南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

04 Greedy Algorithms

CS216 Algorithm Design and Analysis (H)

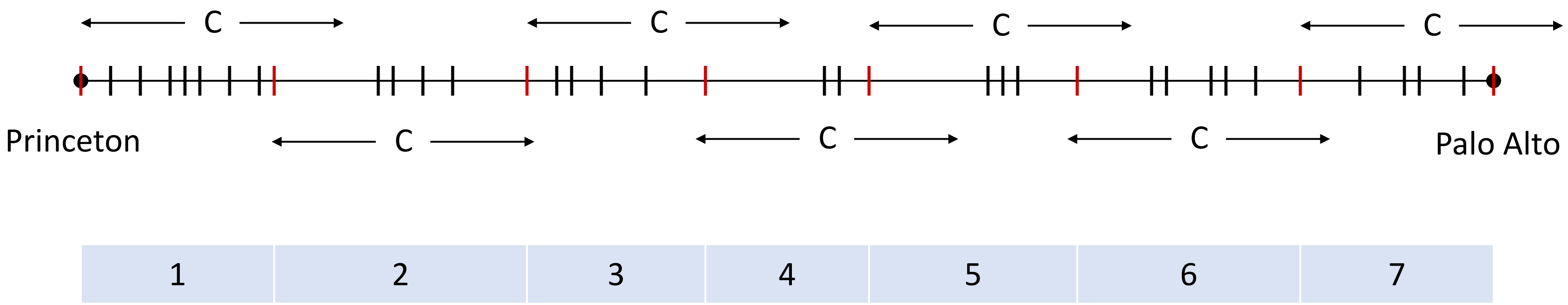
Instructor: Shan Chen

chens3@sustech.edu.cn

Example A: Selecting Breakpoints

Selecting Breakpoints

- **Selecting breakpoints:**
  - Road trip from Princeton to Palo Alto along fixed route.
  - Refueling stations at certain points $b_1, b_2, \ldots, b_n$ on the route.
  - Fuel capacity: c
  - Goal: makes as few refueling stops as possible
- **Greedy approach:** Go as far as you can before refueling.



Selecting Breakpoints: Greedy Algorithm

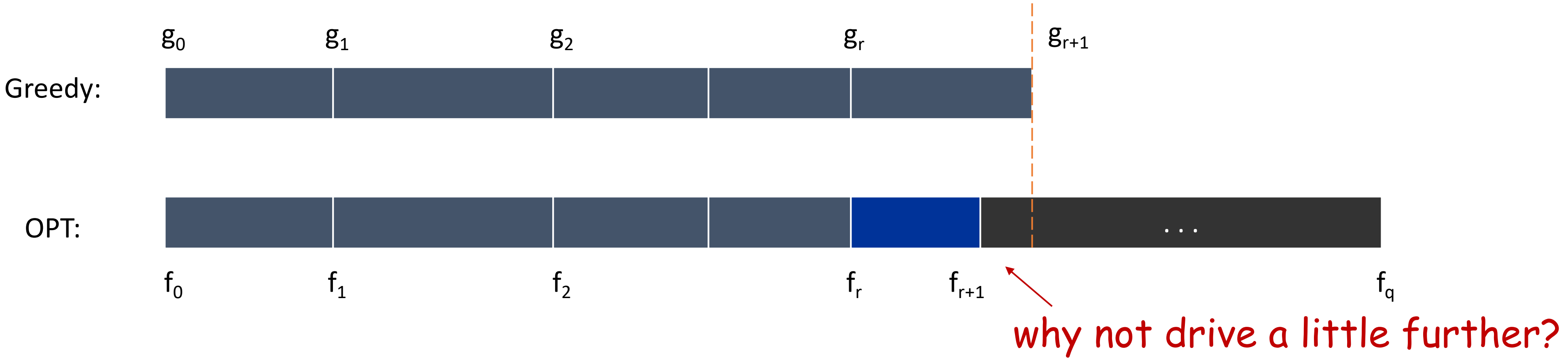
- **Truck driver's algorithm.** Go as far as you can before refueling.

```
Sort breakpoints so that: 0 = b0 < b1 < b2 < ... < bn = L
S ← {0}      ← selected breakpoints
x ← 0        ← current location
while (x ≠ bn)
   let p be largest integer such that bp ≤ x + c
   if (bp = x)
      return "no solution"
   x ← bp
   S ← S ∪ {p}
return S
```

- **Time complexity.** *O(n log n)*
  - sorting breakpoints: *O(n log n)*
  - selecting all breakpoints: *O(n)*

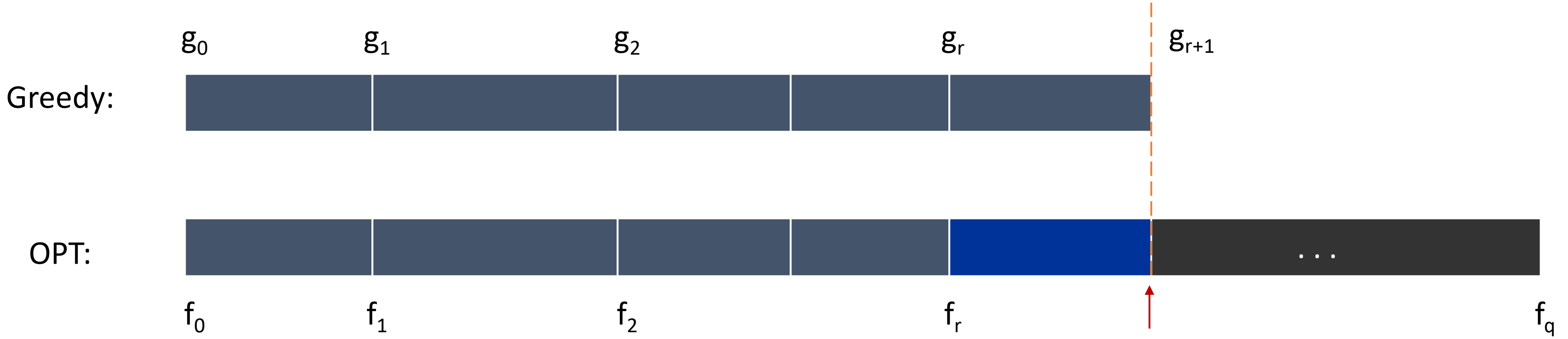
Selecting Breakpoints: Correctness

- **Theorem.** Truck driver's algorithm is optimal.
- **Pf.** (by contradiction)
  - Assume greedy algorithm is not optimal, and let's see what happens.
  - Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote the set of breakpoints chosen greedily.
  - Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote the set of breakpoints in an optimal solution with the largest possible value of r such that $f_0 = g_0, f_1 = g_1, \ldots, f_r = g_r$.
  - Note: $g_{r+1} > f_{r+1}$ by greedy choice of algorithm.



Selecting Breakpoints: Correctness

- **Theorem.** Truck driver's algorithm is optimal.
- **Pf.** (by contradiction)
  - Assume greedy algorithm is not optimal, and let's see what happens.
  - Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote the set of breakpoints chosen greedily.
  - Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote the set of breakpoints in an optimal solution with the largest possible value of r such that $f_0 = g_0, f_1 = g_1, \ldots, f_r = g_r$.
  - Note: $g_{r+1} > f_{r+1}$ by greedy choice of algorithm.



an optimal solution that has r + 1 breakpoints in common, contradiction!

Example B:
Coin Changing

Coin Changing

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

- **Example:** *34¢*

- **Greedy approach:** At each iteration, add coin of the largest value that does not take us past the amount to be paid.

- **Example:** *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 coin denominations: c1 < c2 < … < cn.
S ← ϕ      ← selected coins
while (x ≠ 0) {
   let k be largest integer such that ck ≤ x
   if (k = 0)
      return "no solution found"
   x ← x - ck
   S ← S ∪ {k}
}
return S
```

- **Q.** Is the above greedy algorithm optimal?

Coin Changing: Properties of Optimal Solutions

- **Property.** Number of pennies $P \le 4$.
- **Pf.** Replace 5 pennies with 1 nickel.
- **Property.** Number of nickels $N \le 1$.
- **Pf.** Replace 2 nickels with 1 dime.
- **Property.** Number of quarters $Q \le 3$.
- **Pf.** Replace 4 quarters with 1 dollar.
- **Property.** Number of nickels N + number of dimes $D \le 2$.
- **Pf.** Recall: $N \le 1$
  - Replace 3 dimes with 1 quarter and 1 nickel. => $D \le 2$
  - Replace 2 dimes and 1 nickel with 1 quarter. => $N + D \le 2$

Coin Changing: Analysis of Greedy Algorithm

- **Theorem.** Cashier's algorithm is optimal for U.S. coinage: *1, 5, 10, 25, 100*
- **Pf.** (by induction on the amount to be paid x)
  - Consider the way to change $c_k \leq x < c_{k+1}$: greedy takes coin k.
  - We claim that any optimal solution must also take coin k, reducing x to $x - c_k$.
    - ✓ If not, it needs enough coins of type $c_1, \ldots, c_{k-1}$ to sum up to x.
    - ✓ Table below indicates that no optimal solution can do this.

k	c_k	All optimal solutions must satisfy	Max value of coins from $c_1, c_2, \ldots, c_{k-1}$ in any OPT solution
1	1	$P \leq 4$	0
2	5	$N \leq 1$	4
3	10	$N + D \leq 2$	$4 + 5 = 9$
4	25	$Q \leq 3$	$20 + 4 = 24$
5	100	no limit	$75 + 24 = 99$

Coin Changing: Analysis of Greedy Algorithm

- **Observation.** Cashier's algorithm is not optimal for US postal denominations: *1, 10, 21, 34, 70, 100, 350, 1225, 1500*
- **Example.** *140¢*
  - Greedy: *100, 34, 1, 1, 1, 1, 1, 1.*
  - Optimal: *70, 70.*

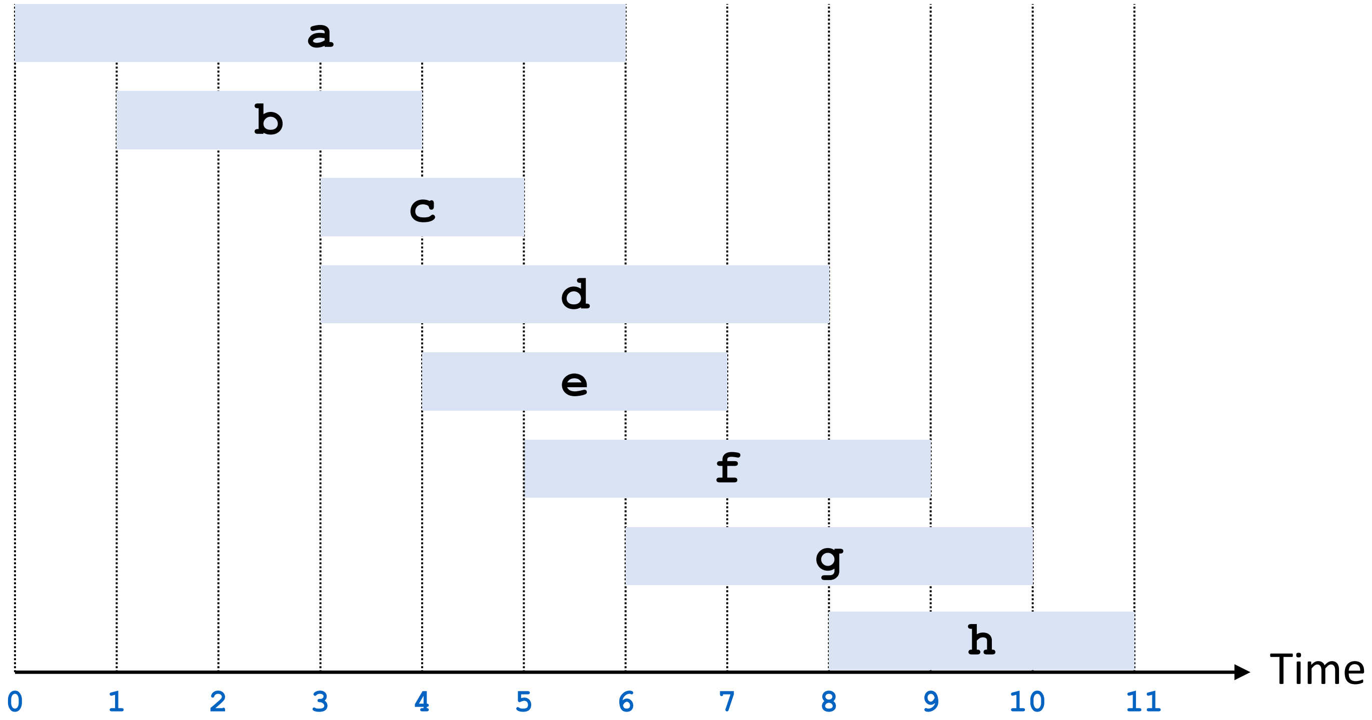
Greedy Algorithms

- Build up a solution in small steps.
- Choose a decision at each step myopically to optimize some underlying criterion.
- May not produce an optimal solution.
- But can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

1. Interval Scheduling

Interval Scheduling

- **Interval scheduling.**
  - Job j starts at s_j and finishes at f_j.
  - Two jobs are 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 if it's compatible with the ones already taken.
  - [Earliest start time] Consider jobs in ascending order of s_j.

    Counterexample:

  - [Earliest finish time] Consider jobs in ascending order of f_j.
  - [Shortest interval] Consider jobs in ascending order of f_j - s_j.

    Counterexample:

  - [Fewest conflicts] For each job j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

    Counterexample:

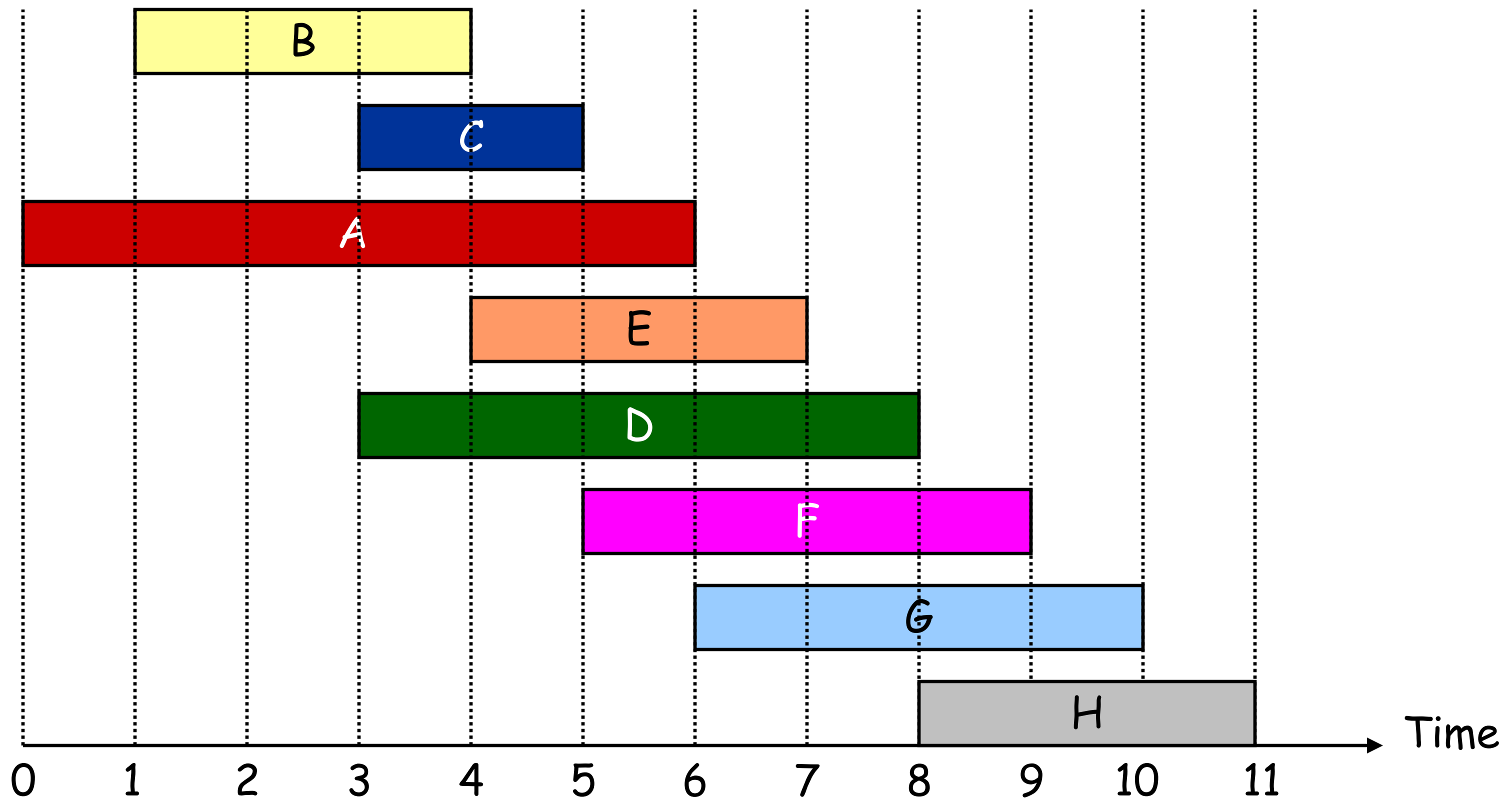
Interval Scheduling: Greedy Algorithm

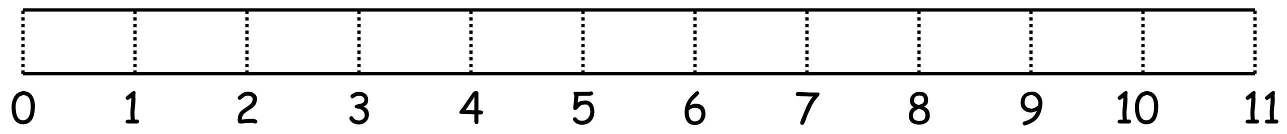
- **Greedy algorithm.** Consider jobs in the increasing order of finish time. Take each job if it's compatible with the ones already taken.

