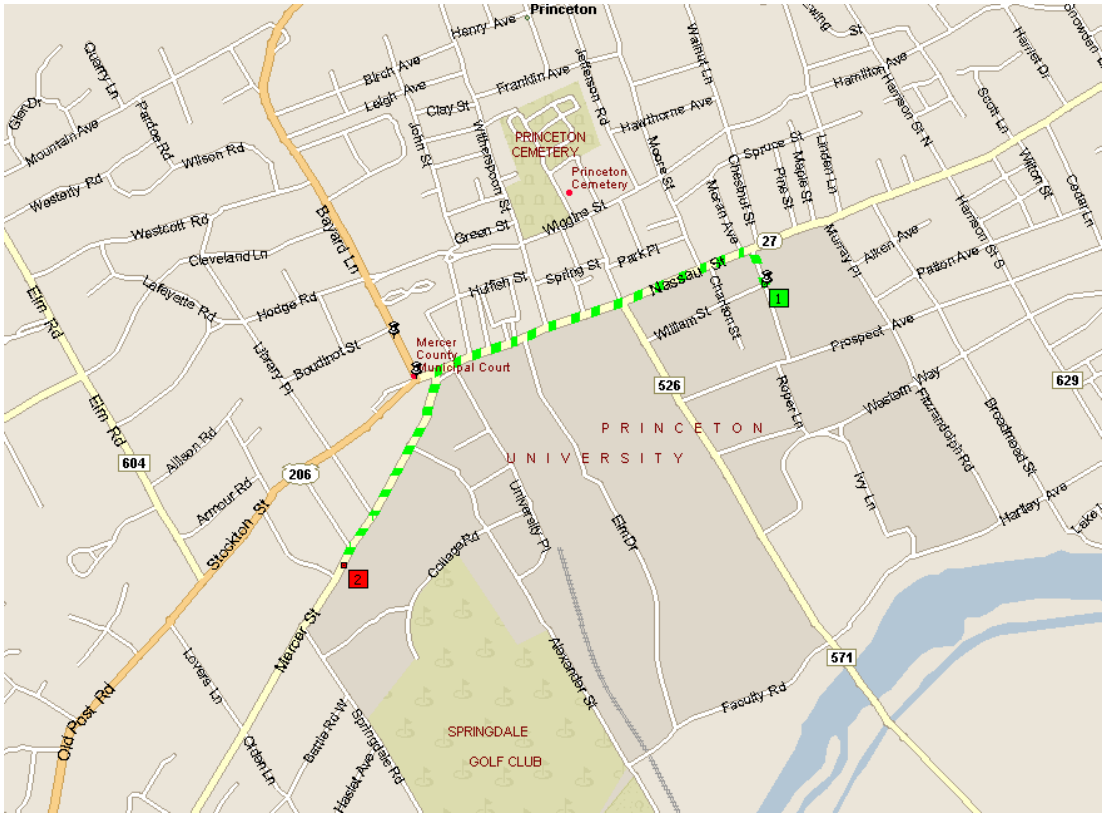
LIFO. Evict page brought in most recently.

LRU. Evict page whose most recent access was earliest.

Theorem. FF is optimal offline eviction algorithm.

- Provides basis for understanding and analyzing online algorithms.
- LRU is k-competitive. [Section 13.8]
- LIFO is arbitrarily bad.

4.4 Shortest Paths in a Graph



shortest path from Princeton CS department to Einstein's house

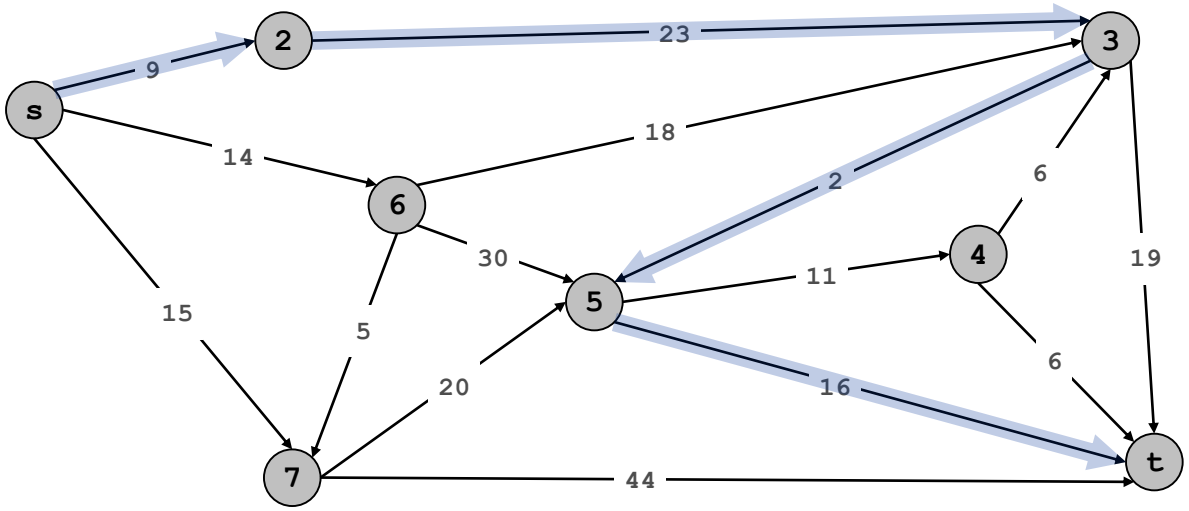
Shortest Path Problem

Shortest path network.

- Directed graph $G = (V, E)$.
- Source s, destination t.
- Length ℓ_e = length of edge e.

Shortest path problem: find shortest directed path from s to t.

cost of path = sum of edge costs in path



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 50.

Dijkstra's Algorithm

Dijkstra's algorithm.

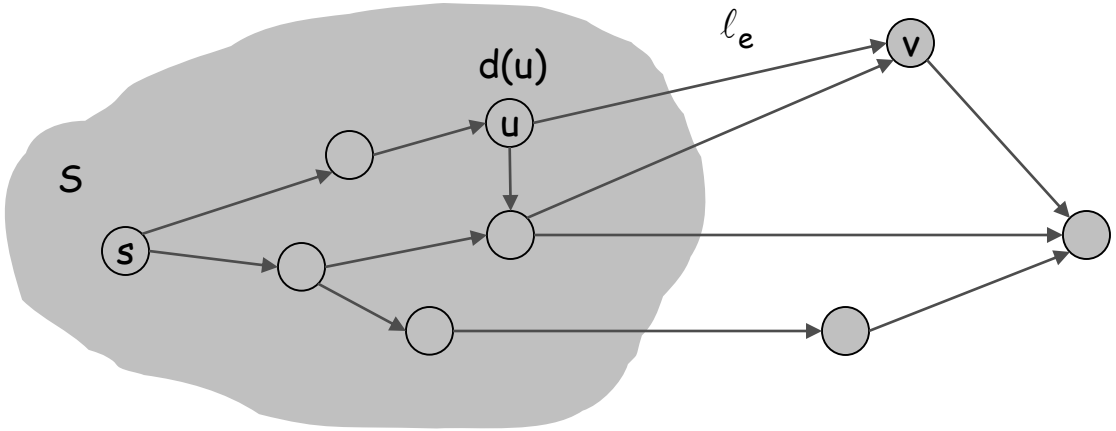
- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v) : u \in S} d(u) + \ell_e ,$$

add v to S, and set d(v) = π(v).

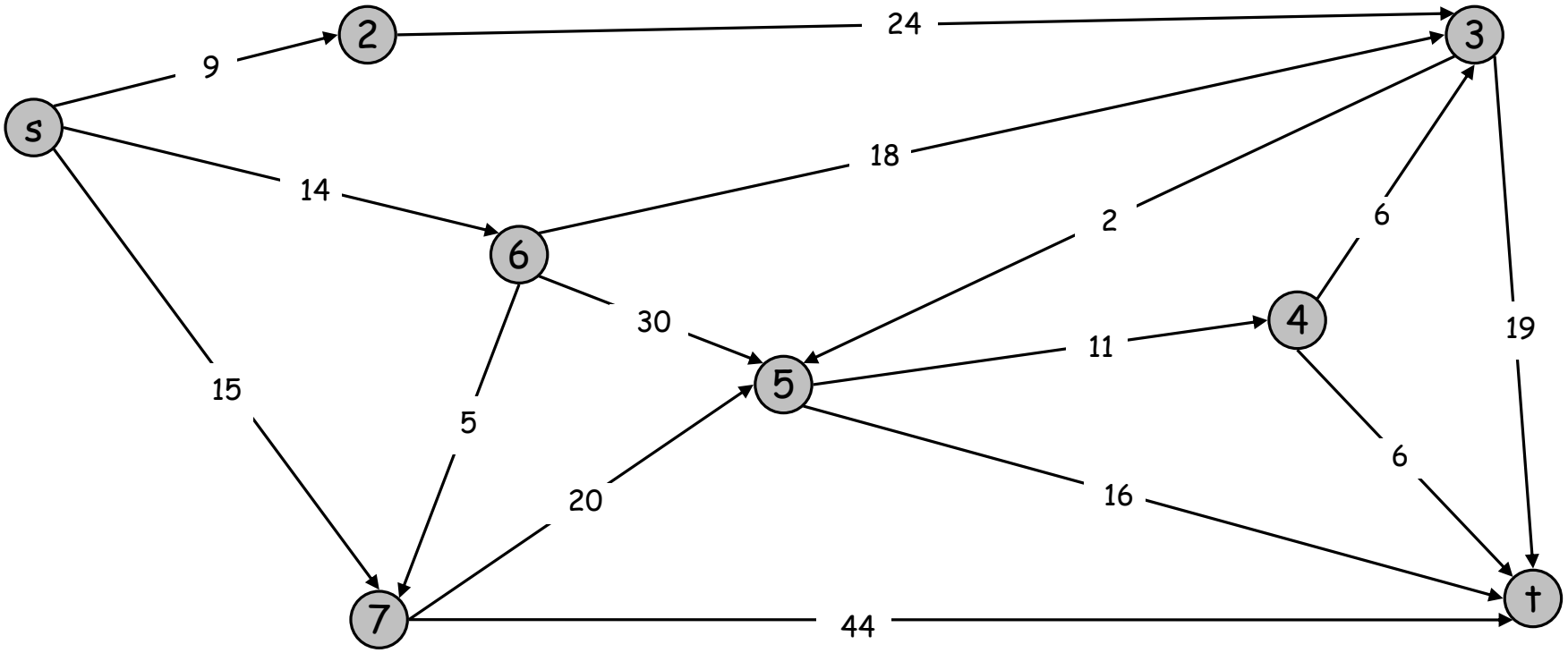
shortest path to some u in explored part, followed by a single edge (u, v)

- (Note: π(v) represents a "temporary" distance label the algorithm runs)



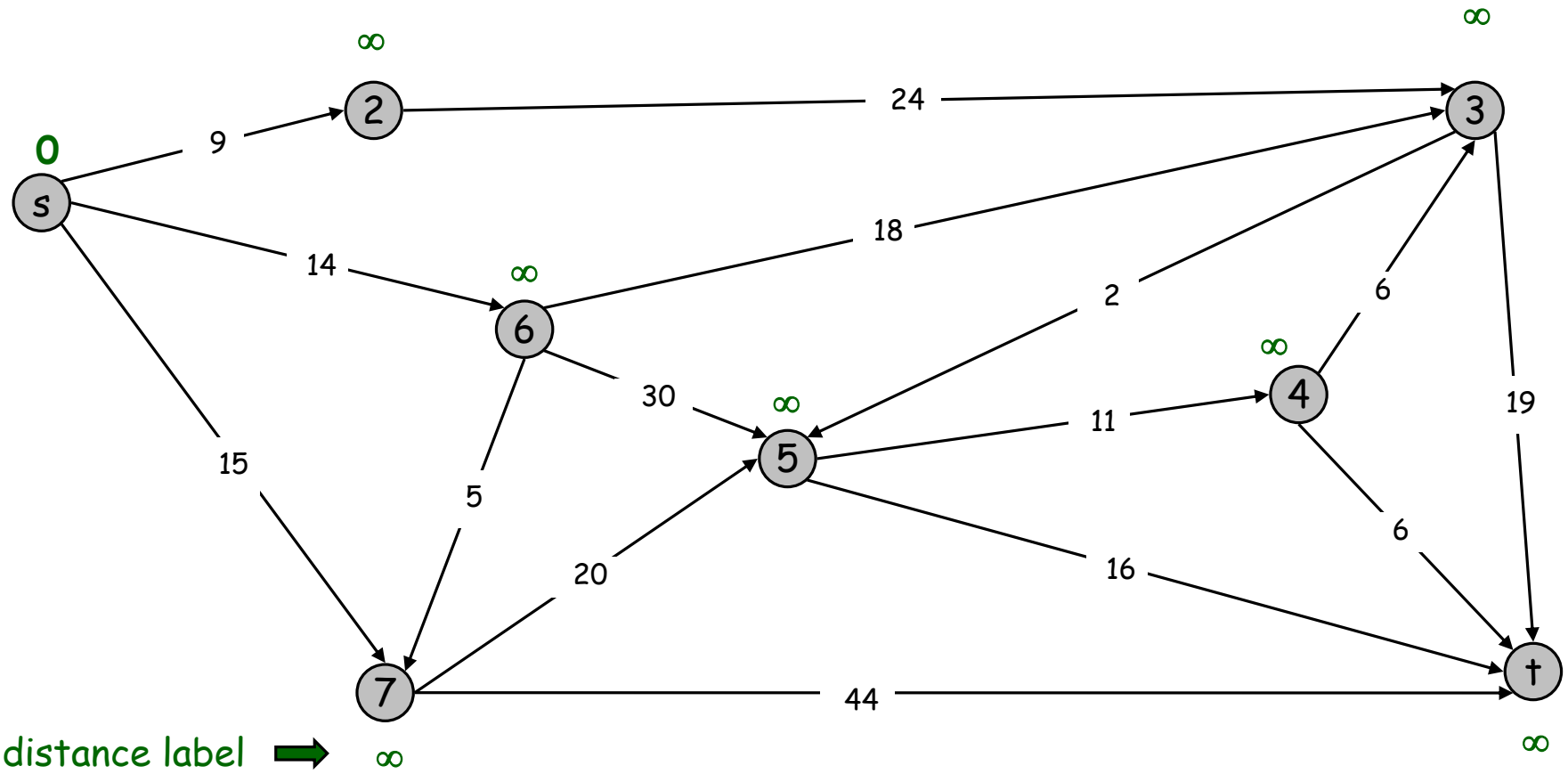
Dijkstra's Shortest Path Algorithm

Find shortest path from s to t.