```
Sort jobs by finish times so that f₁ ≤ f₂ ≤ ... ≤ fₙ.
A ← ϕ   ← selected jobs
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

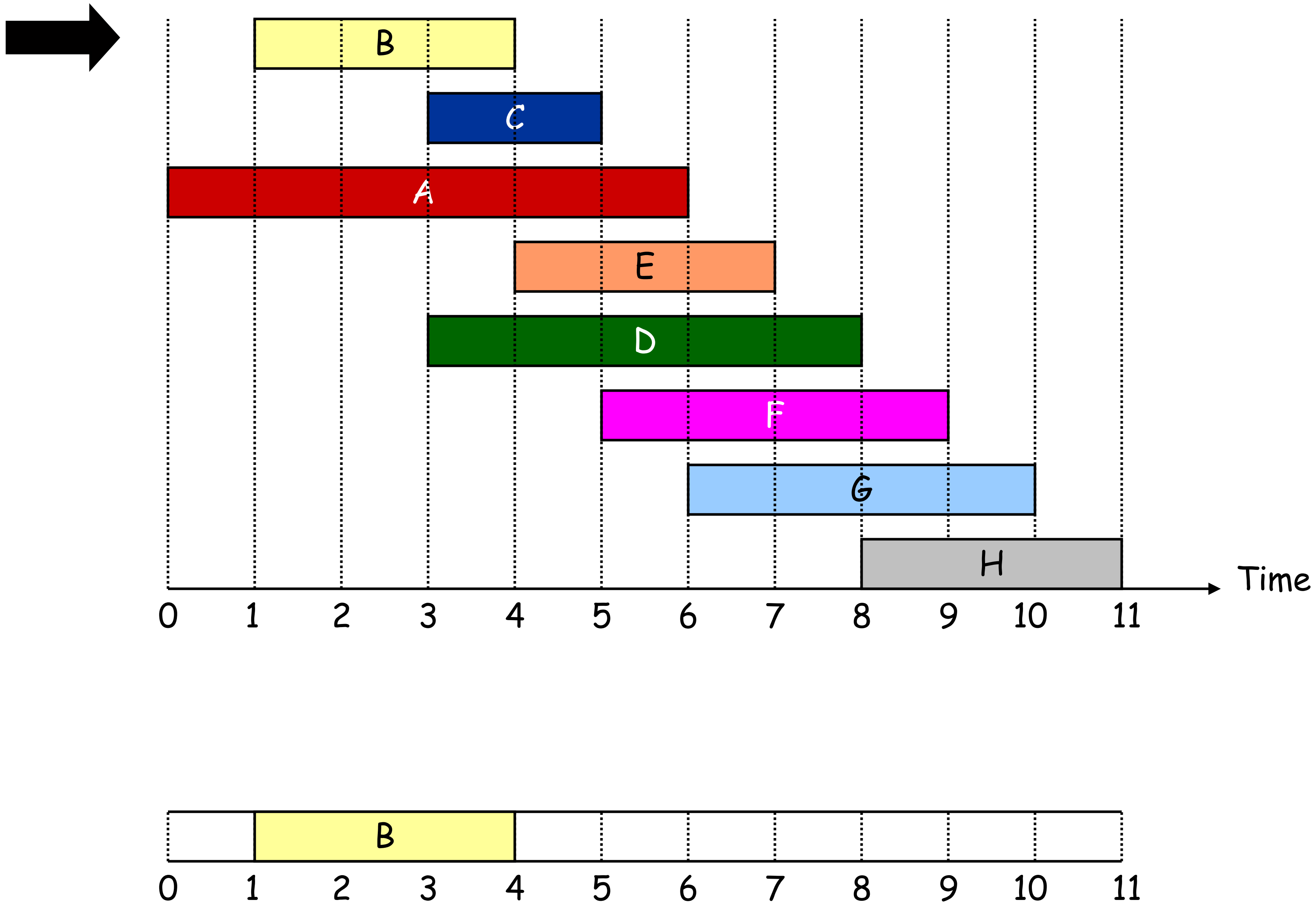
- **Time complexity.** *O(n log n)*
  - Let job j^* be the job that was last added to A.
  - Job j is compatible with A if and only if $s_j \geq f_{j^*}$.