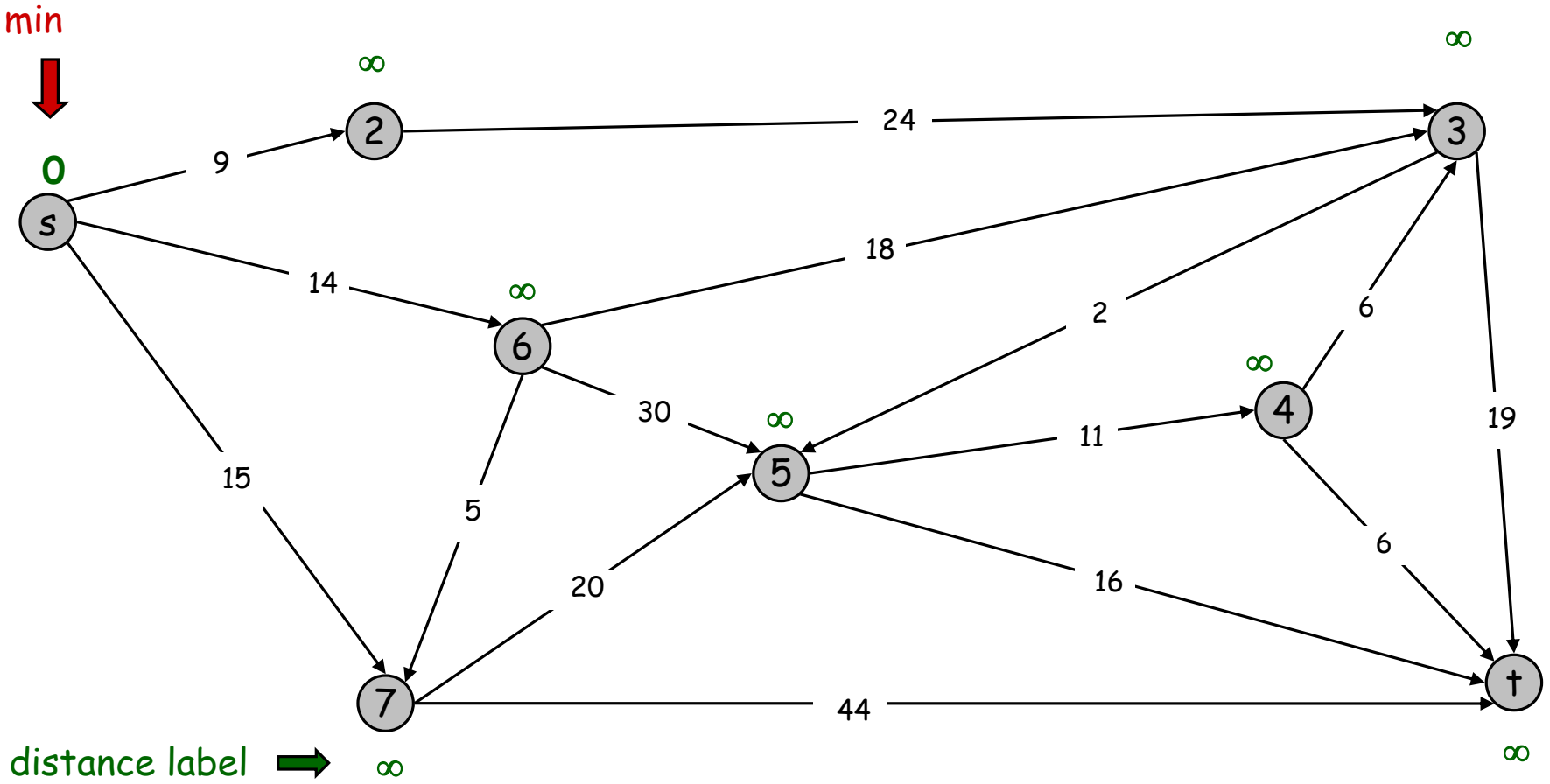
Dijkstra's Shortest Path Algorithm

S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }



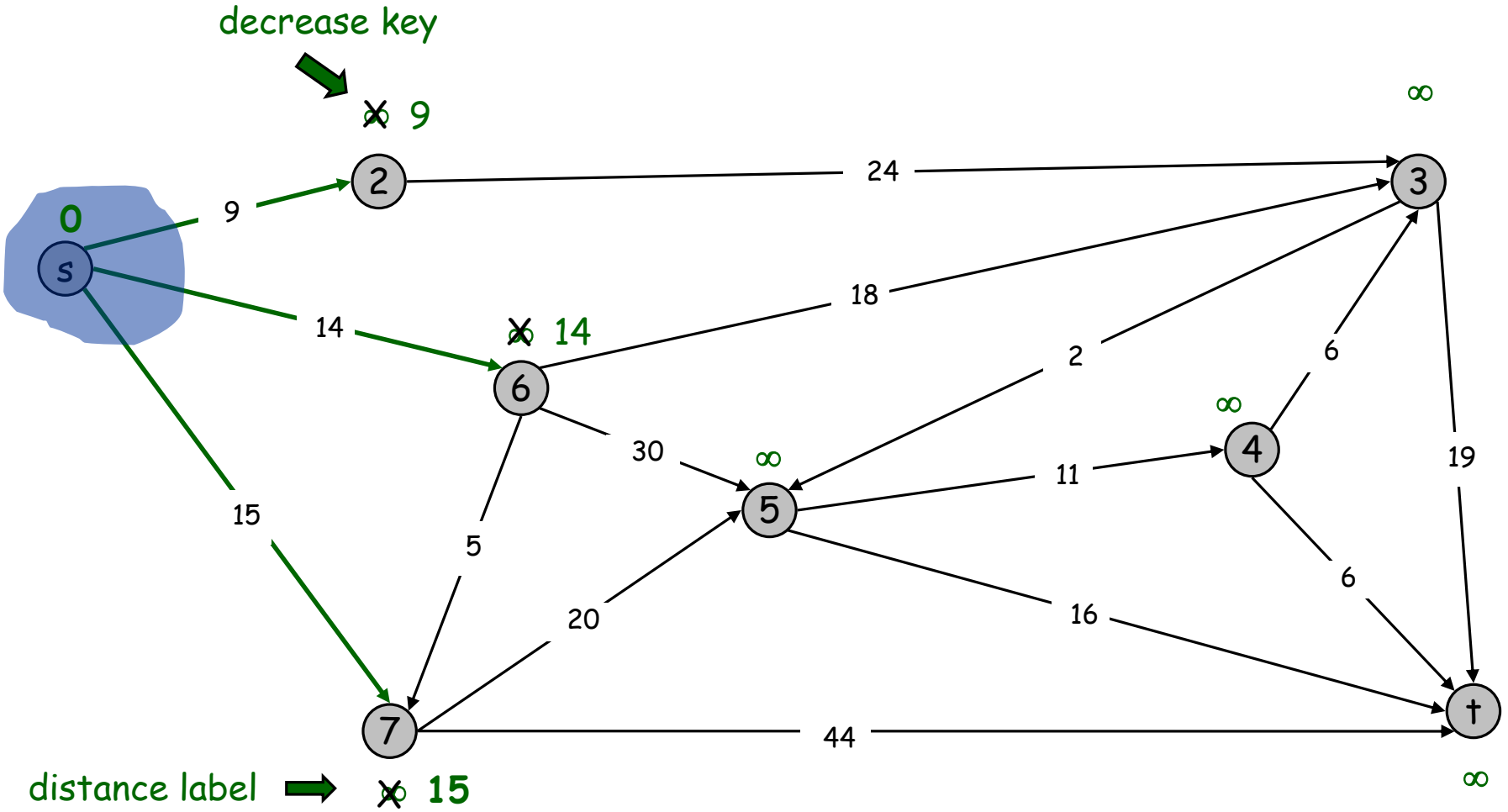
Dijkstra's Shortest Path Algorithm

S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }



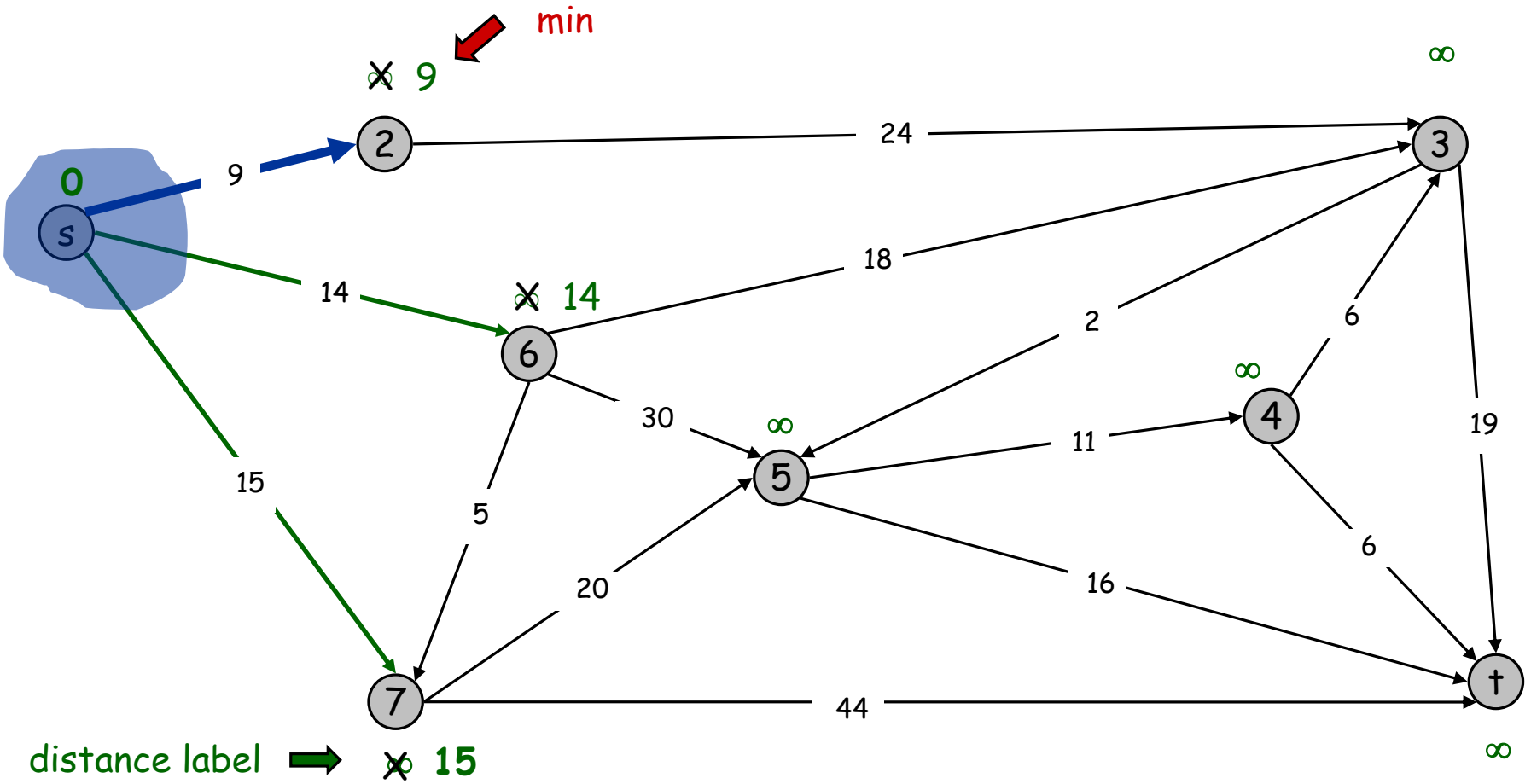
Dijkstra's Shortest Path Algorithm

S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }



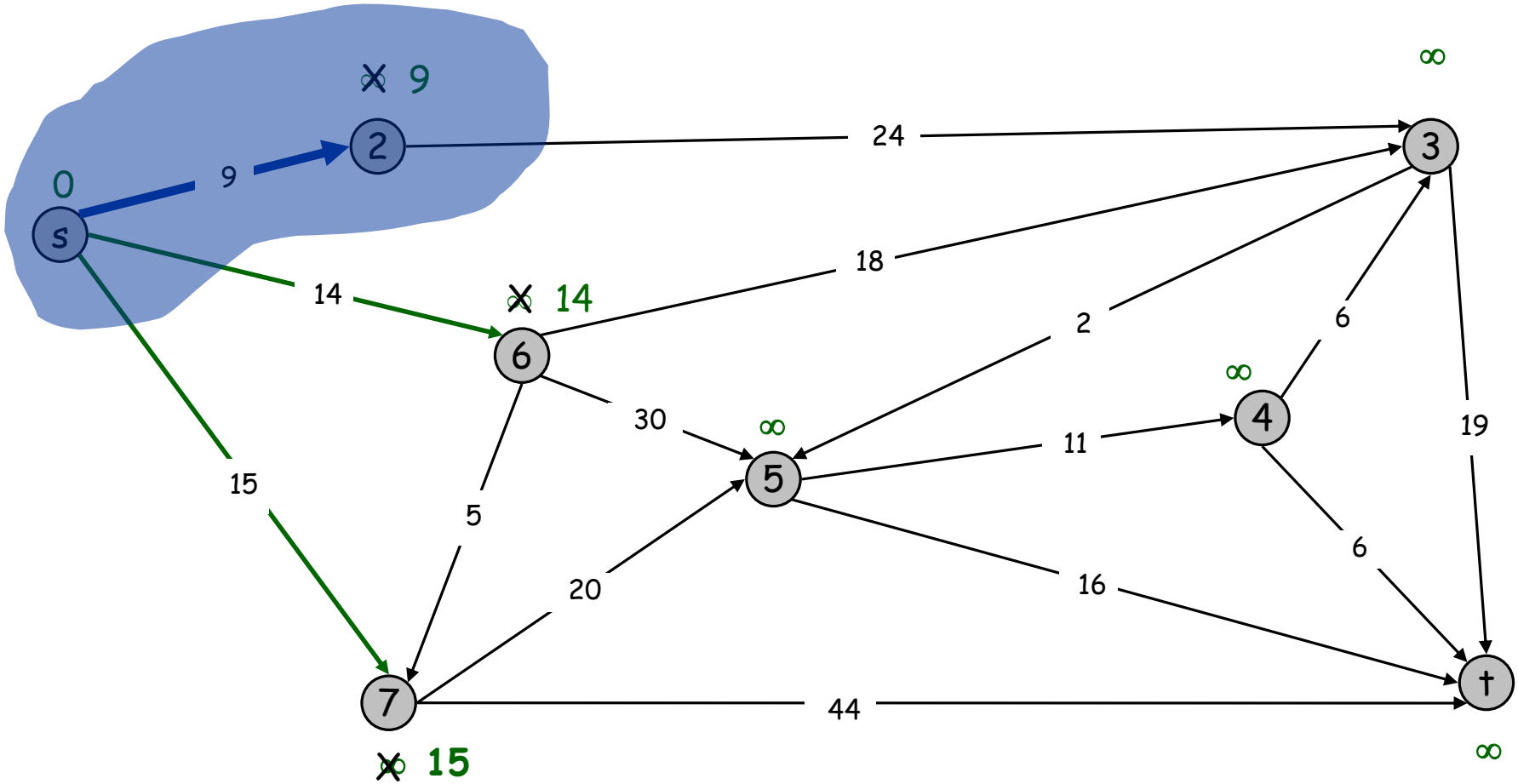
Dijkstra's Shortest Path Algorithm

S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }



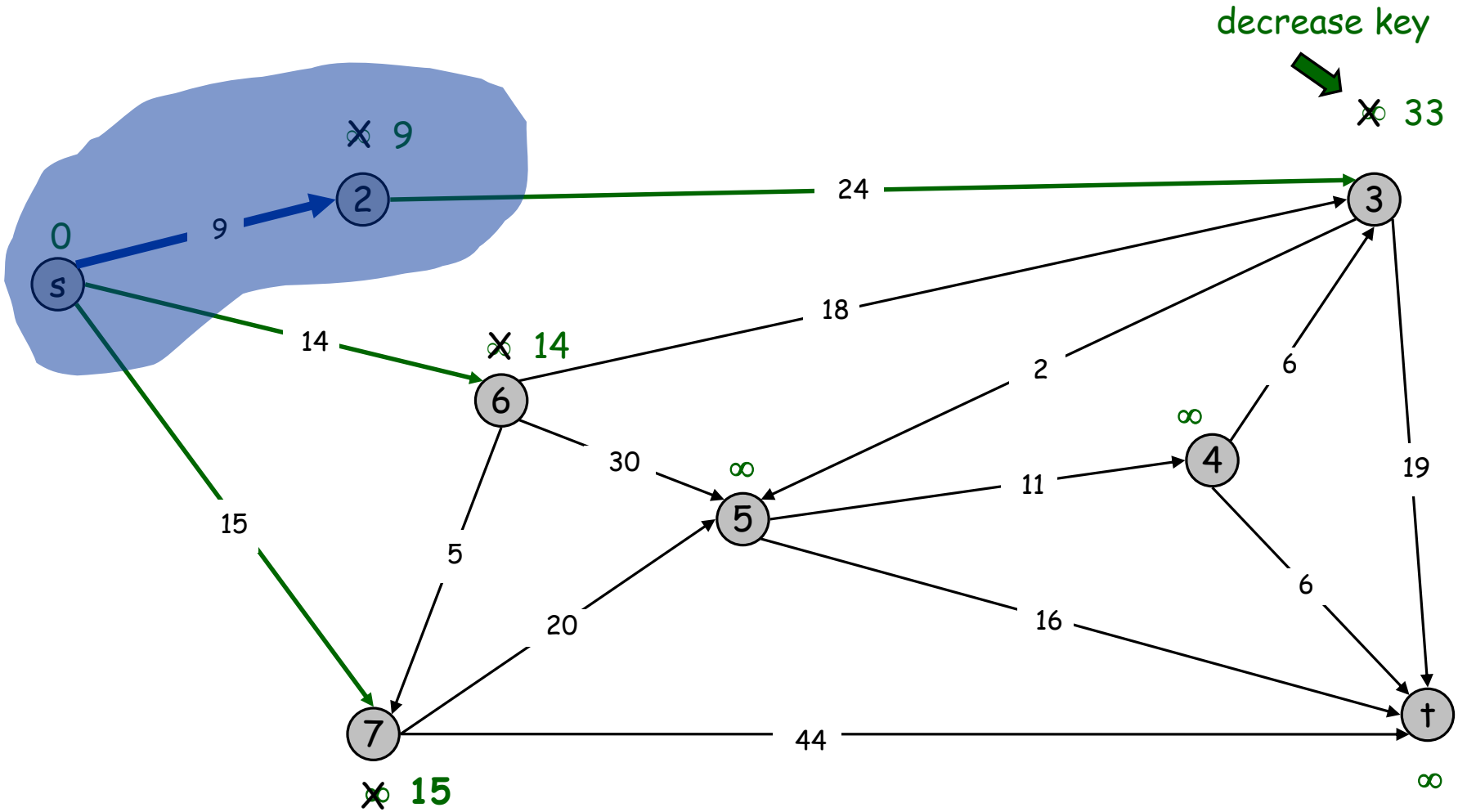
Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