Interval Scheduling

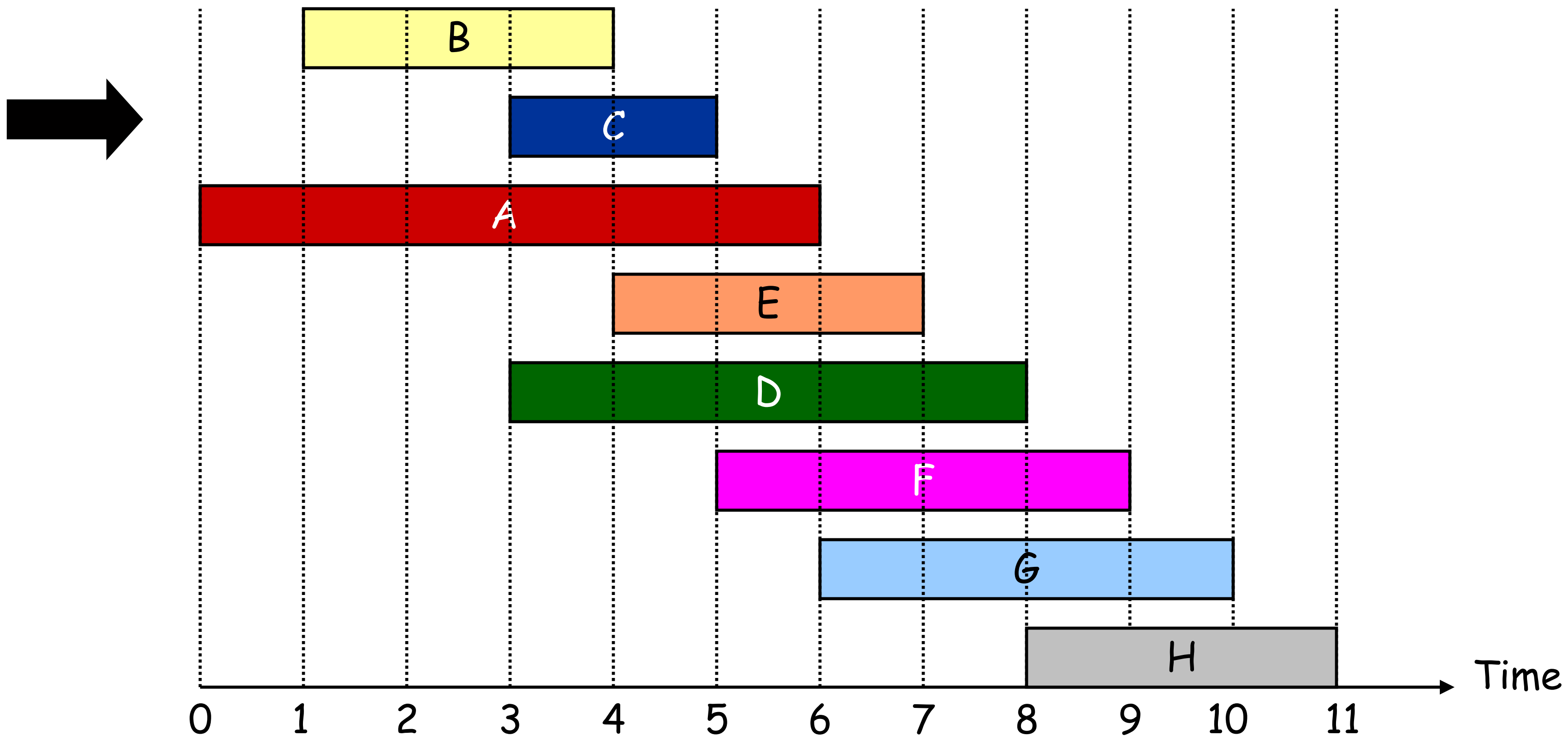


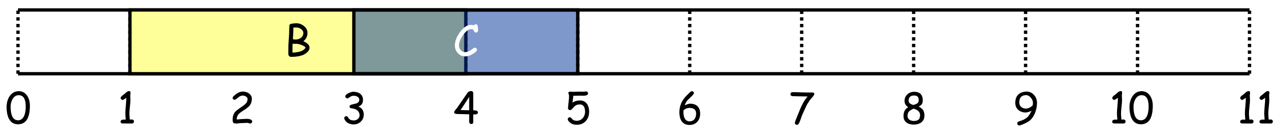


Interval Scheduling

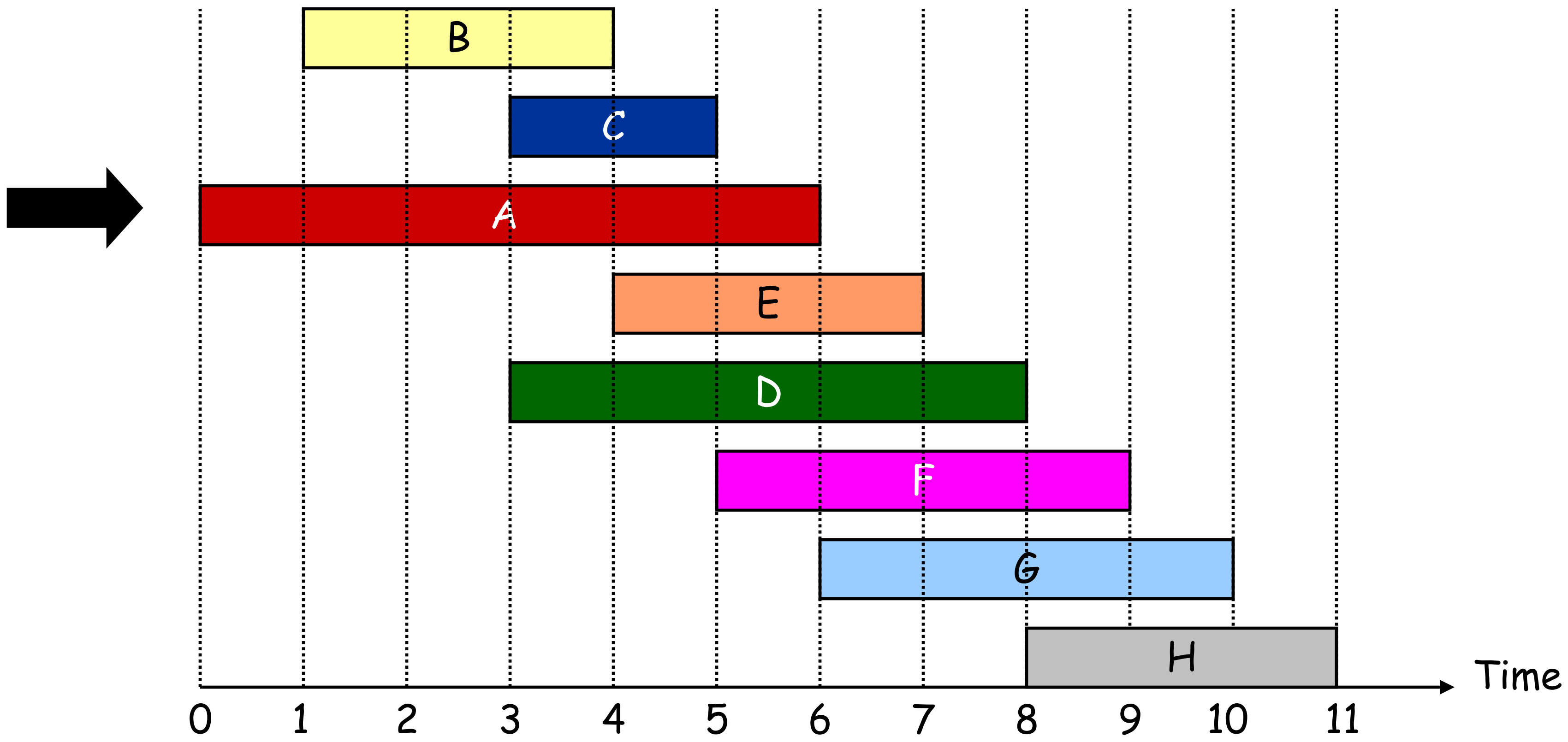


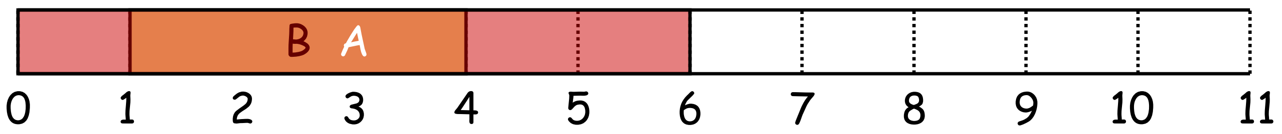
Interval Scheduling



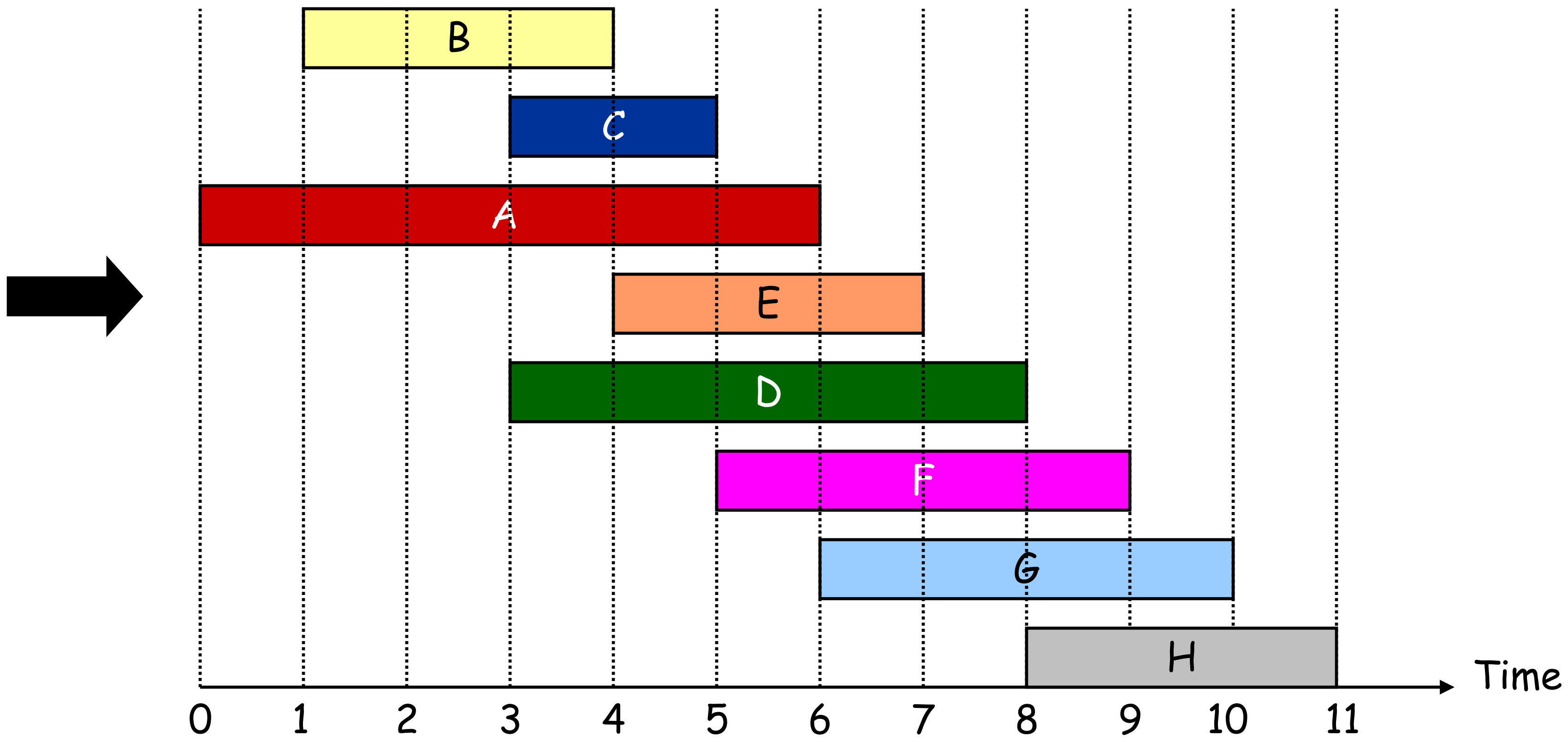


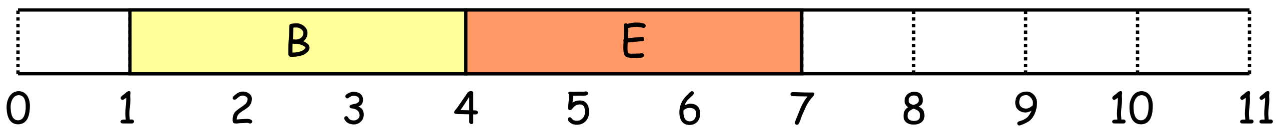
Interval Scheduling



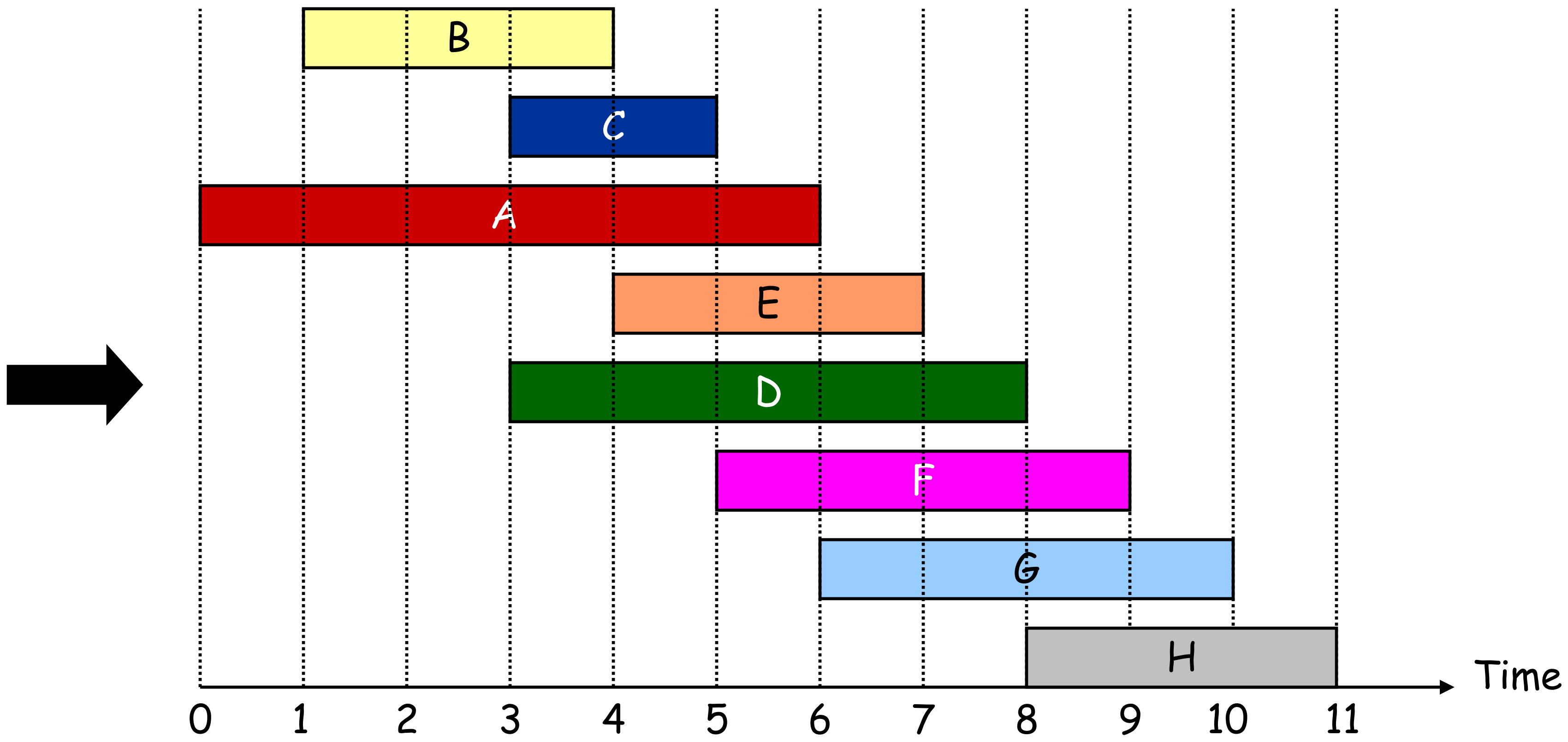


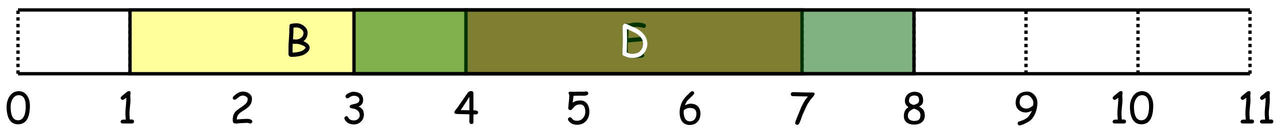
Interval Scheduling



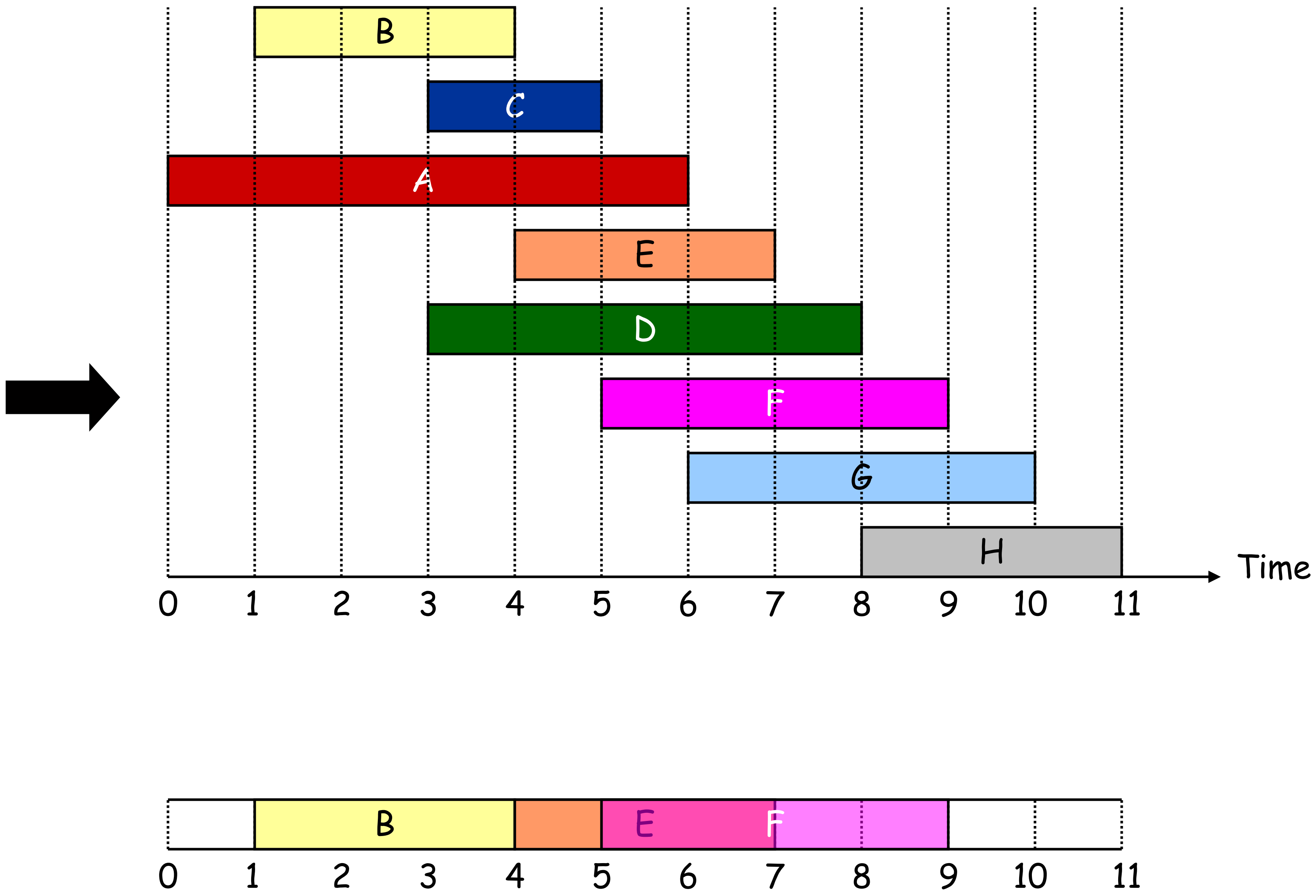


Interval Scheduling

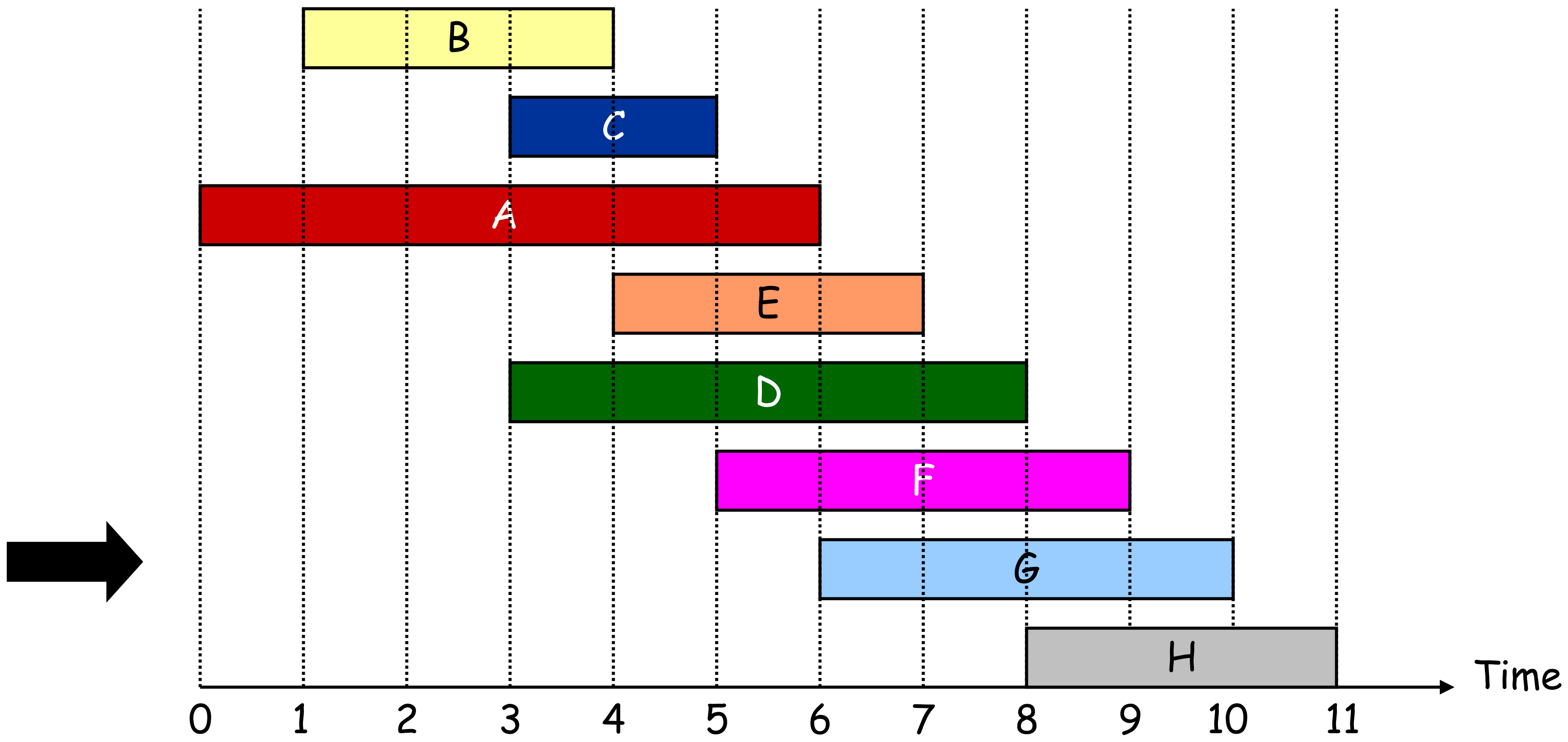


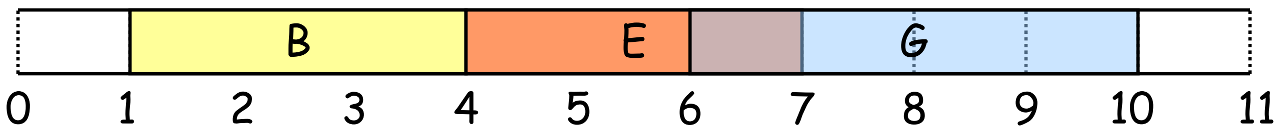


Interval Scheduling

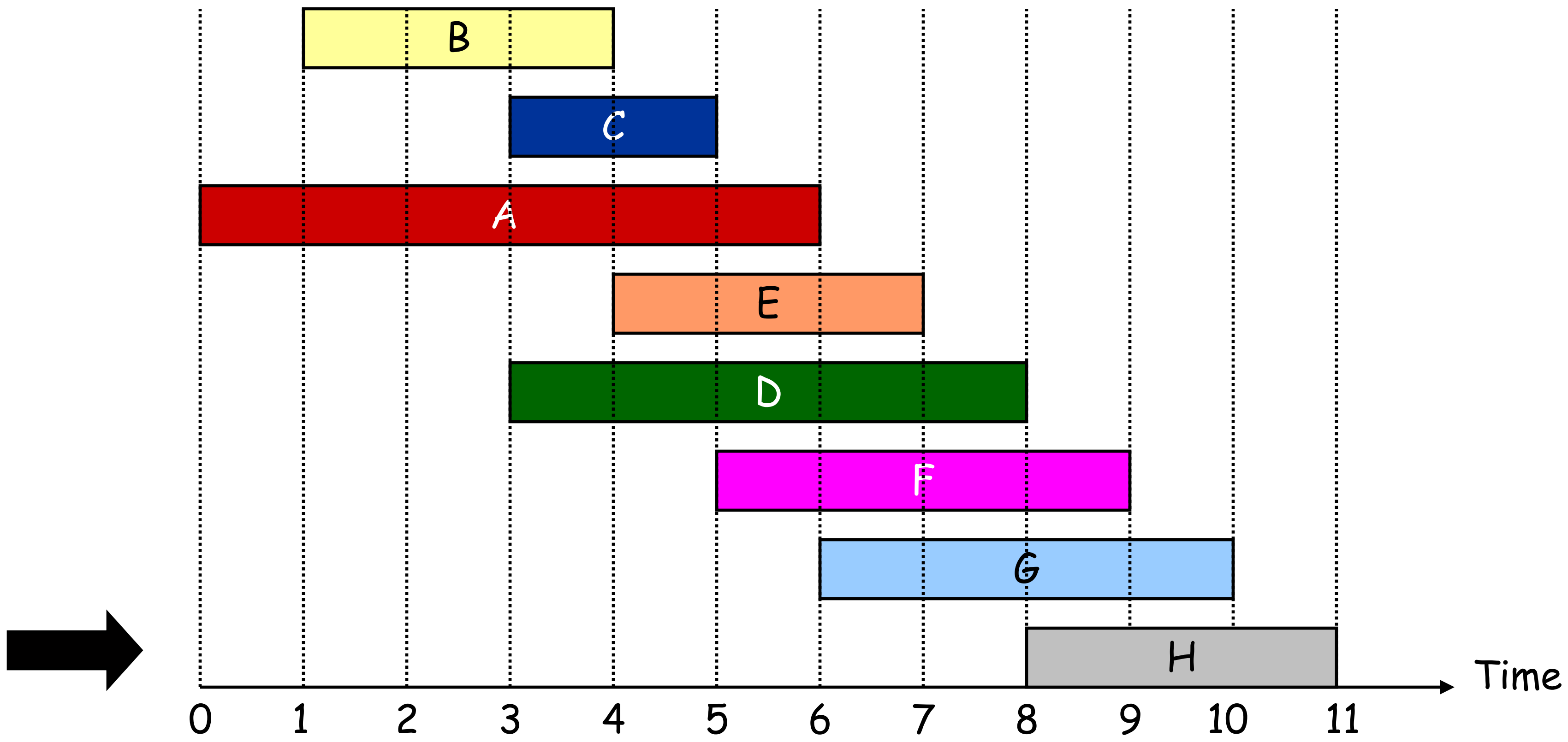


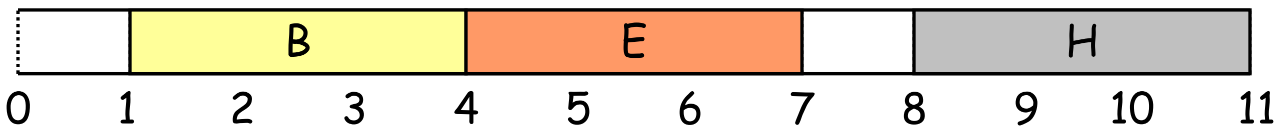
Interval Scheduling





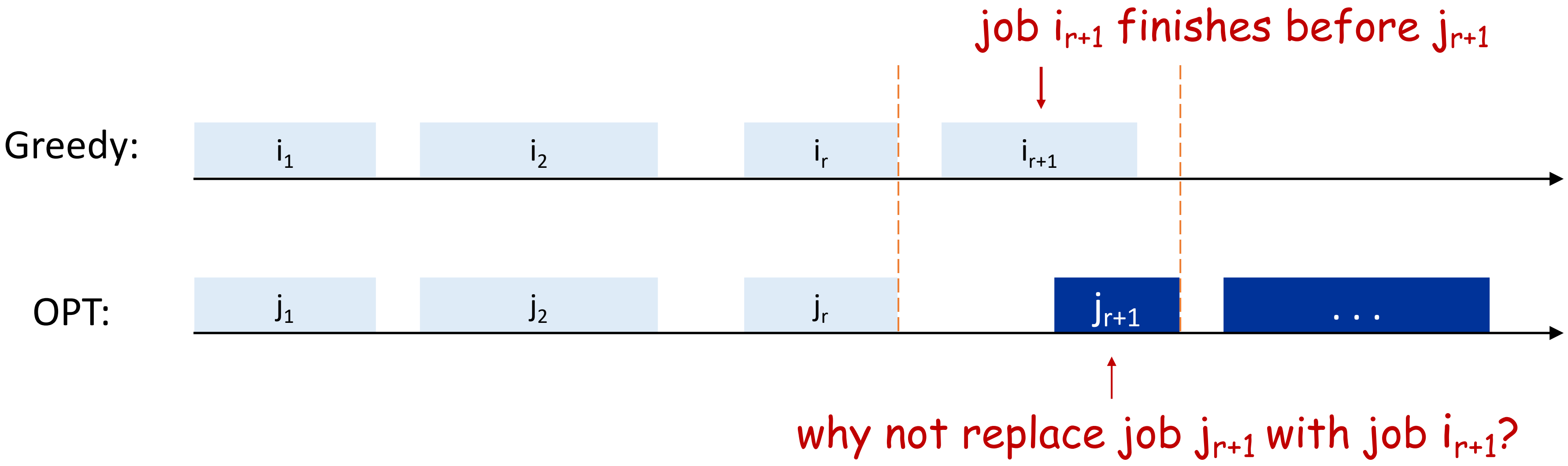
Interval Scheduling





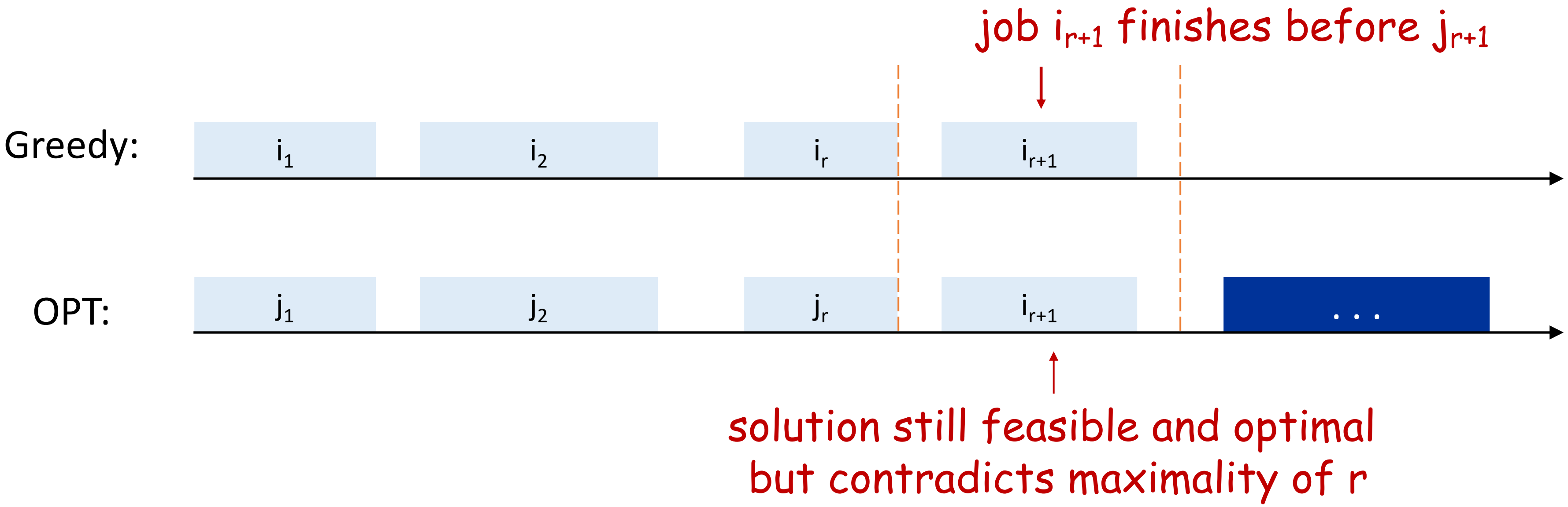
Interval Scheduling: Analysis

- **Theorem.** Greedy algorithm is optimal.
- **Pf.** (by contradiction)
  - Assume greedy algorithm is not optimal, and let's see what happens.
  - Let $\{i_1, i_2, \ldots\, i_n\}$ denote the set of jobs selected by greedy algorithm.
  - Let $\{j_1, j_2, \ldots\, j_m\}$ denote the set of jobs in an optimal solution with the largest possible value of r such that $i_1 = j_1,\ i_2 = j_2,\ \ldots,\ i_r = j_r$.



Interval Scheduling: Analysis

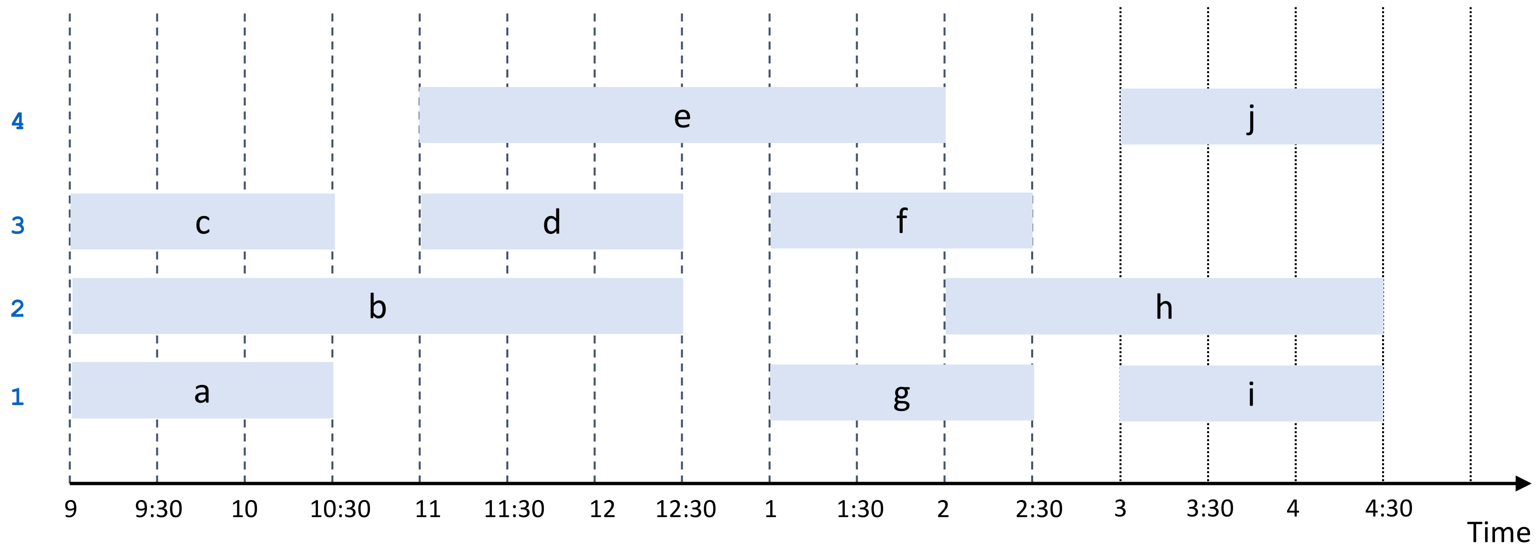
- **Theorem.** Greedy algorithm is optimal.
- **Pf.** (by contradiction)
  - Assume greedy algorithm is not optimal, and let's see what happens.
  - Let $\{i_1, i_2, \ldots i_n\}$ denote the set of jobs selected by greedy algorithm.
  - Let $\{j_1, j_2, \ldots j_m\}$ denote the set of jobs in an optimal solution with the largest possible value of r such that $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$.



2. Interval Partitioning

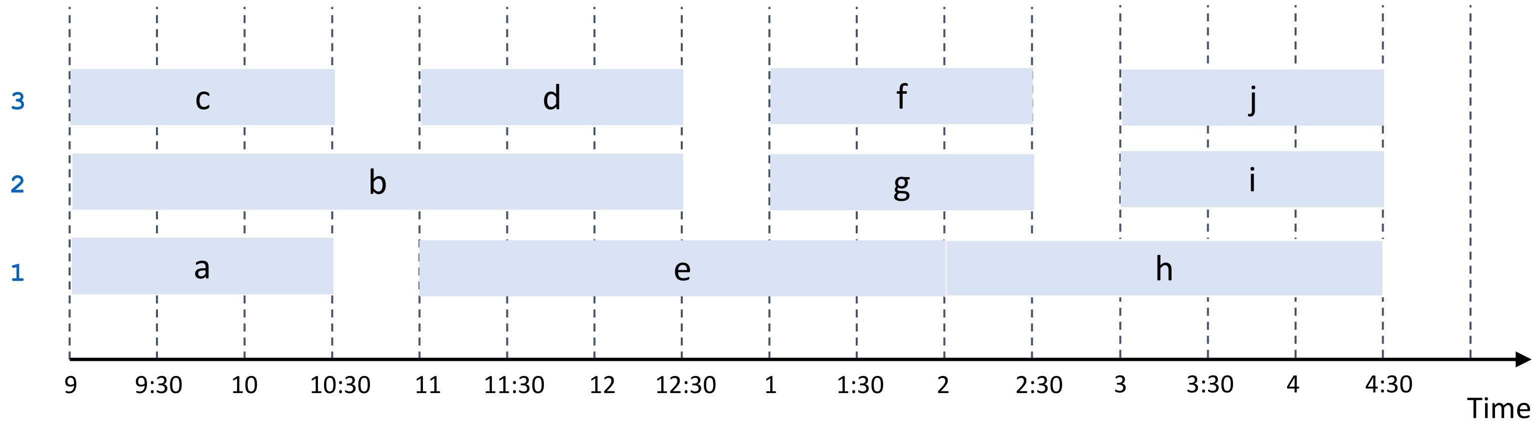
Interval Partitioning

- **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.
- **Example:** A schedule that 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.
- **Example:** A schedule that uses only 3 classrooms.

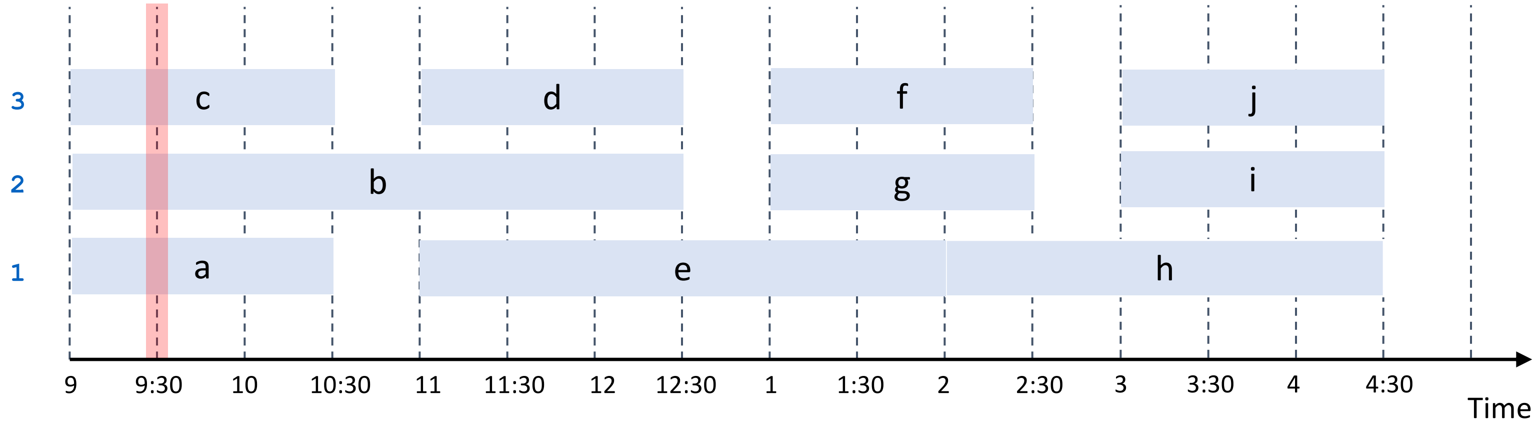