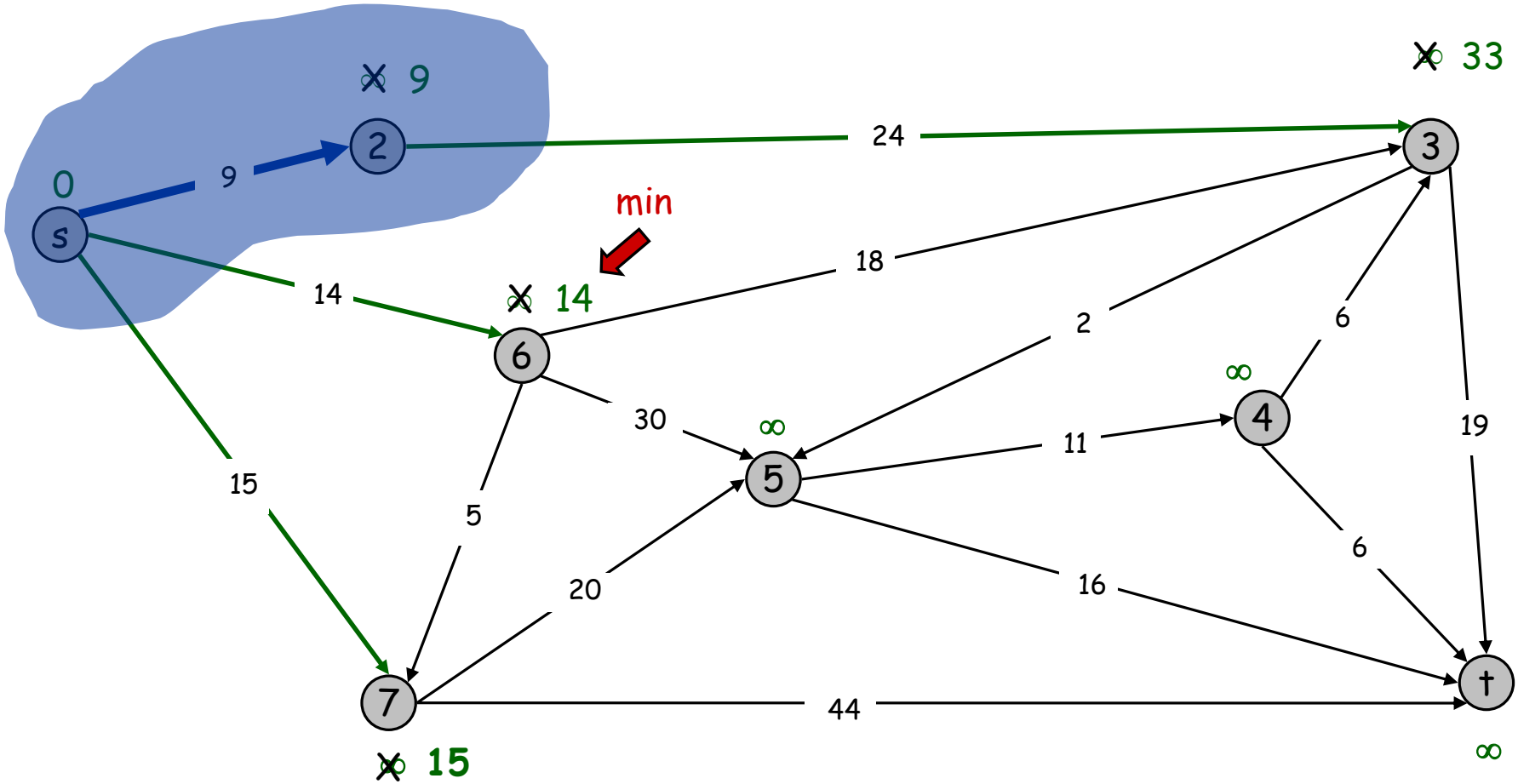
Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



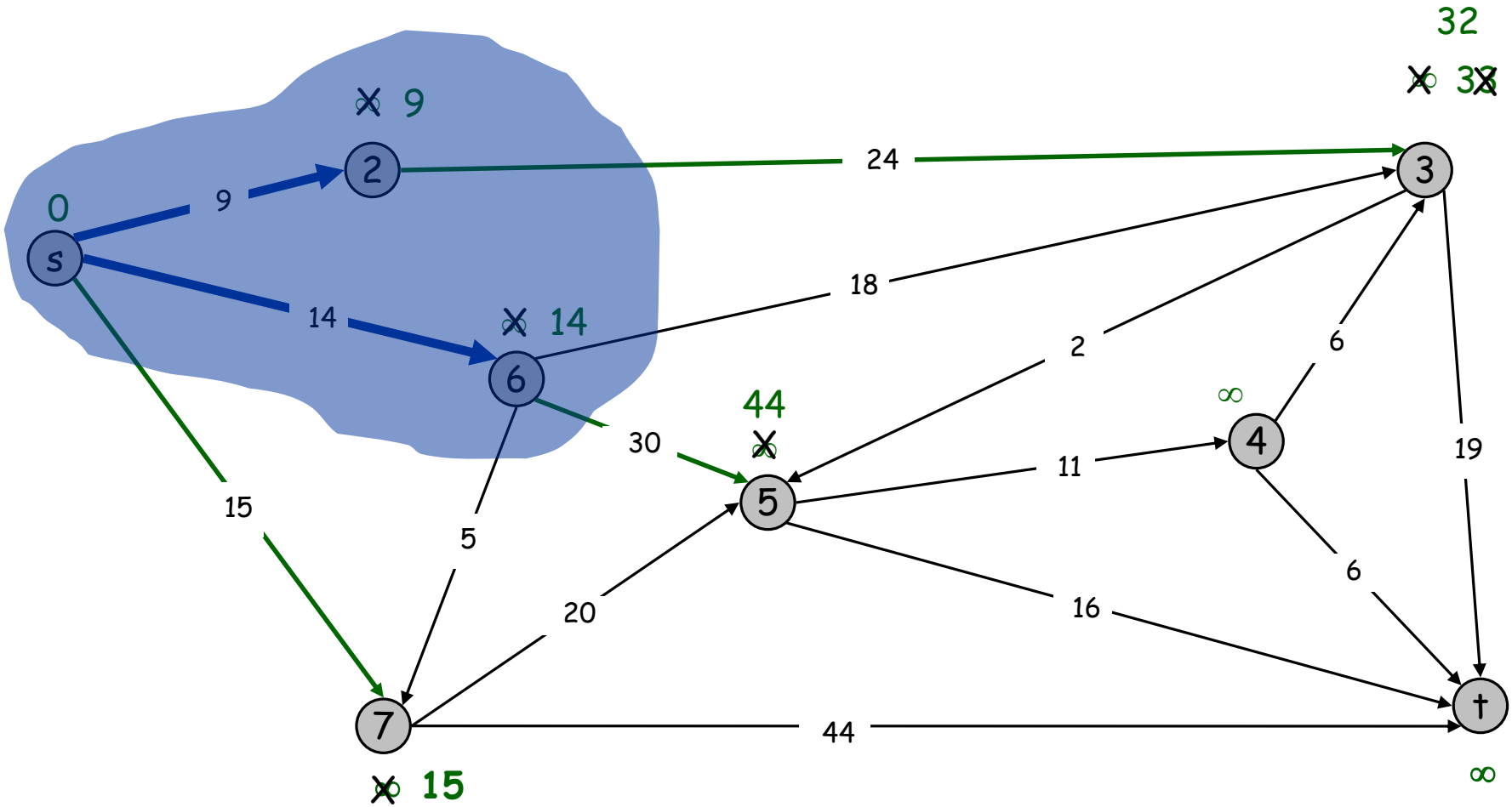
Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



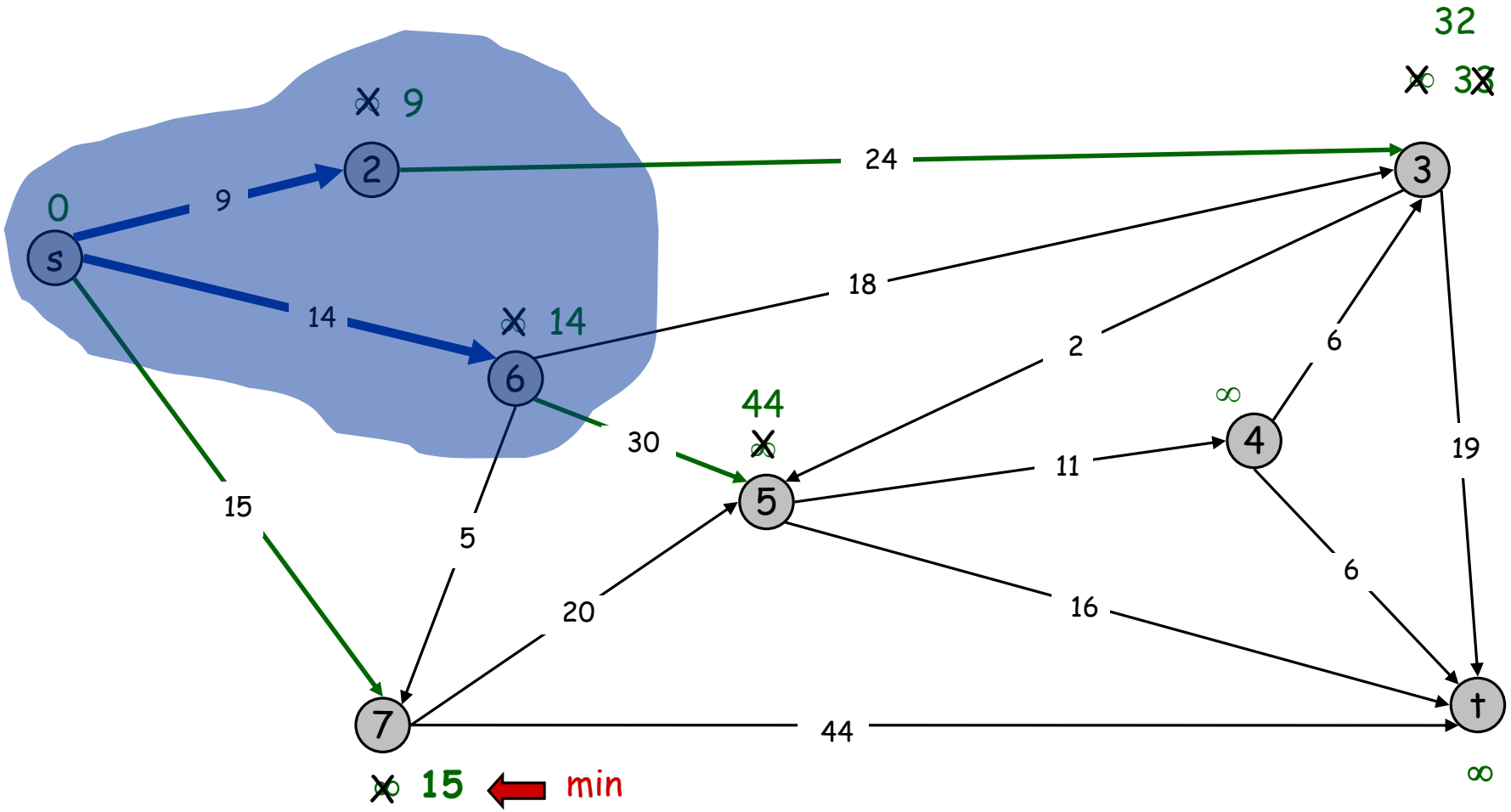
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