Interval Partitioning: Lower Bound

- **Def.** The depth of a set of open intervals is the maximum number of intervals that contain any given time point.
- **Key observation.** Number of classrooms needed ≥ depth.
- **Example:** Depth of schedule below = 3 -> schedule below is optimal.

e.g., 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 each lecture to any compatible classroom.

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
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
}
```

- **Time complexity.** *O(n log n)*
  - For each classroom, maintain the finish time of the last lecture added.
  - Keep the classrooms in a priority queue keyed by the above finish time.

Interval Partitioning: Greedy Analysis

- **Observation.** Greedy algorithm never schedules two incompatible lectures in the same classroom.
- **Theorem.** Greedy algorithm is optimal.
- **Pf.** Let d = number of classrooms that the greedy algorithm allocates.
  - Classroom d is opened because the greedy algorithm needed to schedule a lecture, say j, that is incompatible with all $d - 1$ other classrooms.
  - The $d - 1$ last lectures in those $d - 1$ classrooms each finish after s_j.
  - Since the greedy algorithm sorted lectures by starting time, all those $d - 1$ incompatible lectures start no later than s_j.
  - Thus, we have d lectures overlapping at time $s_j + \varepsilon$ (i.e., right after s_j).
  - Key observation: all schedules must use $\geq d$ classrooms. ▪

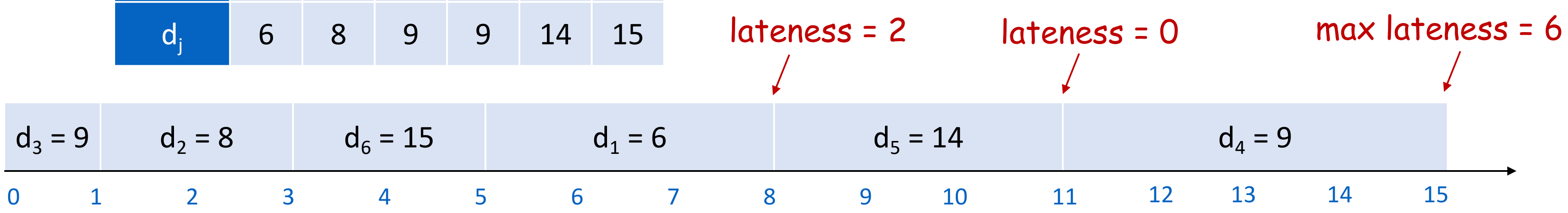
3. Scheduling to Minimize Lateness

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 job 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 $L = max\ \ell_j$

- **Example:**

	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. Assign them one by one.
  - [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

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

	1	2
t_j	1	10
d_j	2	10

counterexample

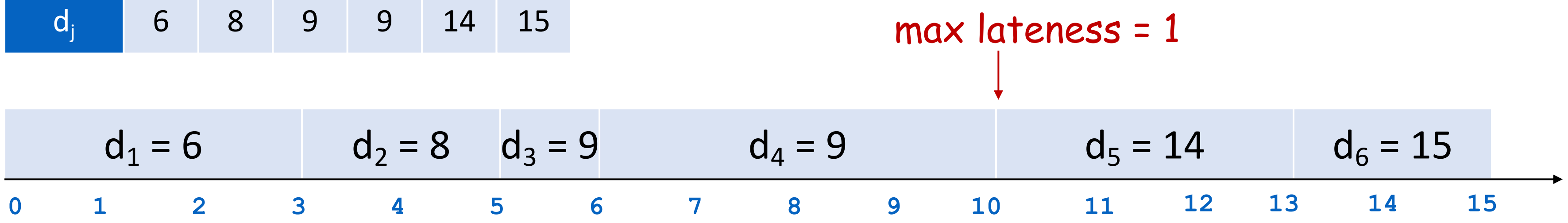
Minimizing Lateness: Greedy Algorithm

- **Greedy algorithm.** Earliest deadline first.