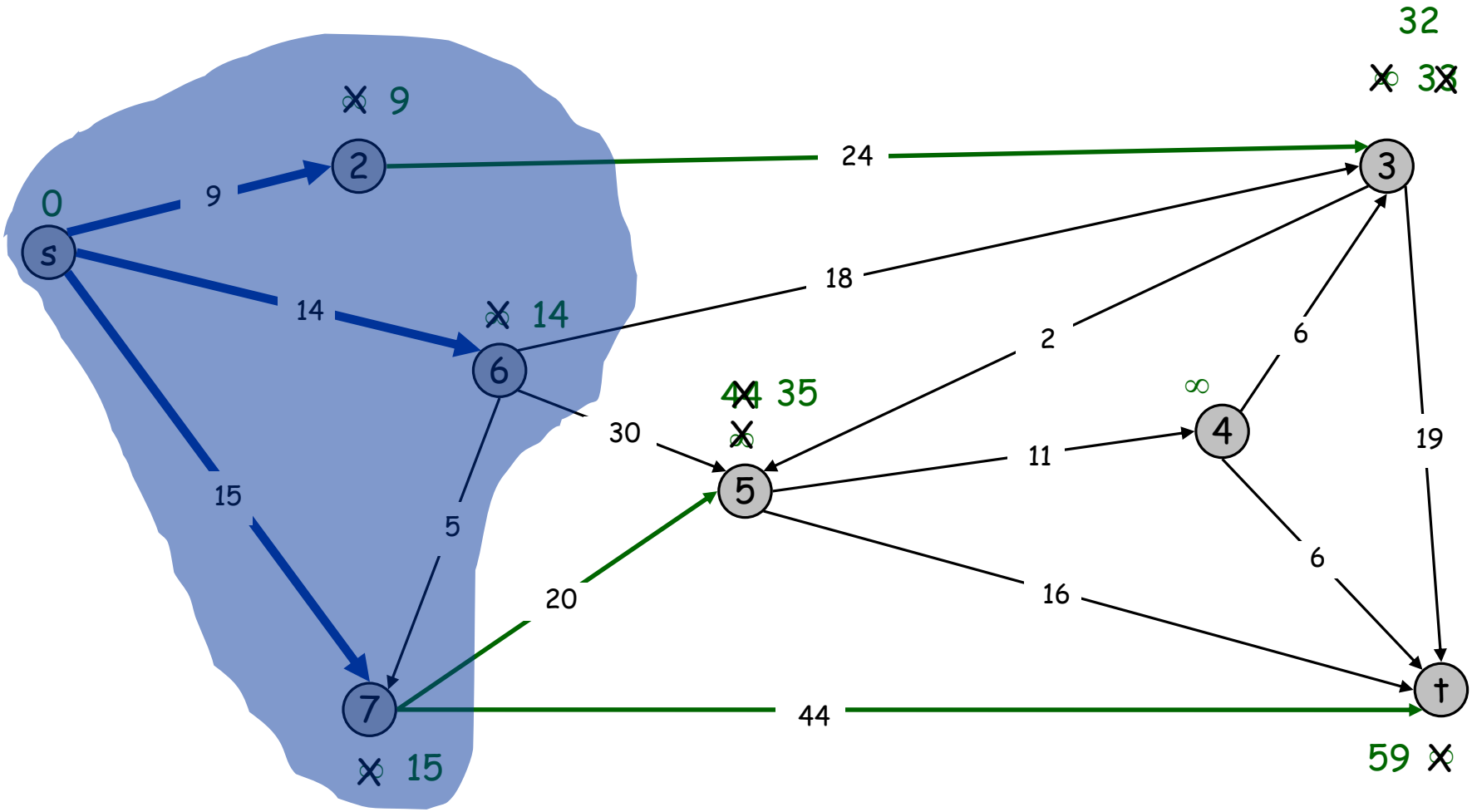
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



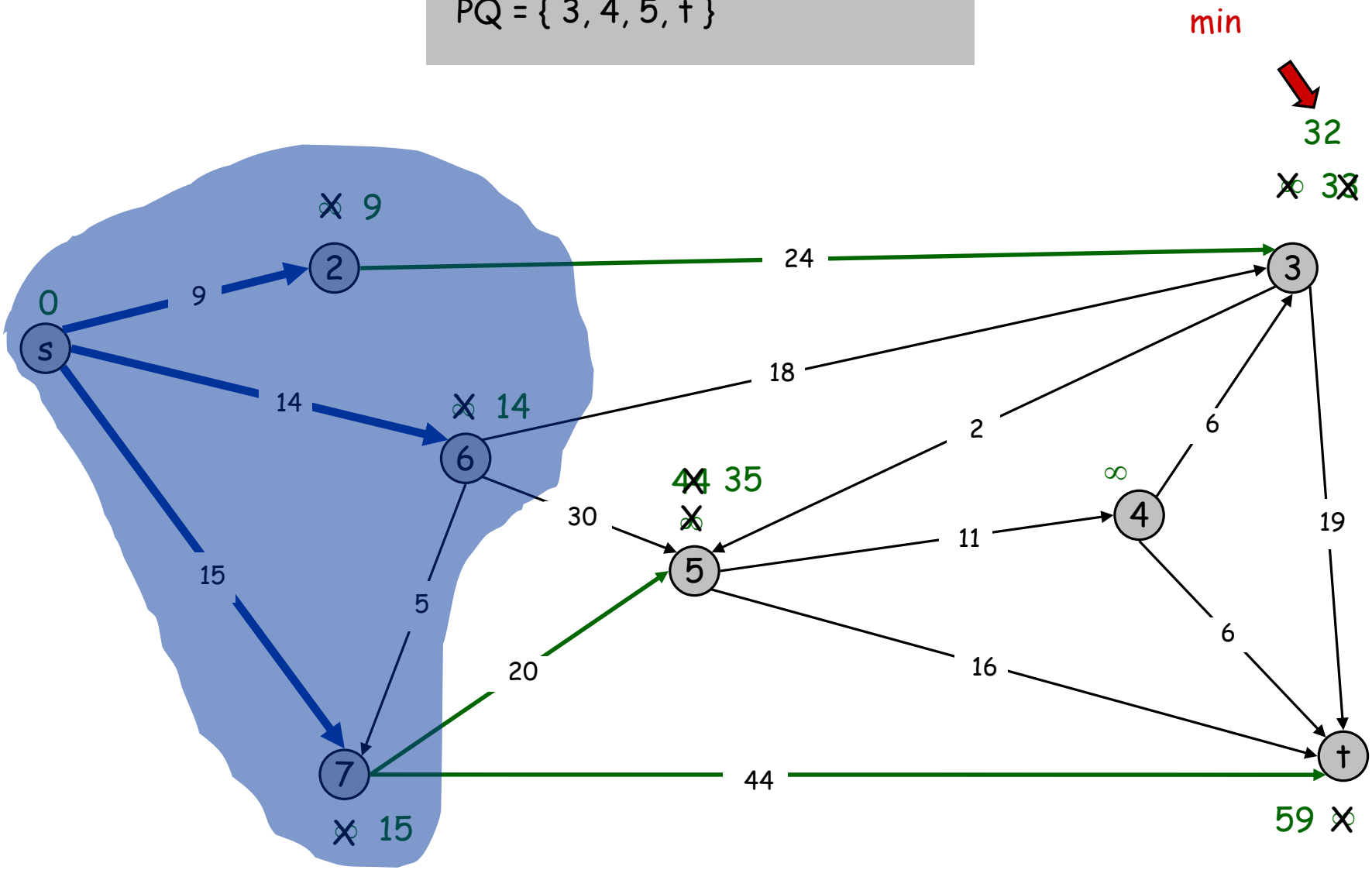
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



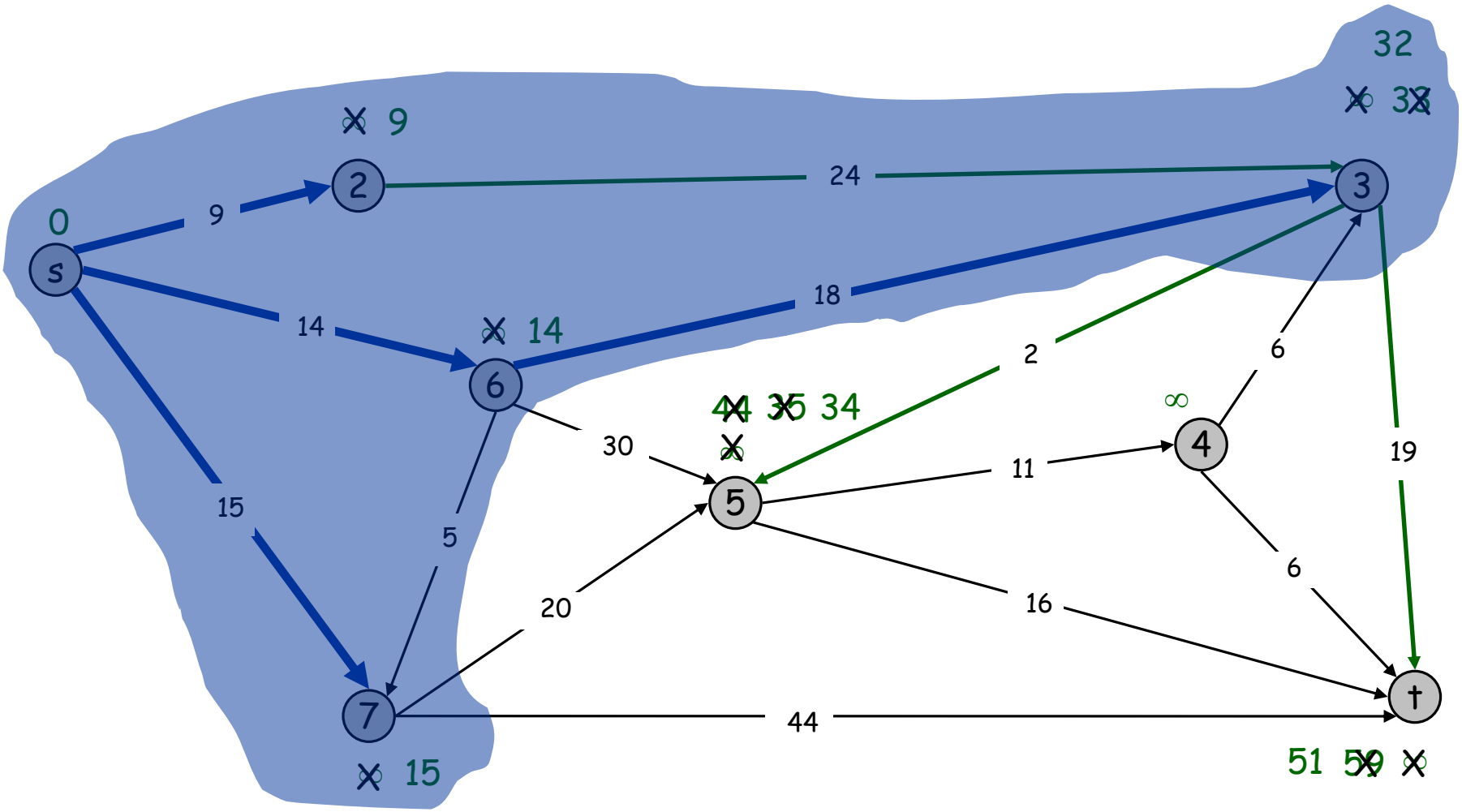
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