```
Sort n jobs by deadline so that d₁ ≤ d₂ ≤ ... ≤ dn
t ← 0        ← current start time
for j = 1 to n
   Assign job j to interval [t, t + tj]
   sj ← t, fj ← t + tj
   t ← t + tj
output intervals [sj, fj]
```

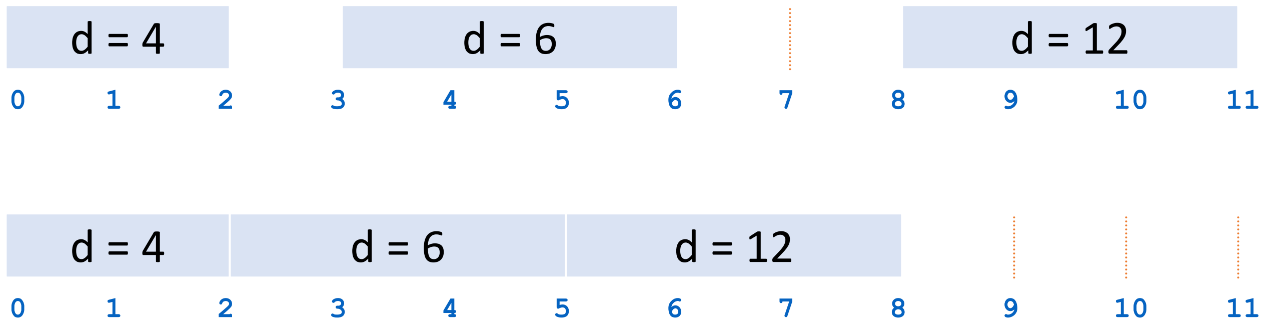
- **Ex:**

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



Minimizing Lateness: No Idle Time

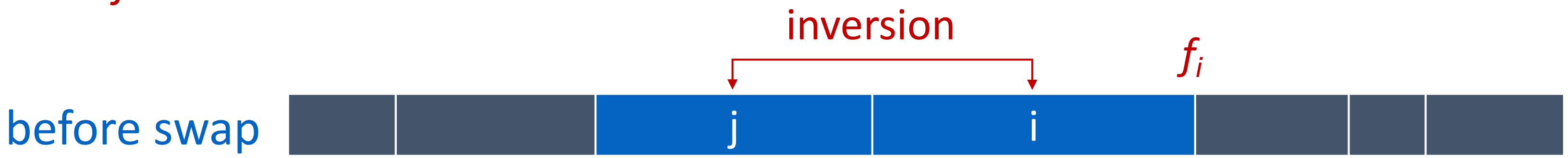
- **Observation.** There exists an optimal schedule with no idle time.



- **Observation.** The greedy schedule has no idle time.

Minimizing Lateness: Inversions

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

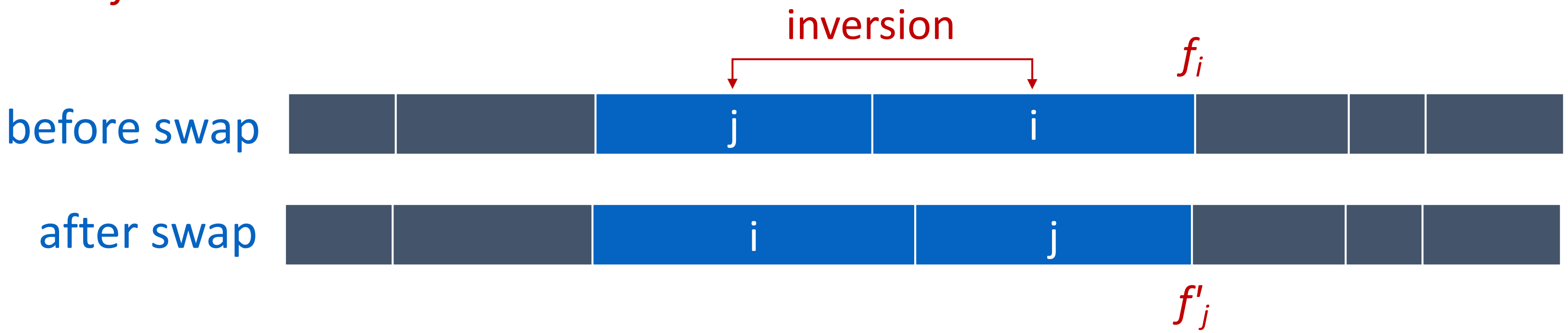


[as before, we assume jobs are numbered such that $d_1 \le d_2 \le \ldots \le d_n$]

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

Minimizing Lateness: Inversions

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



- **Claim.** Swapping two consecutive inverted jobs reduces the number of inversions by one and does not increase the maximum lateness.

- **Pf.** Let ℓ be the lateness before the swap, and let ℓ' be it afterwards.
  - $\ell'_k = \ell_k$ for all $k \neq i, j$
  - $\ell'_i \leq \ell_i$
  - $\ell'_j = \max\{0, f'_j - d_j\} = \max\{0, f_i - d_j\} \leq \max\{0, f_i - d_i\} = \ell_i$

Minimizing Lateness: Greedy Analysis

- **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**.
  - If *S** has an inversion, let job pair *(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 *S**. ■

Greedy Analysis Strategies

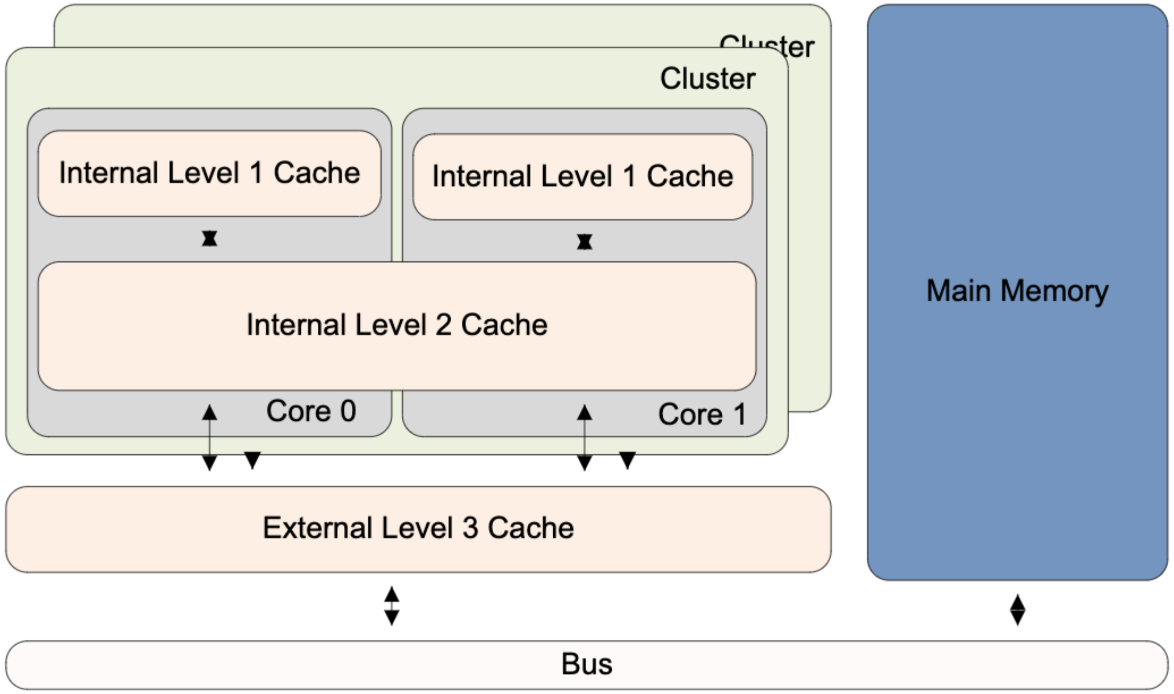
- **Greedy algorithm stays ahead.** Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.
- **Structural.** Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.
- **Exchange argument.** Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.
  - Next we will see a more complex exchange argument: optimal caching
- **Other greedy algorithms.** GS, Dijkstra, *A**, MST algorithms, Chu-Liu, Huffman, etc.

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

4. Optimal Caching

Caching

- HD -> memory -> cache



Memory type	Typical size	Typical access time
Processor registers	128KB	1 cycle
On-chip L1 cache	32KB	1-2 cycle(s)
On-chip L2 cache	128KB	8 cycles
Main memory (L3) dynamic RAM	MB or GB[1]	30-42 cycles
Back-up memory (hard disk) (L4)	MB or GB	> 500 cycles

[1] Size limited by the processor core addressing, for example a 32-bit processor without memory management can directly address 4GB of memory.

https://developer.arm.com/documentation/den0024/a/Caches

Optimal Offline Caching

- **Caching.**
  - Cache with capacity to store k items.
  - Sequence of m item requests $d_1, d_2, \ldots, d_m$. ← offline: prior knowledge
  - 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 evictions.

- **Example:** $k = 2$, initial cache = *ab*, requests: *a, b, <u>c</u>, b, c, <u>a</u>, a, b*.
  - Optimal eviction schedule: *2* evictions.

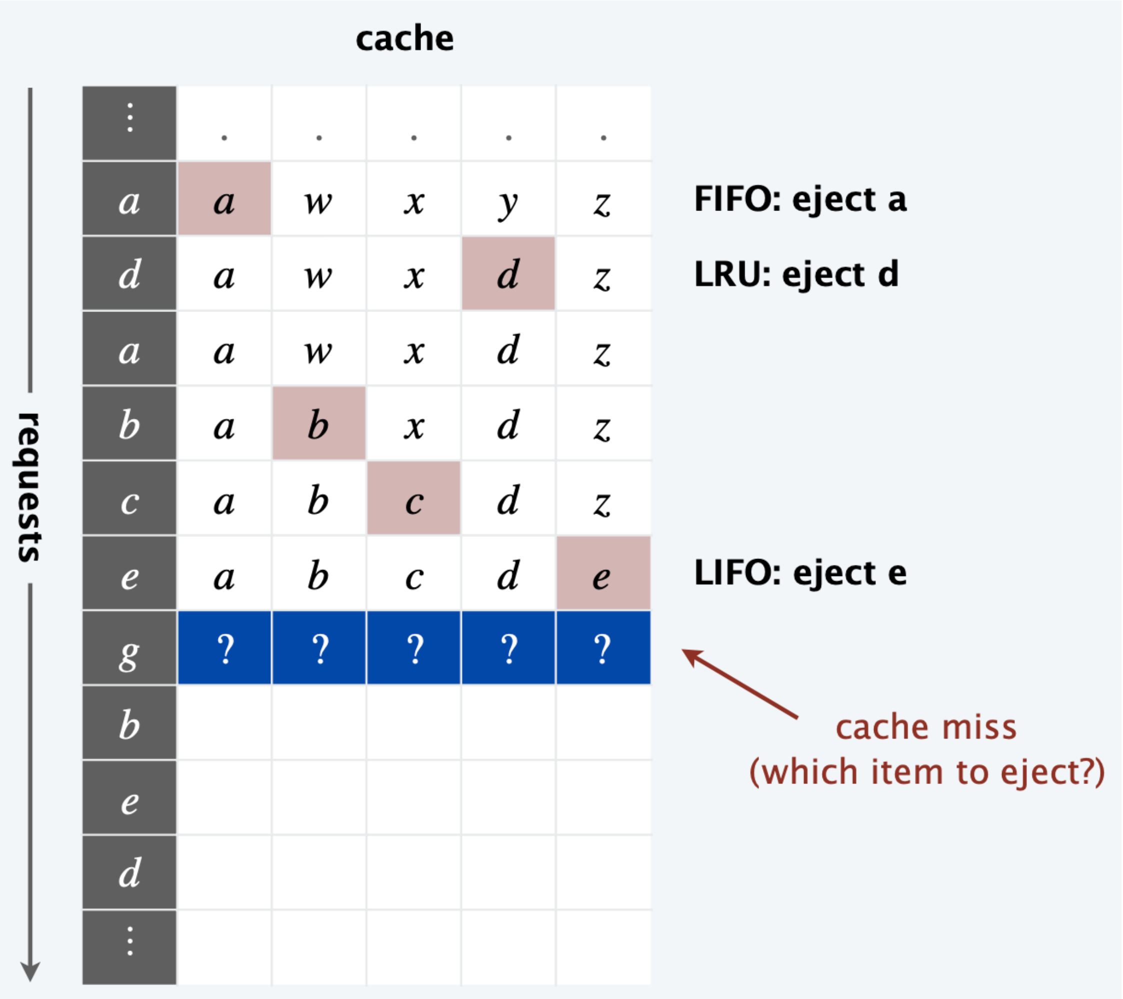
Optimal Offline Caching: Greedy Algorithms

- **Greedy algorithms:**
  - LIFO (last-in-first-out): Evict item brought in most recently.
  - FIFO (first-in-first-out): Evict item brought in least recently.
  - LRU (least-recently-used): Evict item whose most recent access was earliest.
  - LFU (least-frequently-used): Evict item that was least frequently requested.

Optimal Offline Caching: Greedy Algorithms

- **Greedy algorithms:**
  - LIFO (last-in-first-out)
  - FIFO (first-in-first-out)
  - LRU (least-recently-used)
  - LFU (least-frequently-used)

- **Q.** Which one is optimal for offline caching?
- **A.** None of above is optimal. One should somehow utilize prior knowledge of future requests.



Optimal Offline Caching: Farthest-In-Future

- **Farthest-in-Future (FF).** Evict item in the cache that is not requested until farthest in the future.

current cache:

a	b	c	d	e	f

future queries: `g a b c e d a b b a c d e a f a d e f g h ...`

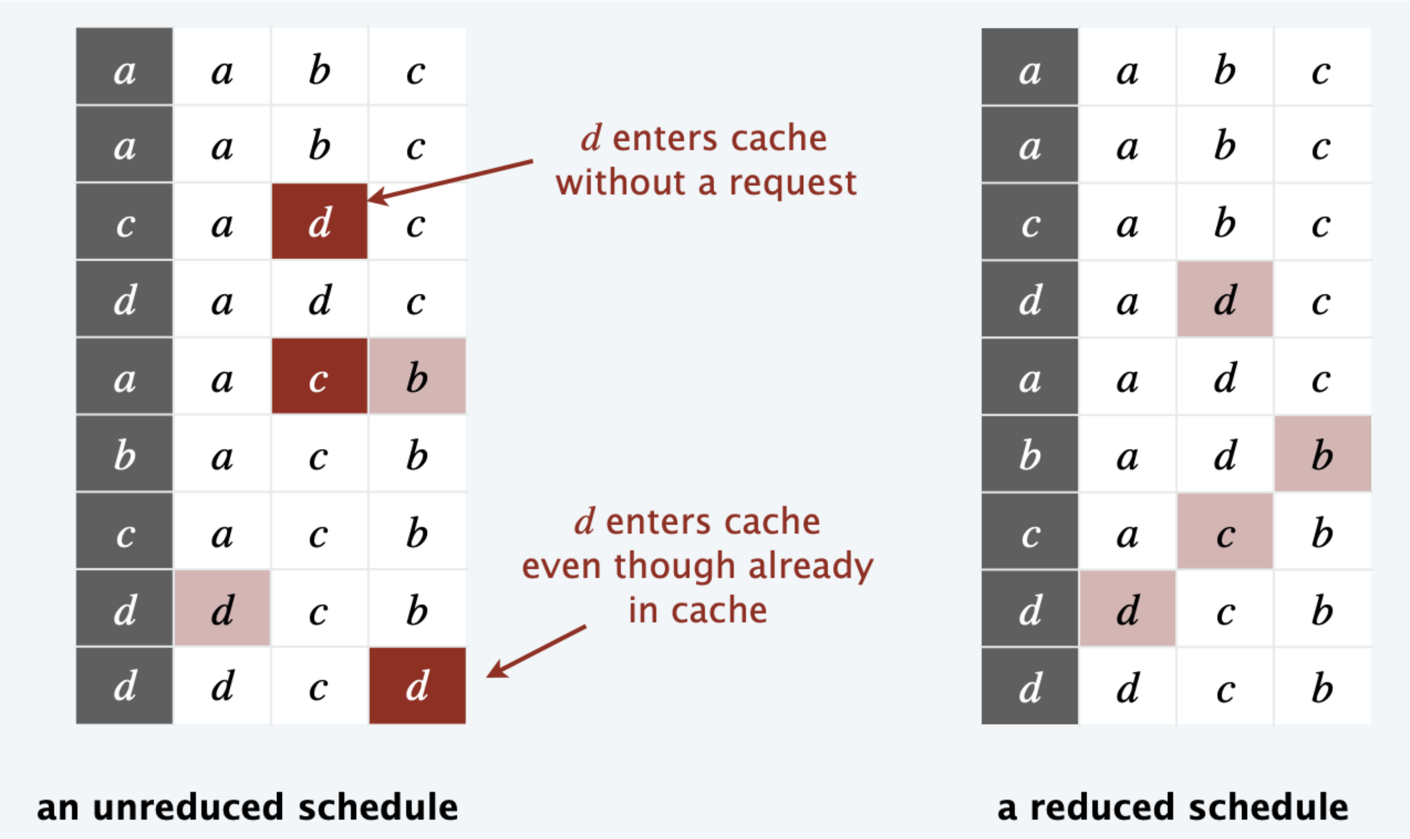
cache miss (at g)

eject this one (at f)

- **Theorem.** [Bellady, 1960s] FF is optimal eviction schedule.
  - Algorithm and theorem are intuitive, but proof is subtle (shown later).

Reduced Eviction Schedules

- **Def.** A reduced schedule is a schedule that only inserts an item into the cache in a step when that item is requested and not yet in cache.

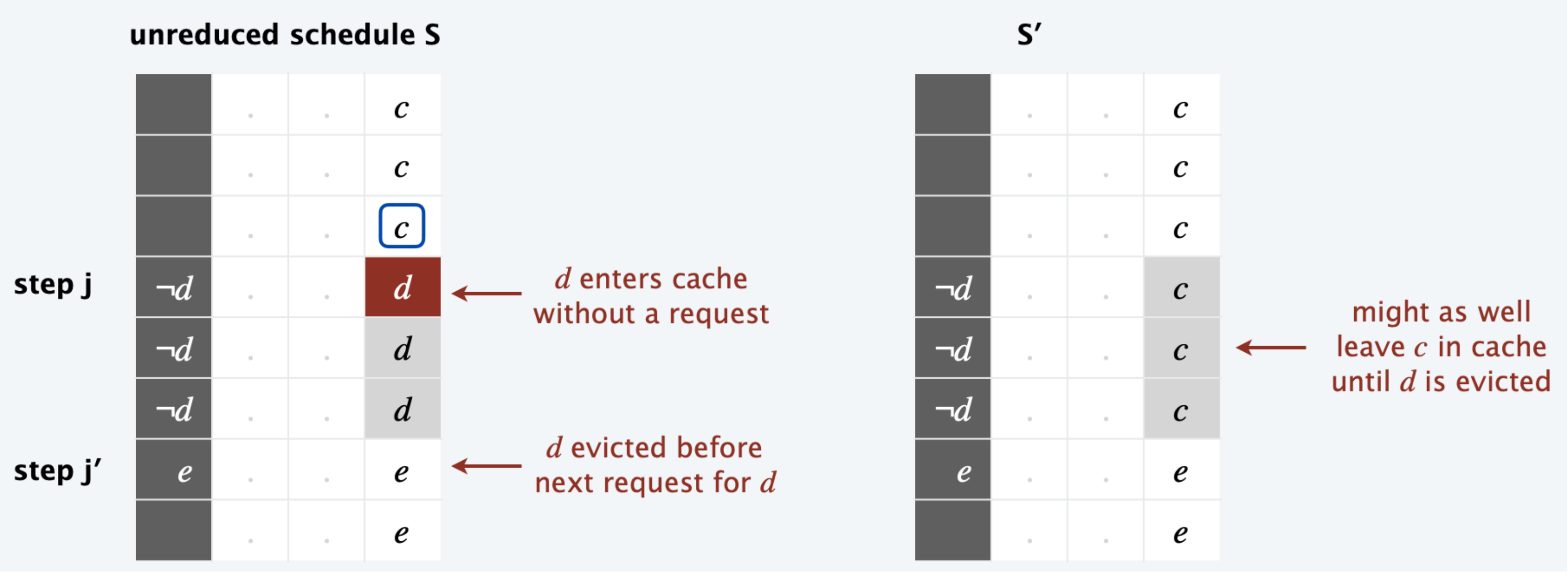


Reduced Eviction Schedules

- **Claim.** Given any unreduced schedule *S*, can transform it into a reduced schedule *S'* with no more evictions.
- **Pf.** (by induction on number of steps *j*) [proof sketch]
  - **Basis Step:** $j = 0$
  - **Inductive Step:**
    - ✓Case 1: *S* brings *d* into the cache in step *j* without a request.
    - ✓Case 2: *S* brings *d* into the cache in step *j* even though *d* is in cache.
  - If there are multiple unreduced items in step *j*, apply each one in turn.
  - Deal with Case 1 before Case 2, since resolving Case 1 might trigger Case 2 ▪

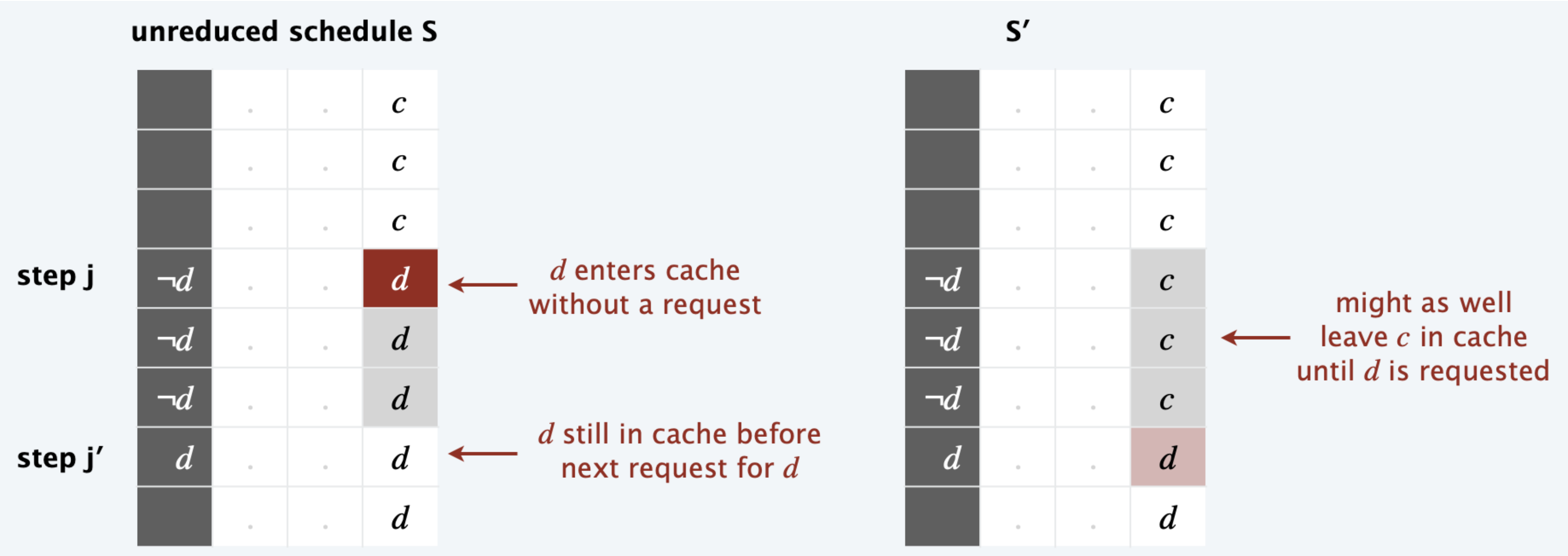
Reduced Eviction Schedules

- **Claim.** Given any unreduced schedule *S*, can transform it into a reduced schedule *S'* with no more evictions.
- **Pf.** (by induction on number of steps *j*) [*d* is inserted in step *j*]
  - Let *c* be the item *S* evicts when it brings *d* into the cache.
  - Case 1a: *d* evicted before next request for *d*.



Reduced Eviction Schedules

- **Claim.** Given any unreduced schedule S, can transform it into a reduced schedule S' with no more evictions.
- **Pf.** (by induction on number of steps j) [d is inserted in step j]
  - Let c be the item S evicts when it brings d into the cache.
  - Case 1b: next request for d occurs before d is evicted.



Reduced Eviction Schedules

- **Claim.** Given any unreduced schedule *S*, can transform it into a reduced schedule *S'* with no more evictions.
- **Pf.** (by induction on number of steps *j*) [*d* is inserted in step *j*]
  - Let *c* be the item *S* evicts when it brings *d* into the cache.
  - Case 2a: *d* evicted before it is needed.

unreduced schedule S

		d_1	a	c	
		d_1	a	c	
		d_1	a	$\boxed{c}$	
step j	d	d_1	a	d_3	← d_3 enters cache even though d_1 is already in cache
	d	d_1	a	d_3	← d_3 not needed
	c	c	a	d_3	
step j'	b	c	a	b	← d_3 evicted
	d	c	a	d_3	← d_3 needed

S'

	d_1	a	c	
	d_1	a	c	
	d_1	a	c	
d	d_1	a	c	← might as well leave c in cache until d_3 in evicted
d	d_1	a	c	
c	c	a	c	
b	c	a	b	
d	c	a	d_3	

Reduced Eviction Schedules

- **Claim.** Given any unreduced schedule *S*, can transform it into a reduced schedule *S'* with no more evictions.
- **Pf.** (by induction on number of steps *j*) [*d* is inserted in step *j*]
  - Let *c* be the item *S* evicts when it brings *d* into the cache.
  - Case 2b: *d* needed before it is evicted.

unreduced schedule S

		d_1	a	c	
		d_1	a	c	
		d_1	a	$\boxed{c}$	
step j	d	d_1	a	d_3	← d_3 enters cache even though d_1 is already in cache
	d	d_1	a	d_3	← d_3 not needed
	c	c	a	d_3	
	a	c	a	d_3	
step j'	d	c	a	d_3	← d_3 needed

S'

	d_1	a	c	
	d_1	a	c	
	d_1	a	c	
d	d_1	a	c	← might as well leave c in cache until d_3 in needed
d	d_1	a	c	
c	c	a	c	
a	c	a	c	
d	c	a	d_3	

Farthest-in-Future: Greedy Analysis

- **Theorem.** Farthest-in-Future (FF) is an optimal eviction algorithm.
- **Pf.** Follows directly from the following invariant.
- **Invariant.** There exists an optimal reduced schedule S that has the same eviction schedule as S_{FF} for any given sequence of request steps.
- **Pf.** (by induction on number of steps j)
  - **Basis Step**: $j = 0$.
  - **Inductive Step:** Let S be reduced schedule that satisfies invariant through j steps. We produce S' that satisfies invariant after $j + 1$ steps.
    - ✓ Let d denote the item requested in step $j + 1$.
    - ✓ Inductive hypothesis: since S and S_{FF} have agreed up until the first j steps, they have the same cache contents before step $j + 1$.

Farthest-In-Future: Greedy Analysis

- **Theorem.** Farthest-in-Future (FF) is an optimal eviction algorithm.
- **Pf.** Follows directly from the following invariant.
- **Invariant.** There exists an optimal reduced schedule S that has the same eviction schedule as S_{FF} for any given sequence of request steps.
- **Pf.** (by induction on number of steps j) [d is requested in step $j + 1$]
  - **Inductive Step:** Let S be reduced schedule that satisfies invariant through j steps. We produce S' that satisfies invariant after $j + 1$ steps.
  - Case 1: d is already in the cache.
    ✓ $S' = S$ satisfies invariant.
  - Case 2: d is not in the cache but S and S_{FF} evict the same item.
    ✓ $S' = S$ satisfies invariant.

Farthest-In-Future: Greedy Analysis

- **Pf.** (continued)
  - Case 3: d is not in the cache; S_{FF} evicts e; S evicts $f \neq e$.
    - ✓Begin construction of S' from S by evicting e instead of f.

j	same	e	f	same	e	f
	S			S'		
j + 1	same	e	d	same	d	f
	S			S'		

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

must involve e or f or both

    - ✓Let S' behave the same as S until S' is forced to take a different action, because (Case 3a/3b) either e or f is requested or because (Case 3c) S evicts e.

Farthest-In-Future: Greedy Analysis

- **Pf.** (continued) [Let j' be the first step after $j + 1$ that S and S' must take different actions, and let g be the item requested at step j'.]
  - Case 3: d is not in the cache; S_{FF} evicts e; S evicts $f \neq e$; S' evicts e.

j'	same	e		same	f
	S			S'	

  - Case 3a: $g = e$
    - ✓Can't happen with FF since there must be a request for f before e.
  - Case 3b: $g = f$

S' agrees with S_{FF} until step j + 1 and S' evicts e in step j + 1

    - ✓Item f is not 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 $e' \neq e$, S' evicts e' and brings e into cache; now S and S' have the same cache.
    - ✓Let S' behave exactly like S for the remaining requests.

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: Greedy Analysis

- **Pf.** (continued) [Let j' be the first step after $j + 1$ that S and S' must take different actions, and let g be the item requested at step j'.]
  - Case 3: d is not in the cache; S_{FF} evicts e; S evicts $f \neq e$; S' evicts e.

j'	same	e		same	f
	S			S'

  - Case 3c: S evicts e (and $g \neq e, f$) at step j'.
    - ✓Make S' evict f .

j'	same	g		same	g
	S			S'

    - ✓Now S and S' have the same cache.
    - ✓Let S' behave exactly like S for the remaining requests. ▪

Optimal Caching: Closing Remarks

- **Online caching vs. offline caching:**
  - 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 computer science.
- **LRU.** Evict item whose most recent access was earliest.
  - LRU is usually used for online caching.

FF with direction of time reversed!

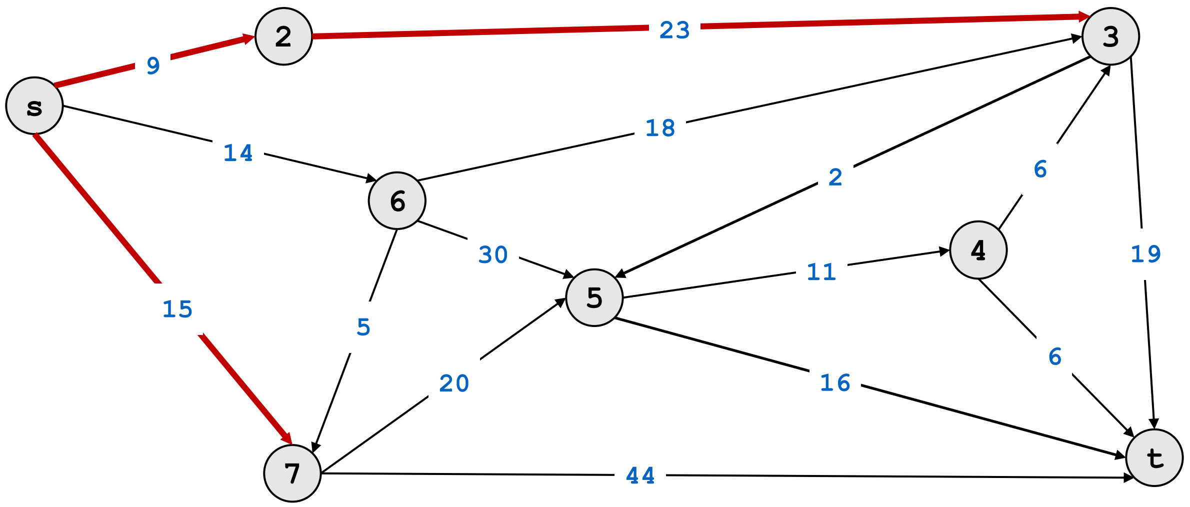
- **Theorem.** FF is optimal offline eviction algorithm.
  - Provides basis for understanding and analyzing online algorithms.
  - LIFO can be arbitrarily bad.
  - LRU is k-competitive, i.e., for any sequence of requests R, $\text{LRU}(R) \leq k\ \text{FF}(R) + k$.
  - Randomized marking is $O(\log k)$-competitive.

out of scope of this course
see section 13.8 of textbook

5. Shortest Paths in a Graph

Single-Source Shortest Path Problem

- **Single-source shortest path problem.**
  - Directed graph $G = (V, E)$ with non-negative edge costs.
  - Source: s undirected graphs can also be viewed as directed graphs
  - ℓ_e = length of edge e path length = sum of edge lengths on path
  - Goal: find a shortest directed path from s to every node.



shortest parth from s to 3: 9 + 23 = 32
shortest parth from s to 7: 15

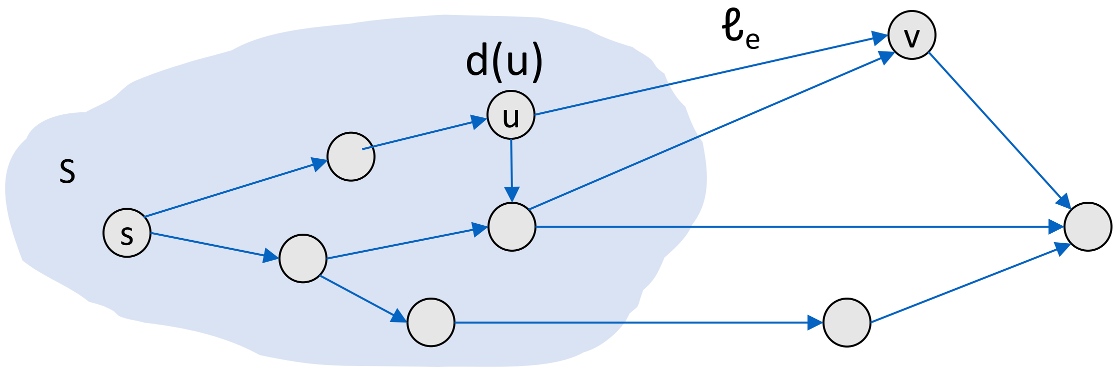
Dijkstra's Algorithm

- **Greedy approach:**
  - Maintain a set *S* of explored nodes 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

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

← shortest s-v path via some u in S followed by a single edge (u, v)

  - Add *v* to *S* and set *d(v) = π(v)*. ← v is added to S only once



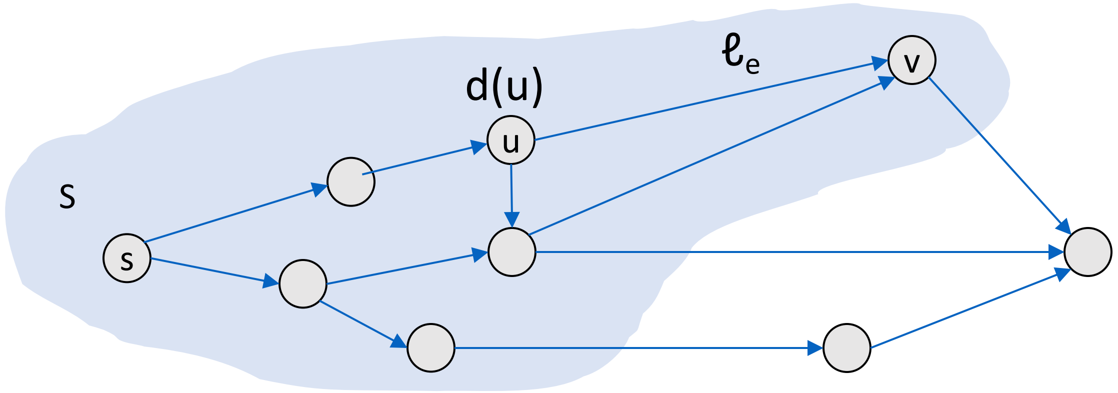
Dijkstra's Algorithm

- **Greedy approach:**
  - Maintain a set *S* of explored nodes 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

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

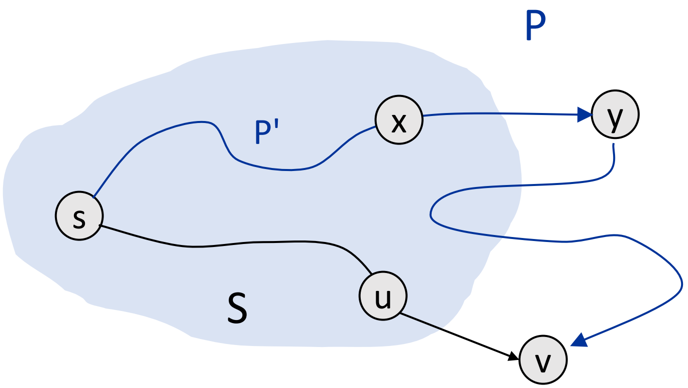
shortest s-v path via some u in S followed by a single edge (u, v)

  - Add *v* to *S* and set *d(v)* = *π(v)*. ← v is added to S only once