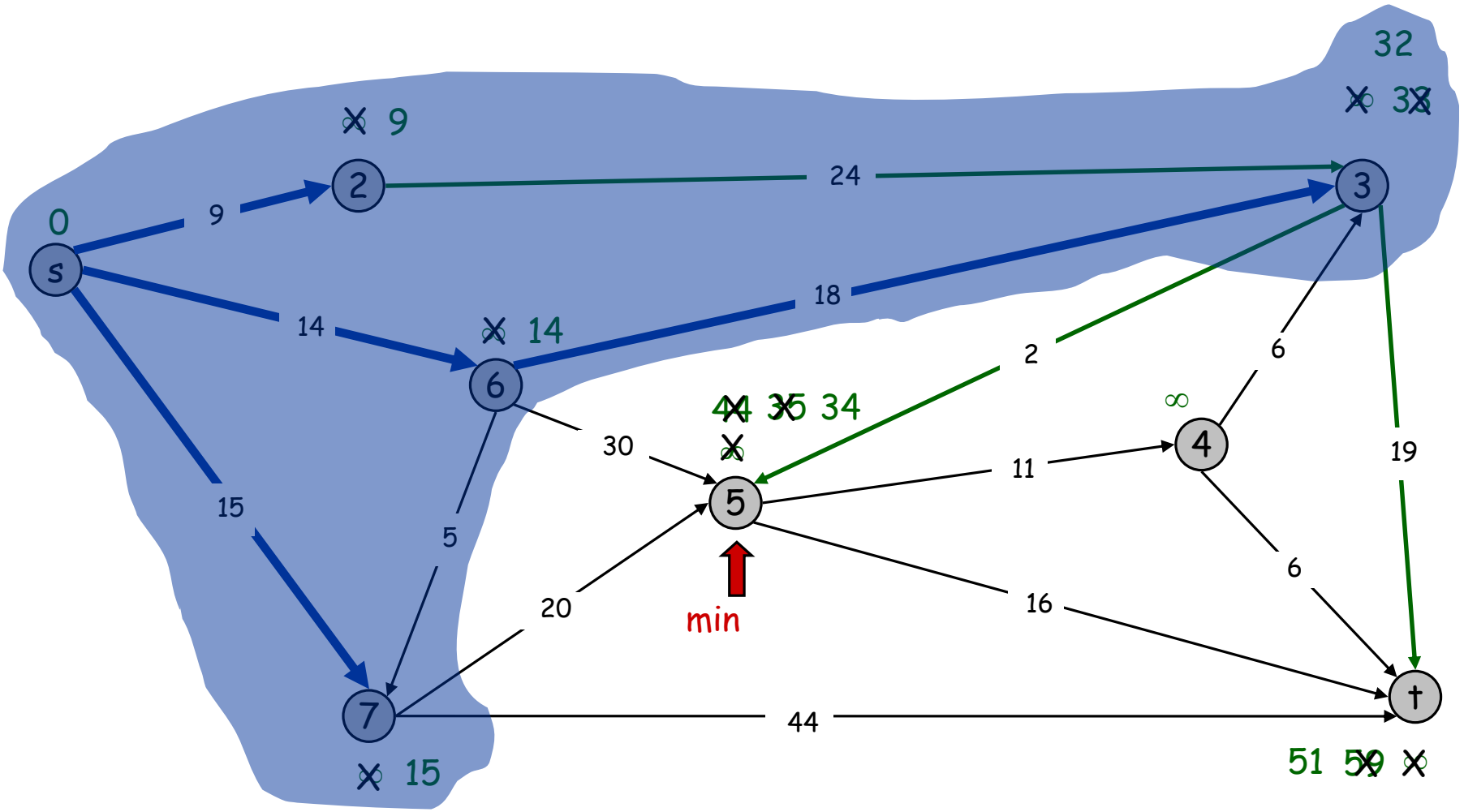
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }



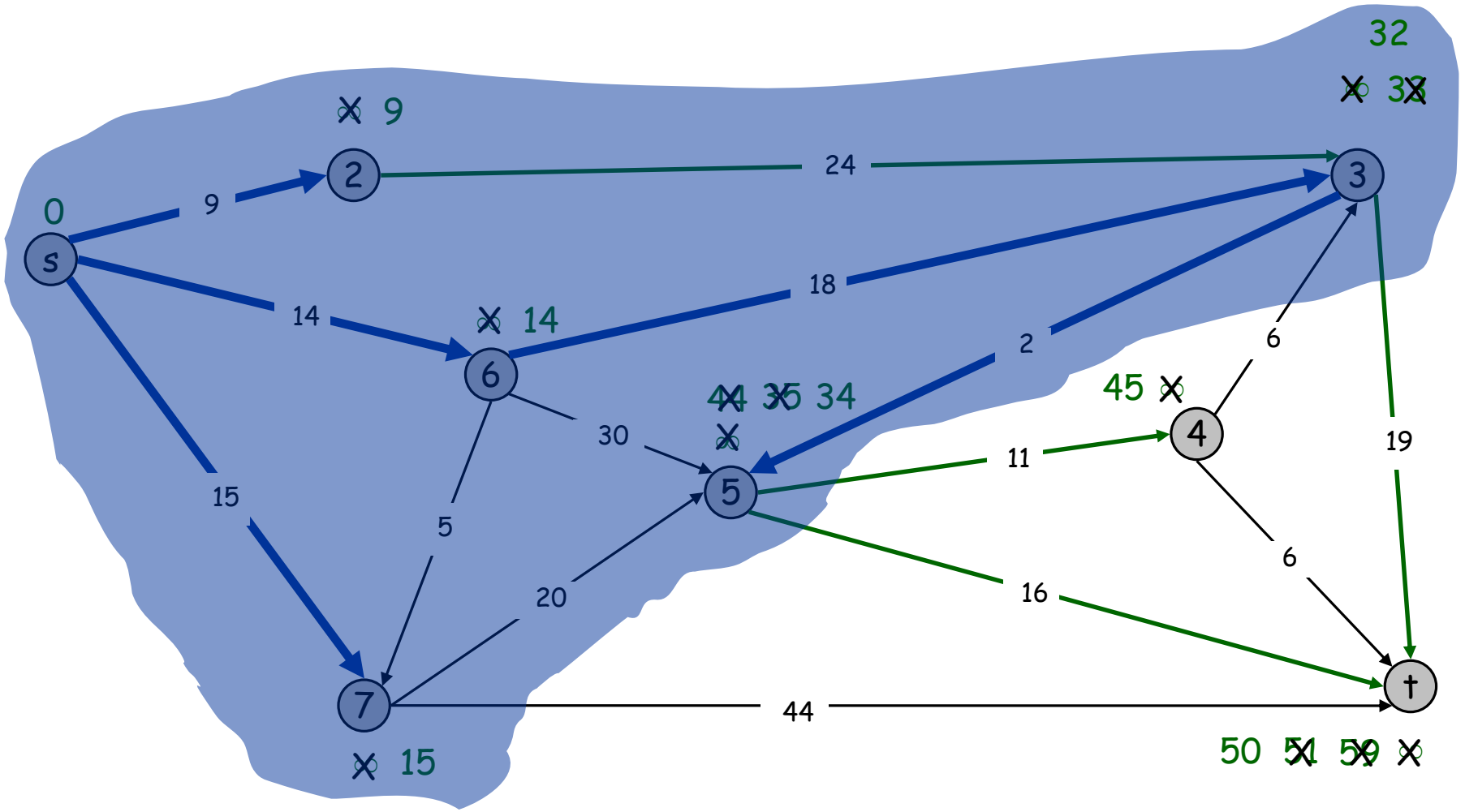
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }



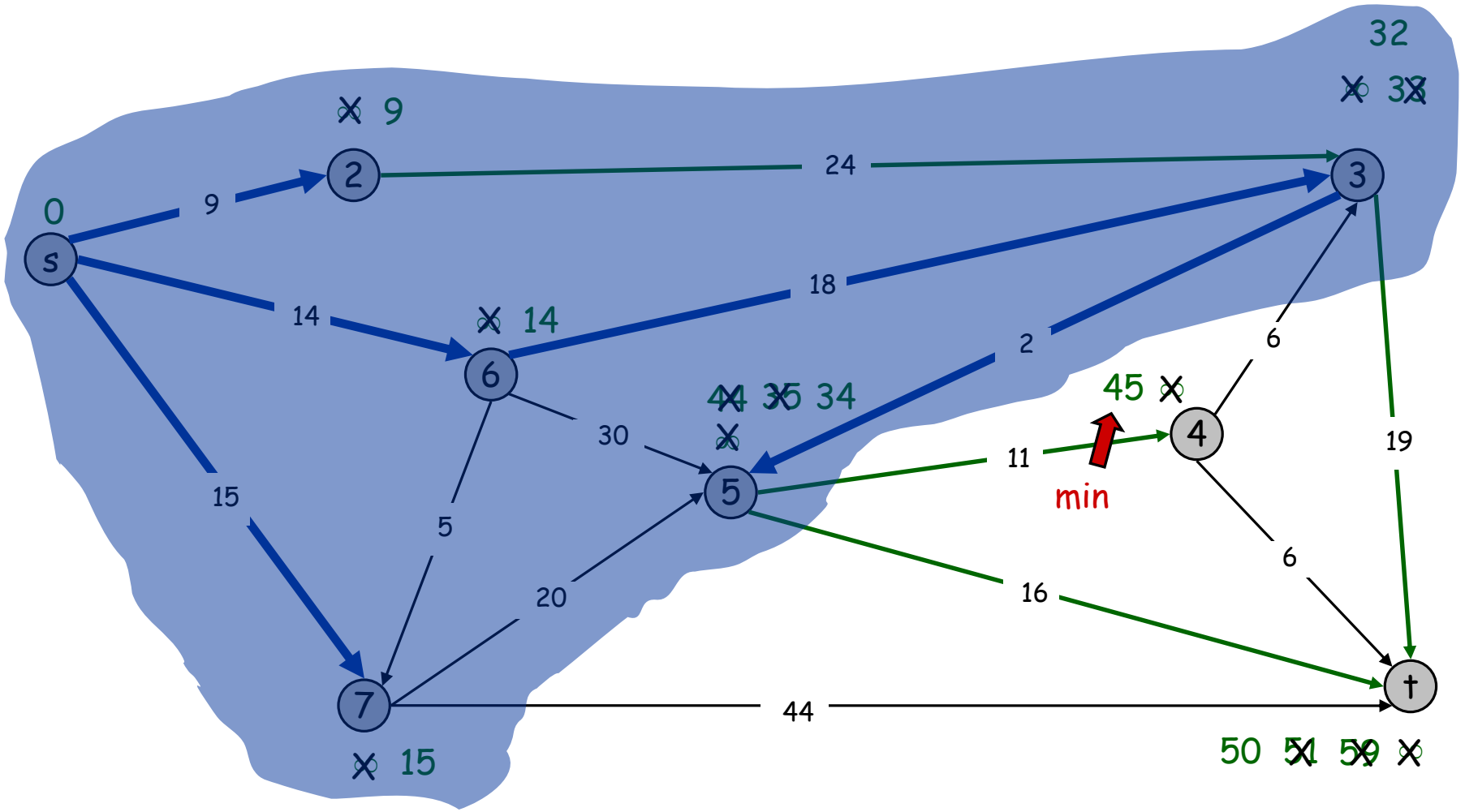
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