Dijkstra's Algorithm: Proof of Correctness

- **Invariant.** For each $u \in S$, $d(u)$ 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 $|S| \geq 1$.
  - Let v be next node added to S and let u-v be the chosen edge.
  - The shortest s-u path plus (u, v) is an s-v path of length $\pi(v)$.
  - Consider any s-v path P. It is no shorter than $\pi(v)$.
    - ✓ Let x-y be the first edge in P that leaves S, and let P' be the sub-path to x.
    - ✓ P is already too long as soon as it leaves S.



$$\underbrace{\ell(P) \geq}_{\text{nonnegative weights}} \ell(P') + \ell(x,y) \underbrace{\geq}_{\text{inductive hypothesis}} d(x) + \ell(x, y) \underbrace{\geq}_{\text{definition of } \pi(y)} \pi(y) \underbrace{\geq}_{\pi(v) \text{ is min of all } \pi(\cdot)} \pi(v)$$

Dijkstra's Algorithm: Implementation

- Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$:
  - Next node v to explore: node with minimum $\pi(v)$. ← find-min
  - When exploring v, for each incident edge $e = (v, w)$, add w to the queue if it has not been added before and update
    insert

$$\pi(w) = \min \{ \pi(w),\ \pi(v) + \ell_e \}.$$ ← decrease-key

  - Remove v from queue. ← v will not be added back to queue again

    delete-min

PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fibonacci heap†
insert	n	n	log n	$d \log_d n$	1
delete-min	n	n	log n	$d \log_d n$	log n
decrease-key	m	1	log n	$\log_d n$	1
find-min	n	n	1	1	1
Total		n^2	m log n	$m \log_{m/n} n$	m + n log n

† Individual ops are amortized bounds

Dijkstra's Algorithm: History

" What's the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. "

--- Edsger W. Dijkstra

More about Edsger W. Dijkstra

Dijkstra was immensely influential in many fields of computing: compilers, operating systems, concurrent programming, software engineering, programming languages, algorithm design, and teaching (among others!)

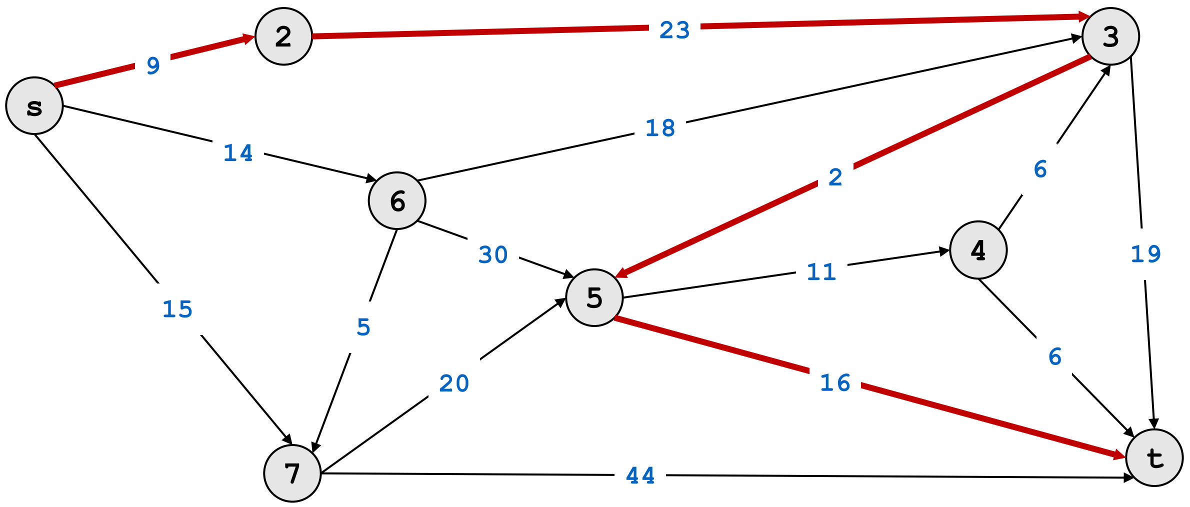
It would be hard to pin down what he is most famous for because he has influenced so much CS.

Dijkstra was also influential in making programming more structured -- he wrote a seminal paper titled, "Goto Considered Harmful" where he lambasted the idea of the "goto" statement (which exists in C++ -- you will rarely, if ever, use it!)

Single-Pair Shortest Path Problem

- **Single-pair shortest path problem.**
  - Directed graph $G = (V, E)$ with non-negative edge costs.
  - Source s, destination t.
  - ℓ_e = length of edge e.
  - Goal: find a shortest directed path from s to t.

only one destination is considered



shortest path from s to t:
9 + 23 + 2 + 1 = 50

Single-Pair Shortest Path Problem

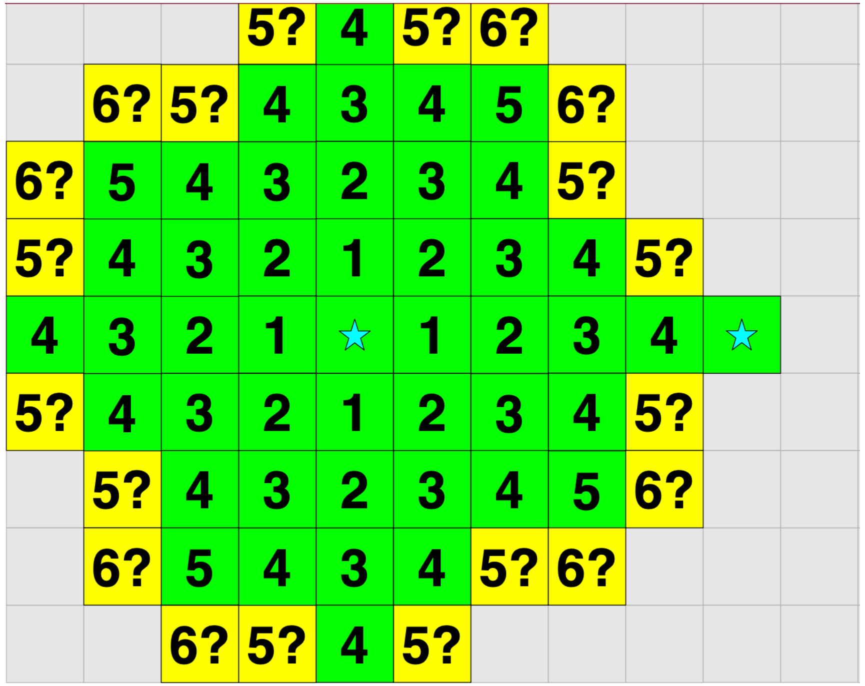
- **Single-pair shortest path problem.**
  - Directed graph $G = (V, E)$ with non-negative edge costs.
  - Source s, destination t.
  - ℓ_e = length of edge e.
  - Goal: find a shortest directed path from s to t.

only one destination is considered

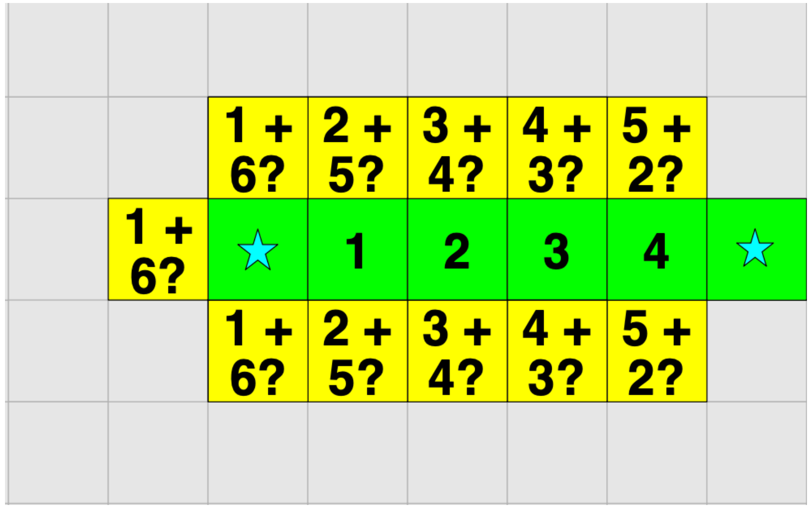
- **Q.** Can we beat Dijkstra when we have only one destination?
  - e.g., if traveling from Beijing to Shanghai we will go south.
- **A.** add some heuristic information to guide the search
  - e.g., direction (in the case of a street map)

Single-Pair Shortest Path Problem

- **Dijkstra priority** *π(v)*:
  - *s-v* distance



- **Ideal priority** *p(v)*:
  - *π(v)* + exact *v-t* distance *d(v, t)*



we should prioritize search to the right

$A*$ Search Algorithm

- **$A*$ priority $f(v)$:**
  - s-v distance $\pi(v)$ (same as Dijkstra) + heuristic (estimated) v-t distance $h(v, t)$
- **Greedy approach:**
  - Maintain a set Q of open nodes for which we would like to explore.
  - Initialize $Q = \{s\}, \pi(s) = 0, f(s) = h(s, t)$.
  - Repeatedly choose node v that minimizes $f(v)$.

    v may be added back to Q again later
  - If $v = t$, returns shortest s-t path and $\pi(v)$; otherwise, remove v from Q and explore v — for each incident edge $e = (v, w)$, do the following if $\pi(v) + \ell_e < \pi(w)$:

    ✓ $\pi(w) \leftarrow \pi(v) + \ell_e$

    ✓ $f(w) \leftarrow \pi(w) + h(w, t)$

    ✓ If w is not in Q, add w to Q. ← w could be added to Q multiple times

*A** Search Algorithm: Demo

- **Dijkstra:**

8	7	6	5	4	5	6	7	8	9?				
7	6	5	4	3	4	5	6	7	8	9?			
6	5	4	3	2	3	4	5	6	7	8	9?		
5	4	3	2	1	2	3		7	8	9?			
4	3	2	1	☆	1	2		8	☆				
5	4	3	2	1	2	3		7	8	9?			
6	5	4	3	2	3	4	5	6	7	8	9?		
7	6	5	4	3	4	5	6	7	8	9?			
8	7	6	5	4	5	6	7	8	9?				

- ***A****: (*h(v, t)* = Hamming distance)

				3 + 8?	4 + 7?	5 + 6?	6 + 5?	7 + 4?					
			3 + 8?	2	3	4	5	6	7 + 2?				
		3 + 8?	2	1	2	3		7 + 2?					
	3 + 8?	2	1	☆	1	2		8	☆				
		3 + 8?	2	1	2	3		7	8 + 1?				
			3 + 8?	2	3	4	5	6	7	8 + 3?			
				3 + 8?	4 + 7?	5 + 6?	6 + 5?	7 + 4?	8 + 3?				

*A** Search Algorithm: Choice of *h(v, t)*

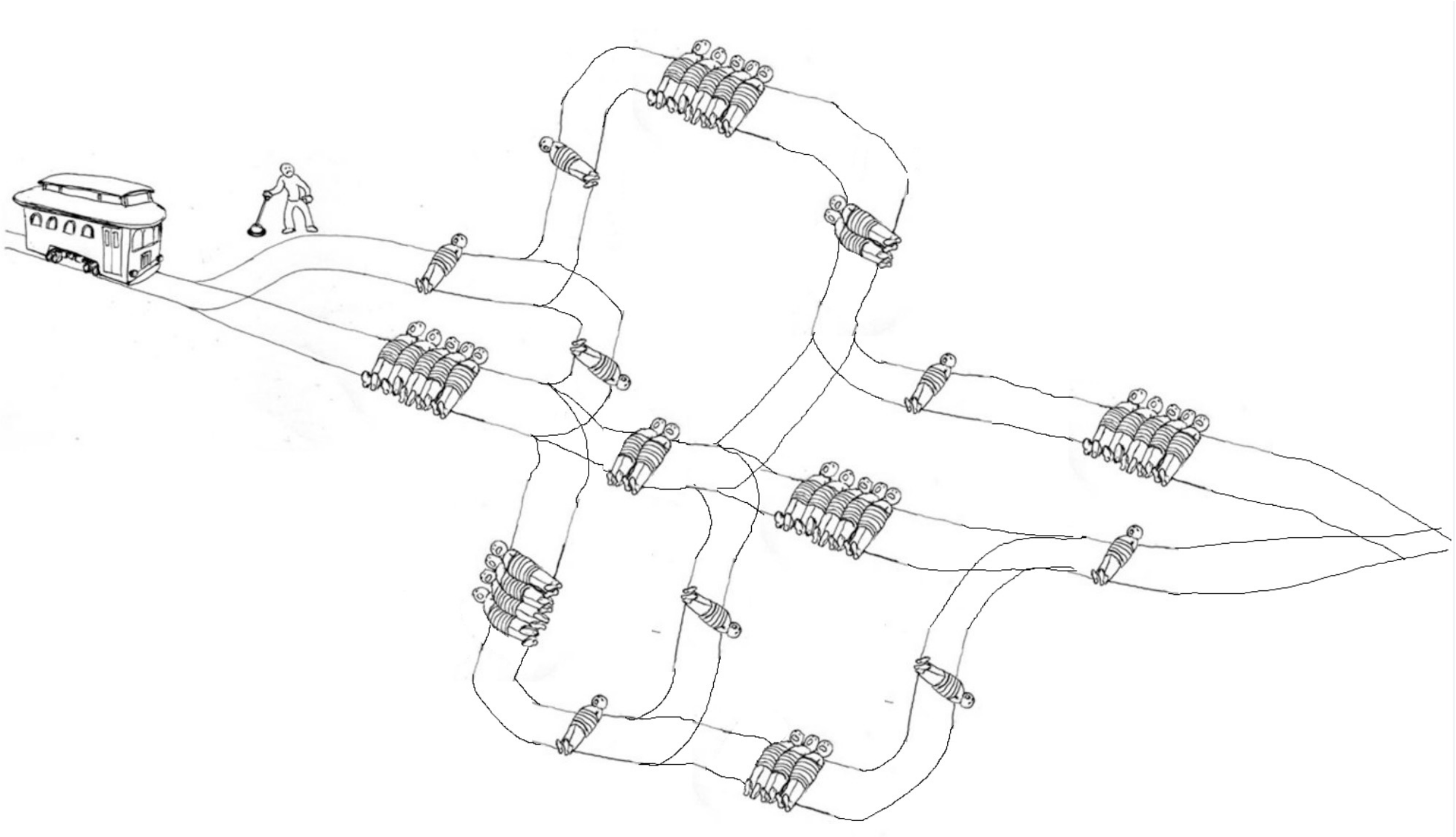
- **Q.** How does *h(v, t)* estimate the true *v-t* distance *d(v, t)*?
- **A.** Usually best-possible distance.
  - ➢ e.g., Hamming distance, Euclidean distance, etc.
- **Choice of** *h(v, t)* ($h(v, t) \geq 0$):
  - ➢ $h(v, t) = 0$: same as Dijkstra
  - ➢ $h(v, t) < d(v, t)$: same or faster (shortest path guaranteed) than Dijkstra
  - ➢ $h(v, t) = d(v, t)$: fastest (shortest path ensured) but requires perfect knowledge
  - ➢ $h(v, t) > d(v, t)$: not necessarily find shortest path (but might run even faster)
- **Takeaway:** Never overestimate the true future cost *d(v, t)* for *A**.
  - ➢ *h(v, t)* is called admissible if $0 \leq h(v, t) \leq d(v, t)$. In this case, a shortest path is guaranteed. (The proof is left as an exercise.)

More on $A*$ Search Algorithm

- **Consistency of** $h(v, t)$**:** for every edge $e = (u, v), h(u, t) \leq \ell_e + h(v, t)$ holds.
  - ➢ If $h(v, t)$ is consistent, $A*$ is guaranteed to find a shortest path without adding any node to Q more than once. (The proof is left as an exercise.)

- **Q.** If we are only interested to know if there is some path with length $\leq L$, can we improve the performance of $A*$?
- **A.** Yes. We can skip exploring v if $f(v) = \pi(v) + h(v, t) > L$.
  - ➢ not necessary to maintain a priority queue since any path of length $\leq L$ suffices