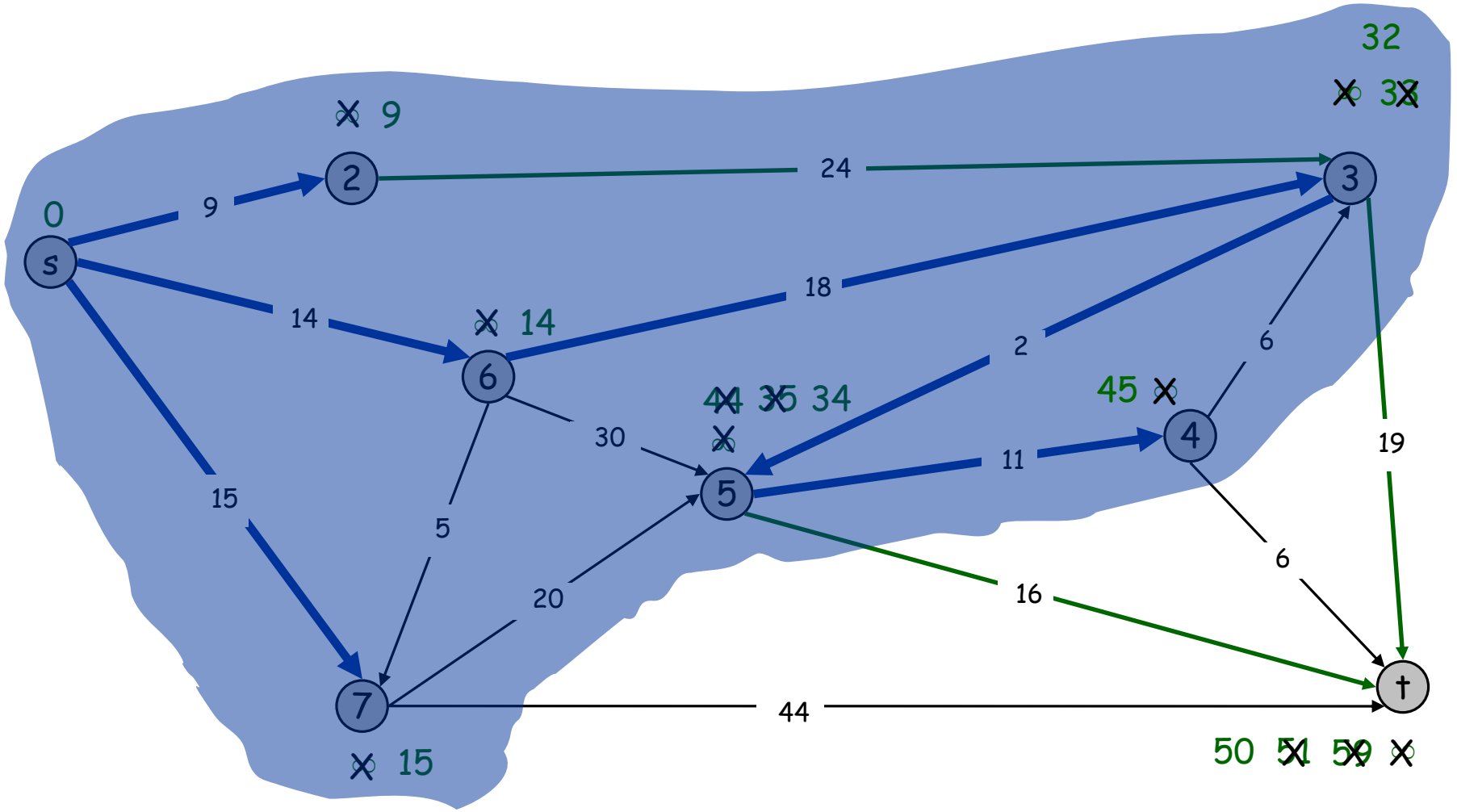
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



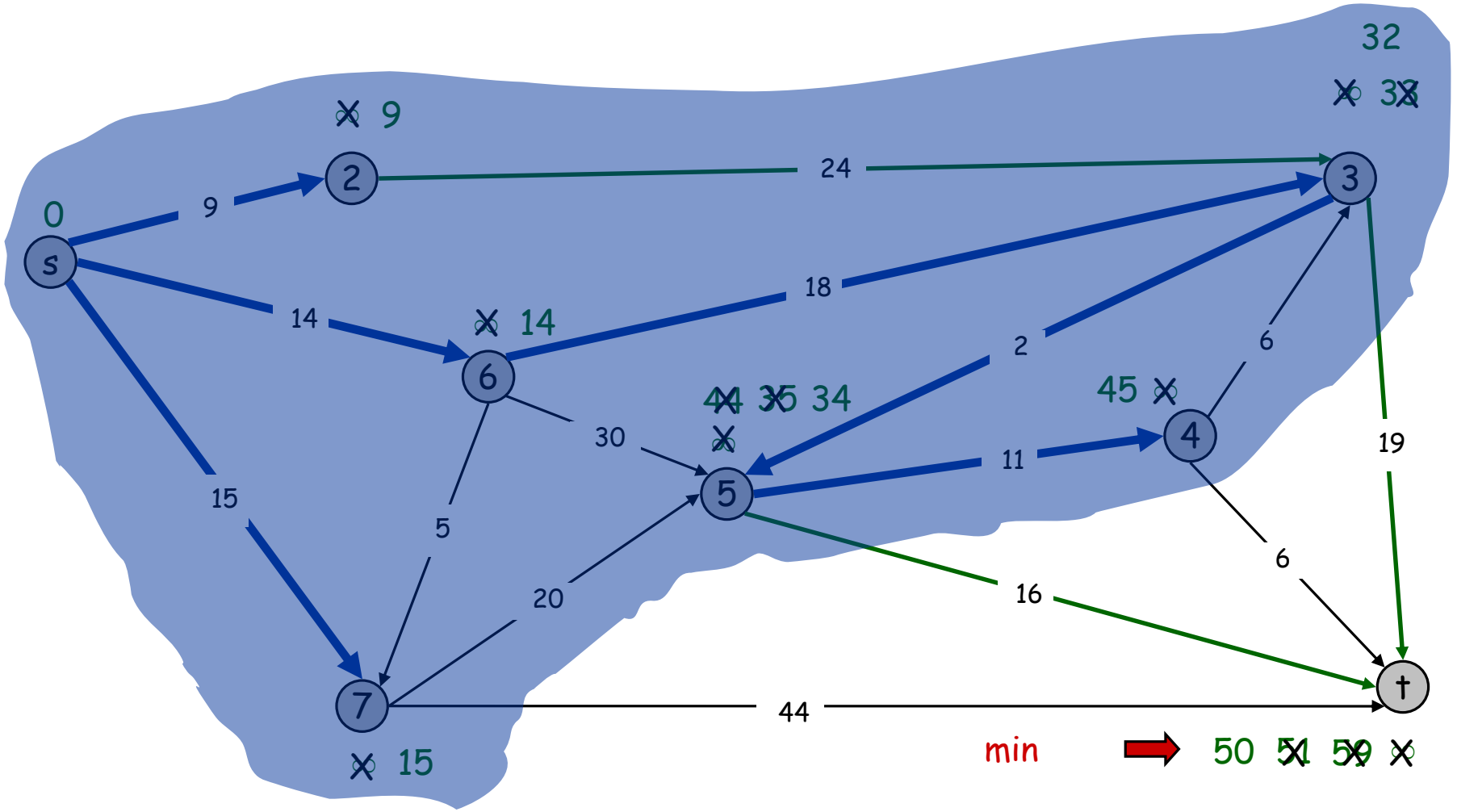
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



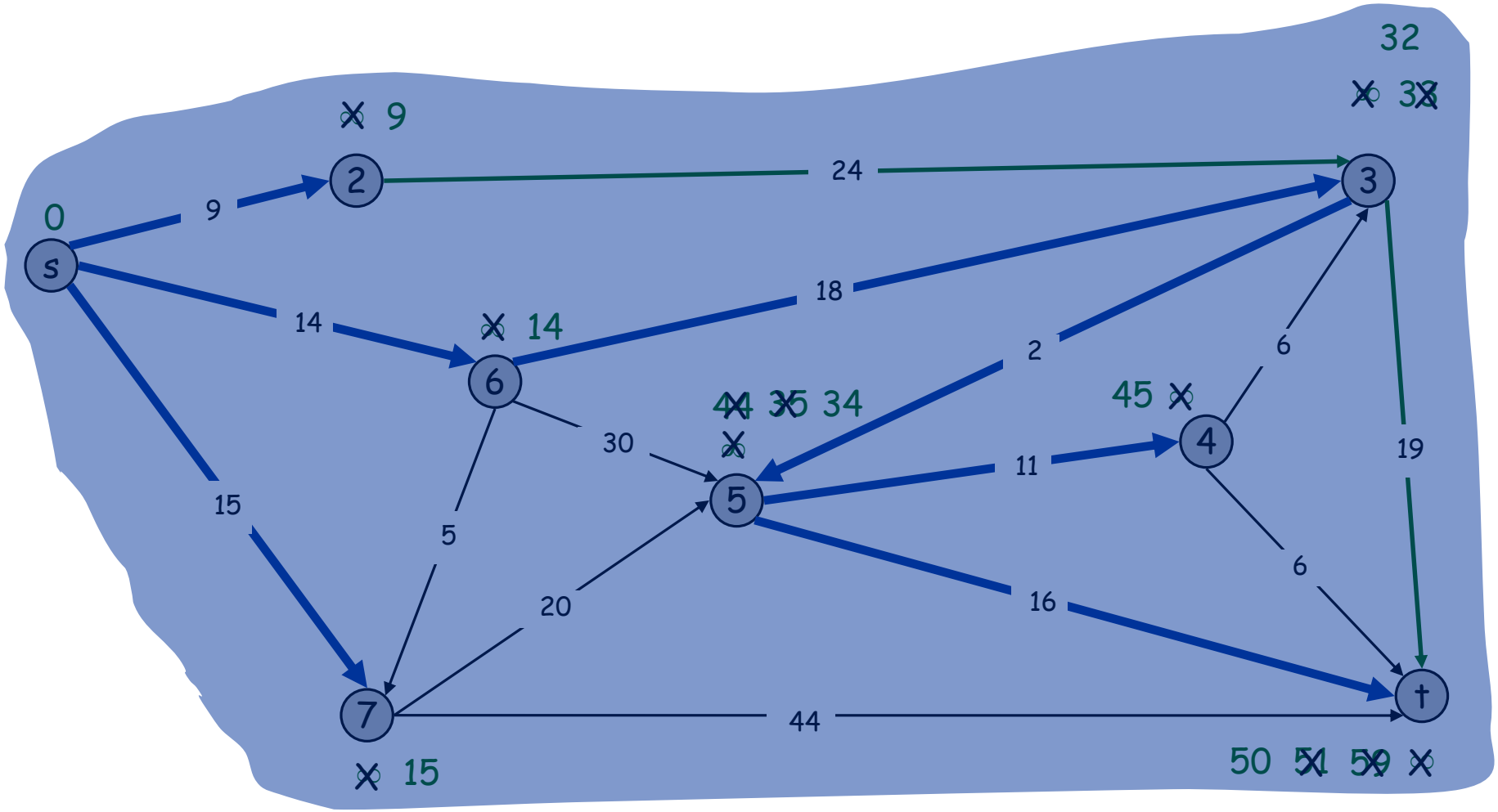
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



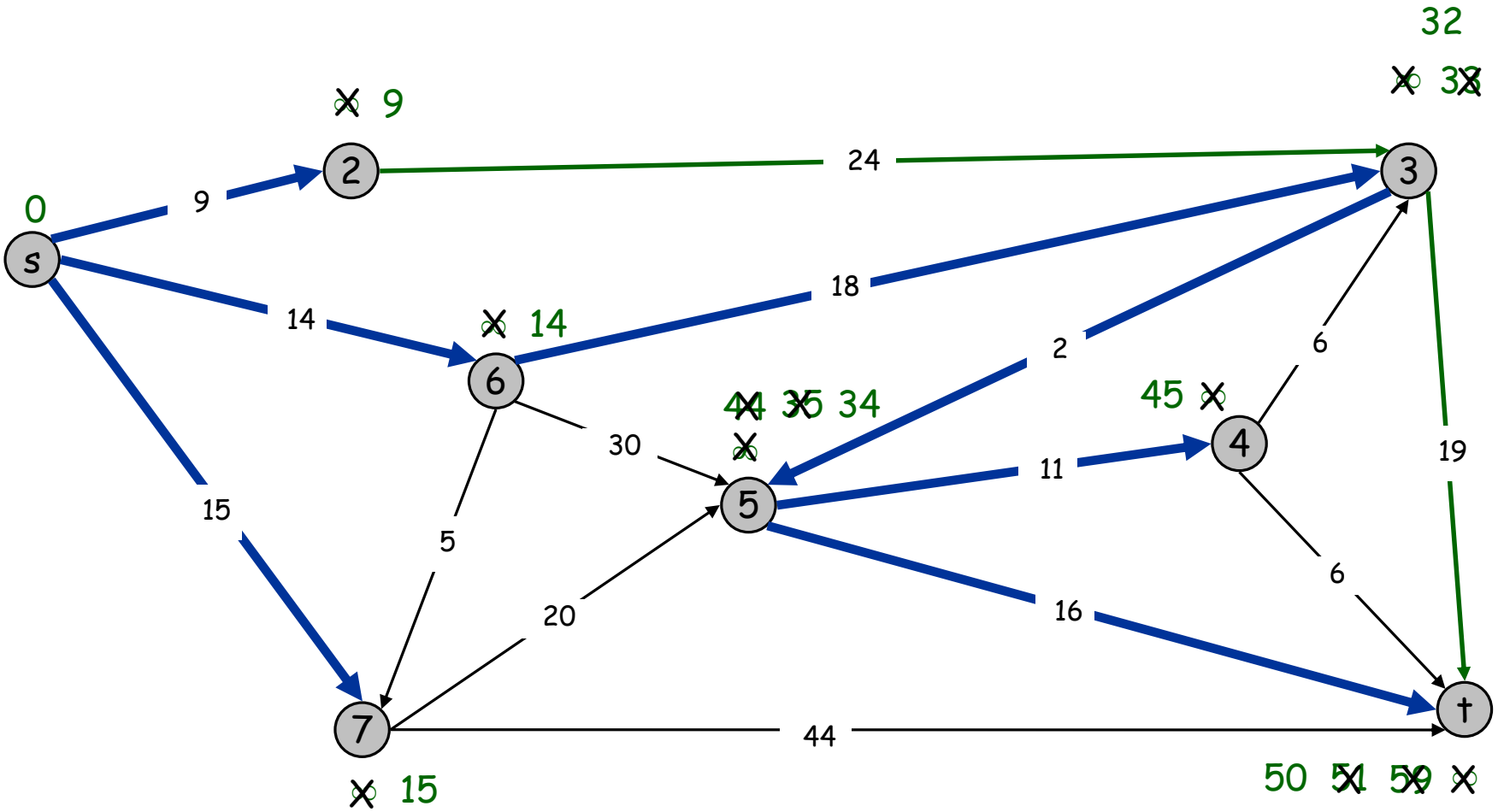
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



Dijkstra's Algorithm: Proof of Correctness

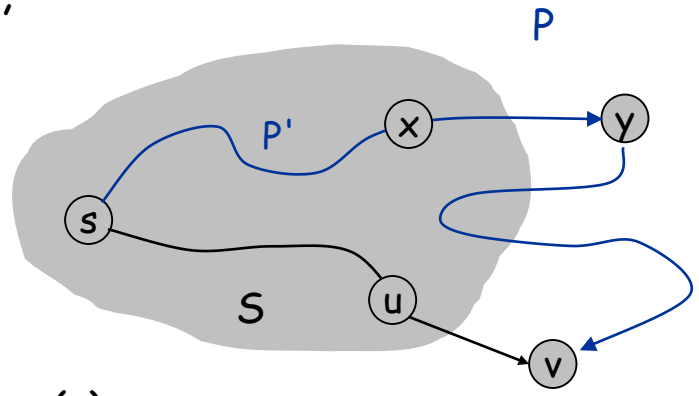
For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.

Pf. (by induction on $|S|$)

Base case: $|S| = 1$ is trivial.

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S, and let u-v be the chosen edge.
- (Claim) The shortest s-u path plus (u, v) is an s-v path of length $\pi(v)$.
- Consider any s-v path P. We'll see that it's no shorter than $\pi(v)$.
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S.



$$\ell(P) \geq \ell(P') + \ell(x,y) \geq d(x) + \ell(x,y) \geq \pi(y) \geq \pi(v)$$

- Nonnegative weights (triangle inequality)
- inductive hypothesis
- defn of $\pi(y)$
- Dijkstra chose v instead of y

Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain $\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e$.

- Next node to explore = node with minimum $\pi(v)$.
- When exploring v, for each incident edge e = (v, w), update

$$\pi(w) = \min \{\ \pi(w),\ \pi(v) + \ell_e \}.$$

Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.

PQ Operation	Dijkstra	Array	Binary heap
Insert	n	n	log n
ExtractMin	n	n	log n
ChangeKey	m	1	log n
IsEmpty	n	1	1
Total		n^2	m log n

Edsger W. Dijkstra

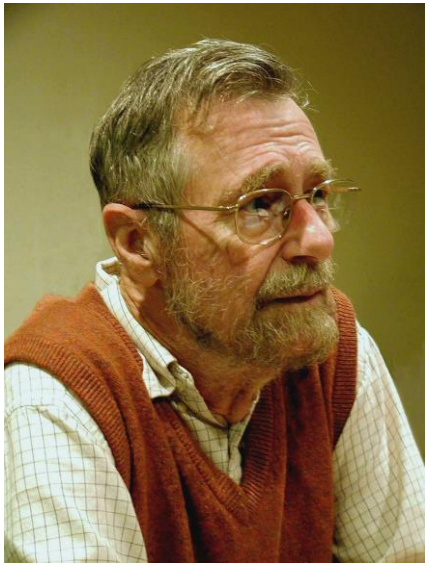
The question of whether computers can think is like the question of whether submarines can swim.

Do only what only you can do.

In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.

The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.

APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.

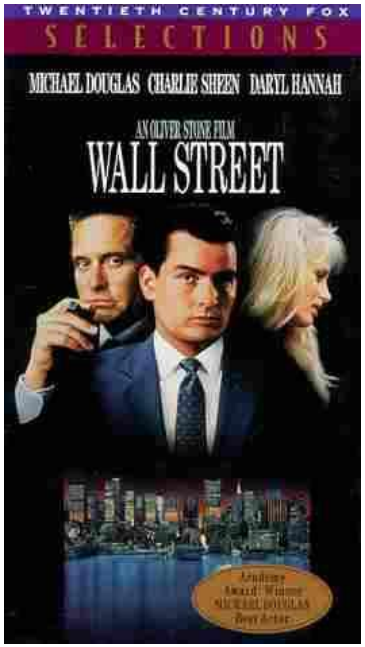


Extra Slides

Coin Changing

Greed is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit.

- Gordon Gecko (Michael Douglas)



Coin Changing

Goal. Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

Ex: 34¢.

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

Ex: $2.89.

Coin-Changing: Greedy Algorithm

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

```
Sort coins denominations by value: c_1 < c_2 < … < c_n.

S ← ϕ        (coins selected)
while (x ≠ 0) {
   let k be largest integer such that c_k ≤ x
   if (k = 0)
      return "no solution found"
   x ← x - c_k
   S ← S ∪ {k}
}
return S
```

Q. Is cashier's algorithm optimal?

Coin-Changing: Analysis of Greedy Algorithm

Theorem. Greedy algorithm is optimal for U.S. coinage: 1, 5, 10, 25, 100.

Pf. (by induction on x)

- Consider optimal way to change $c_k \le x < c_{k+1}$: greedy takes coin k.
- We claim that any optimal solution must also take coin k.
  - if not, it needs enough coins of type $c_1, \ldots, c_{k-1}$ to add up to x
  - table below indicates no optimal solution can do this
- Problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by greedy algorithm. ▪

k	c_k	All optimal solutions must satisfy	Max value of coins 1, 2, …, k-1 in any OPT
1	1	$P \le 4$	-
2	5	$N \le 1$	4
3	10	$N + D \le 2$	4 + 5 = 9
4	25	$Q \le 3$	20 + 4 = 24
5	100	no limit	75 + 24 = 99

Coin-Changing: Analysis of Greedy Algorithm

Observation. Greedy algorithm is sub-optimal for US postal denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

Counterexample. 140¢.

- Greedy: 100, 34, 1, 1, 1, 1, 1, 1.
- Optimal: 70, 70.