- **Q.** When there are too many nodes in the graph (i.e., n is very large), how can you do $A*$ search within limited memory?
- **A.** Use DFS instead of BFS (e.g., Dijkstra and $A*$ are both BFS-style).

Shortest Path Problem: Moral Implications



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

6. Minimum Spanning Trees

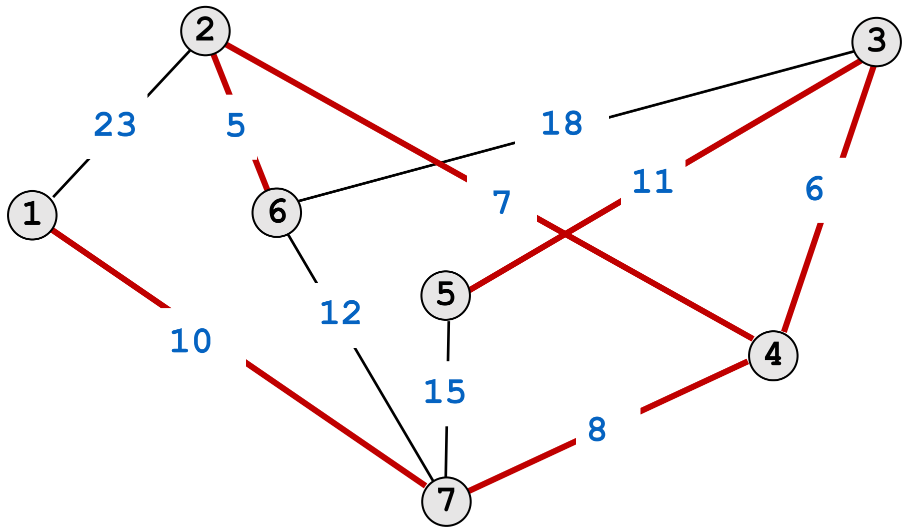
Spanning Trees

- **Def.** Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$. H is a spanning tree of G if H is both acyclic and connected.
- **Property.** All the following statements are equivalent:
  - H is a spanning tree of G.
  - H is acyclic and connected.
  - H is connected and has $|V| - 1$ edges.
  - H is acyclic and has $|V| - 1$ edges.
  - H is minimally connected: removal of any edge disconnects it.
  - H is maximally acyclic: addition of any edge creates a cycle.

Minimum Spanning Trees (MSTs)

- **Def.** Given a connected, undirected graph $G = (V, E)$ with edge costs, a minimum spanning tree (MST) (V, T) is a spanning tree of G such that the sum of the edge costs in T is minimized.
- **Cayley's theorem:** K_n has n^{n-2} spanning trees. $(|V| = n, |E| = m)$

cannot solve by brute-force

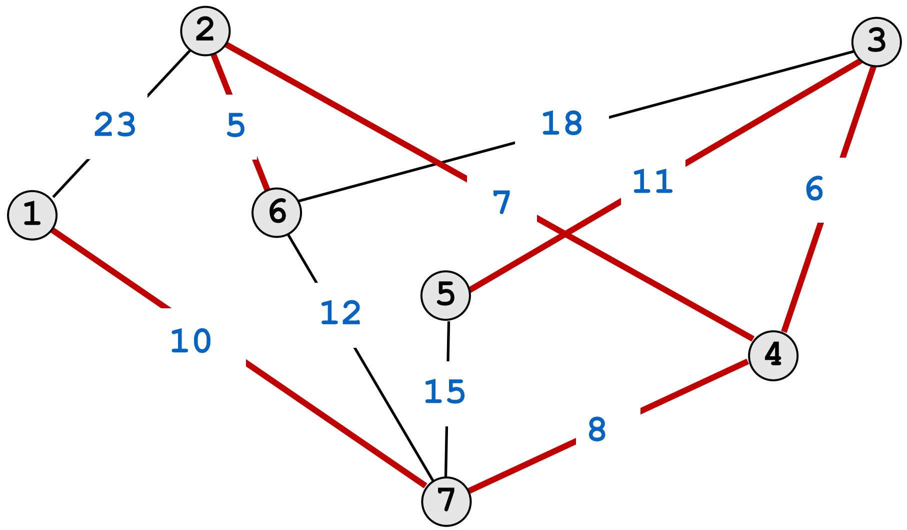


MST cost: 5 + 6 + 7 + 8 + 10 + 11 = 47

Prim's Algorithm

- **Greedy approach:**
  - Initialize $S = \{s\}$ for any node s and initialize edge set $T = \emptyset$.
  - Repeat $n - 1$ times: find-min
    - ✓Add to T a min-cost edge e with exactly one endpoint in S.
    - ✓Add to S the other endpoint of e and update min-cost edge for $V - S$.

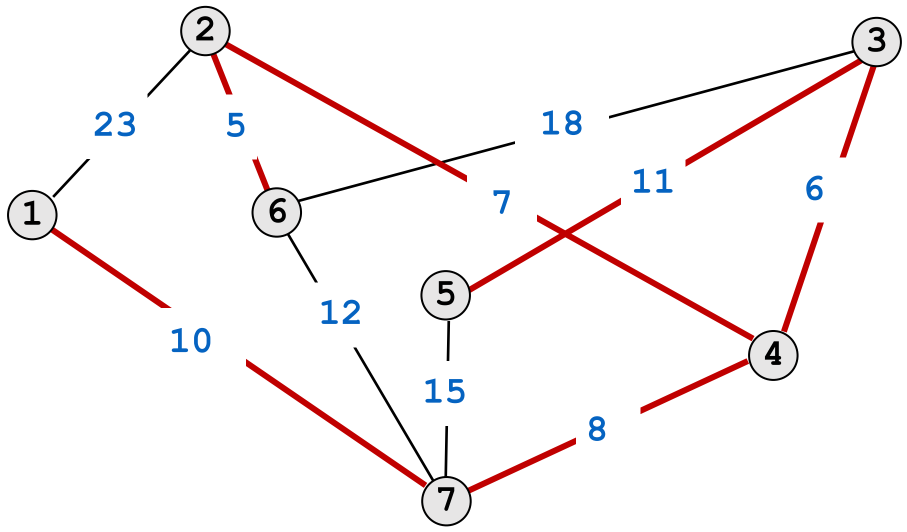
delete-min decrease-key



time complexity: O(m log n)

Kruskal's Algorithm

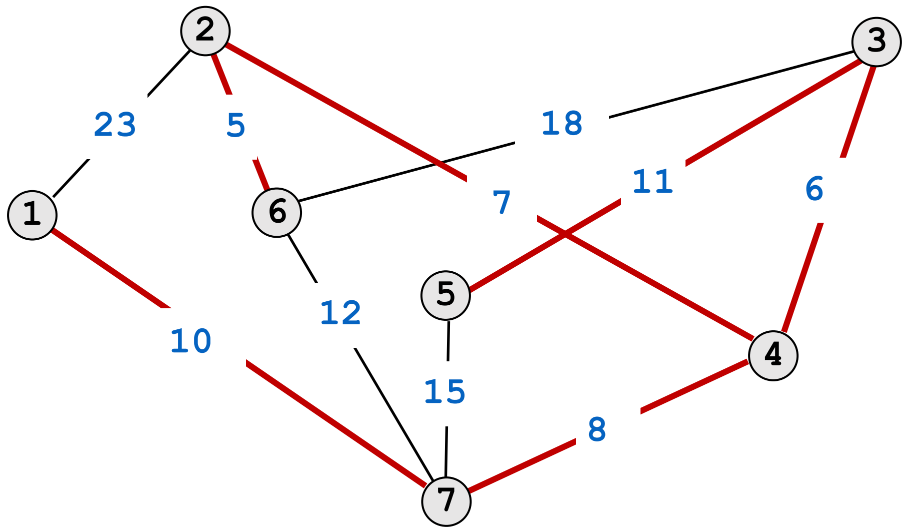
- **Greedy approach:**
  - Initialize edge set $T = \emptyset$.
  - Sort edges in ascending order of cost.
  - Repeat m times:
    - ✓Add to T the considered edge unless it creates a cycle. ← Union Find



time complexity: O(m log m)

Reverse-Delete Algorithm

- **Greedy approach:**
  - Initialize edge set $T = E$.
  - Sort edges in descending order of cost.
  - Repeat m times:
    - ✓Delete from T the considered edge unless it would disconnect T.



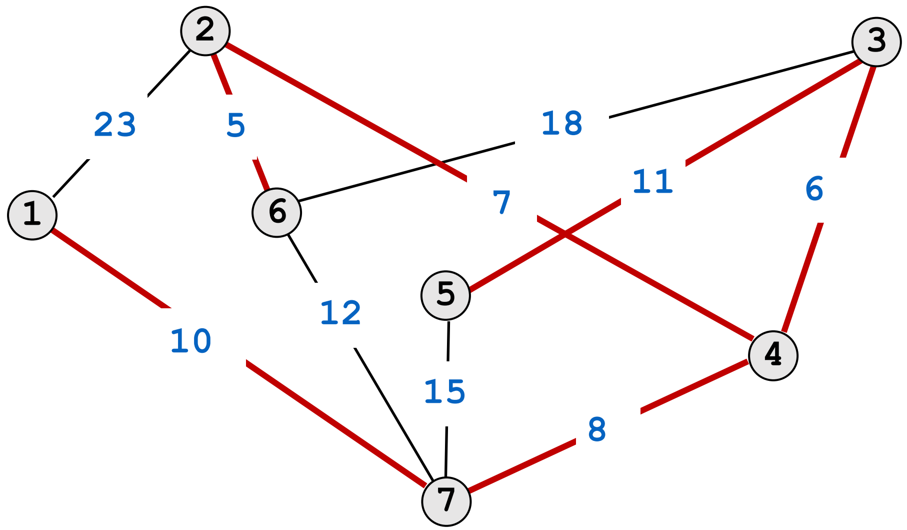
time complexity: $O(m \log n (\log \log n)^3)$

[Thorup 2000]

Borůvka's Algorithm

- **Greedy approach:**
  - Initialize edge set $T = \emptyset$ (T has n connected components, one for each node).
  - Repeat until only one connected component is left: ← O(log n) rounds
    - ✓For each edge (u, v), if u, v are in different components, use the cost of (u, v) to update the min-cost edge for both components. ← no need to sort edges
    - ✓Add to T the min-cost edges for each component.

number of components at least halved



time complexity: O(m log n)

Minimum Spanning Trees: Summary

- **Summary.** The learned MST greedy algorithms follow similar ideas and share roughly the same time complexity $O(m \log n)$.
  - Prim: extend a single connected component with a min-cost edge
  - Kruskal: extend connected components with a min-cost edge
  - Reverse-Delete: remove a max-cost edge and maintain connected
  - Borůvka: extend connected components with a min-cost edge (without sorting)

- **Remark.** All of the above greedy algorithms can be extended to find minimum spanning forests.

- **Q.** Does a linear-time $O(m)$ compare-based MST algorithm exist?

Minimum Spanning Trees: Linear Algorithms

- [Karger-Klein-Tarjan 1995] $O(m)$ randomized MST algorithms do exist!
- It is still open for deterministic compare-based MST algorithms:

year	worst case	discovered by
1975	$O(m \log \log n)$	Yao
1976	$O(m \log \log n)$	Cheriton–Tarjan
1984	$O(m \log^* n)$, $O(m + n \log n)$	Fredman–Tarjan
1986	$O(m \log (\log^* n))$	Gabow–Galil–Spencer–Tarjan
1997	$O(m\, \alpha(n) \log \alpha(n))$	Chazelle
2000	$O(m\, \alpha(n))$	Chazelle
2002	*asymptotically optimal*	Pettie–Ramachandran
20xx	$O(m)$	???

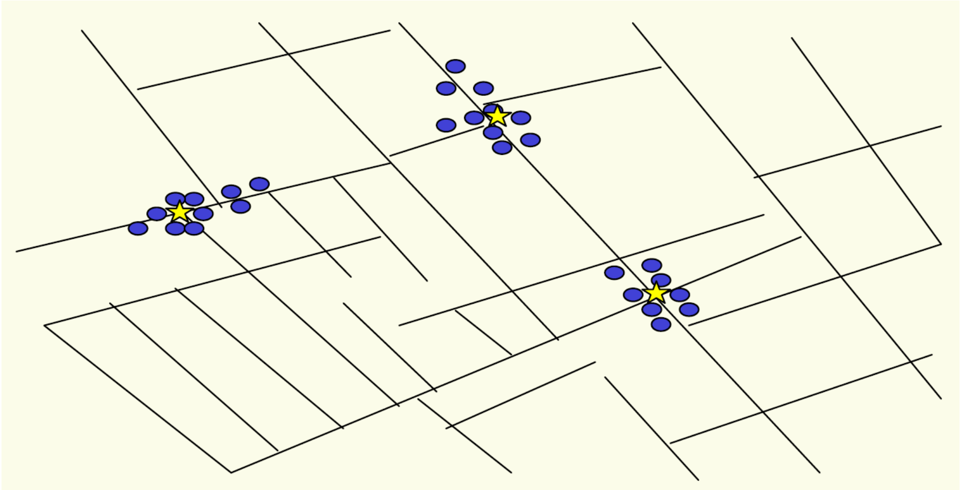
7. Single-Link Clustering

Clustering

- **Goal.** Given a set U of n objects labeled $p_1, \ldots, p_n$, partition into clusters so that objects in different clusters are far apart.

e.g., photos, documents, etc.

w.r.t. some distance, e.g., number of pixels that differ by some threshold

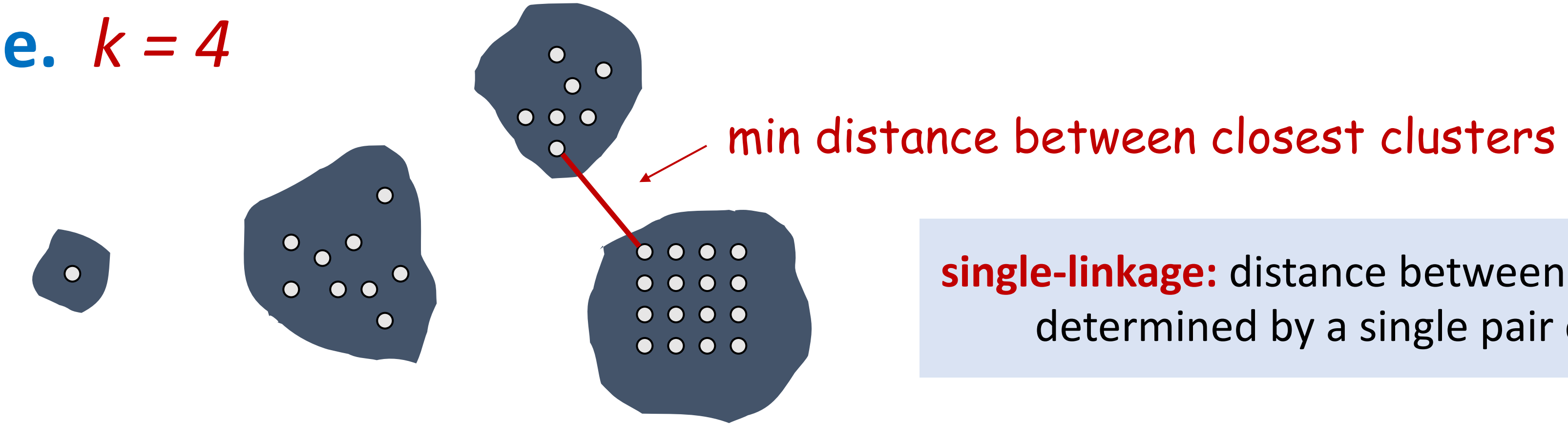


Outbreak of cholera deaths in London in 1850s. Reference: Nina Mishra, HP Labs

- **Applications.** Routing, categorization, similarity searching, etc.

Single-Linkage Clustering of Max Spacing

- ***k*-clustering.** Divide objects into k non-empty groups.
- **Distance function.** Numeric value specifying "closeness" of two objects.
  - $d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernibles)
  - $d(p_i, p_j) \geq 0$ (nonnegativity)
  - $d(p_i, p_j) = d(p_j, p_i)$ (symmetry)
- **Spacing.** Min distance between any pair of points in different clusters.
- **Goal.** Given an integer k, find a k-clustering of maximum spacing.
- **Example.** $k = 4$



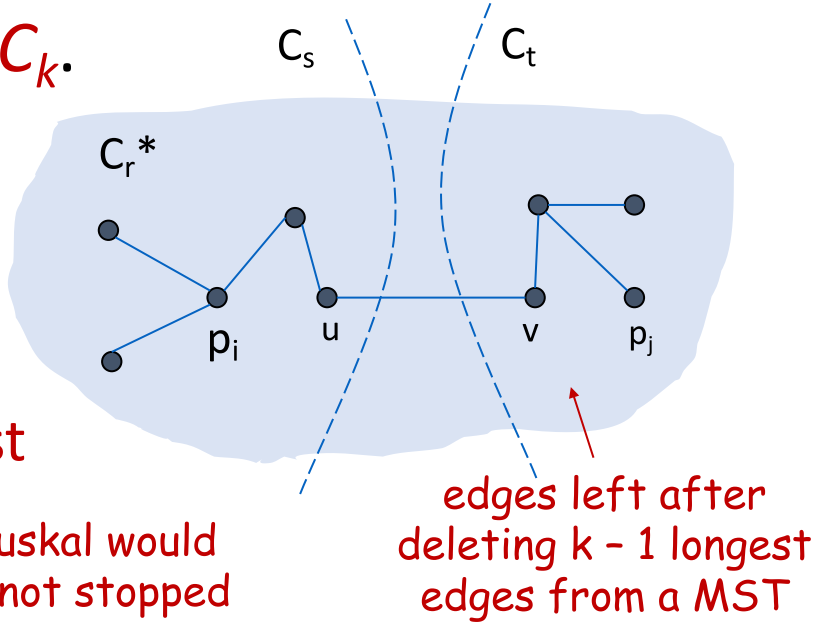
single-linkage: distance between two clusters is determined by a single pair of objects

Single-Link k-Clustering: Greedy Algorithm

- **Greedy approach:**
  - Form a graph on the object set U, with n clusters in the beginning.
  - Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.
  - Repeat $n - k$ times until there are exactly k clusters.
- **Key observation.** This procedure is precisely Kruskal's algorithm, with a complete graph K_n where edge costs are distances (except we stop when there are k connected components).
- **Remark.** Equivalent to finding an MST and deleting the $k - 1$ most expensive edges.

Single-Link *k*-Clustering: Analysis

- **Theorem.** Let C^* be the k-clustering $C_1^*, \ldots, C_k^*$ formed by deleting the $k - 1$ most expensive edges of an MST. C^* is a k-clustering of max spacing.
- **Pf.** Let C denote any other k-clustering $C_1, \ldots, C_k$.
  - Let p_i, p_j be in the same cluster in C^*, say C_r^*, but in different clusters in C, say C_s and C_t.
  - Some edge (u, v) on the $p_i - p_j$ path in C_r^* spans two different clusters in C.
  - Spacing of C^* = length d^* of the $(k - 1)$-st longest edge in the corresponding MST.



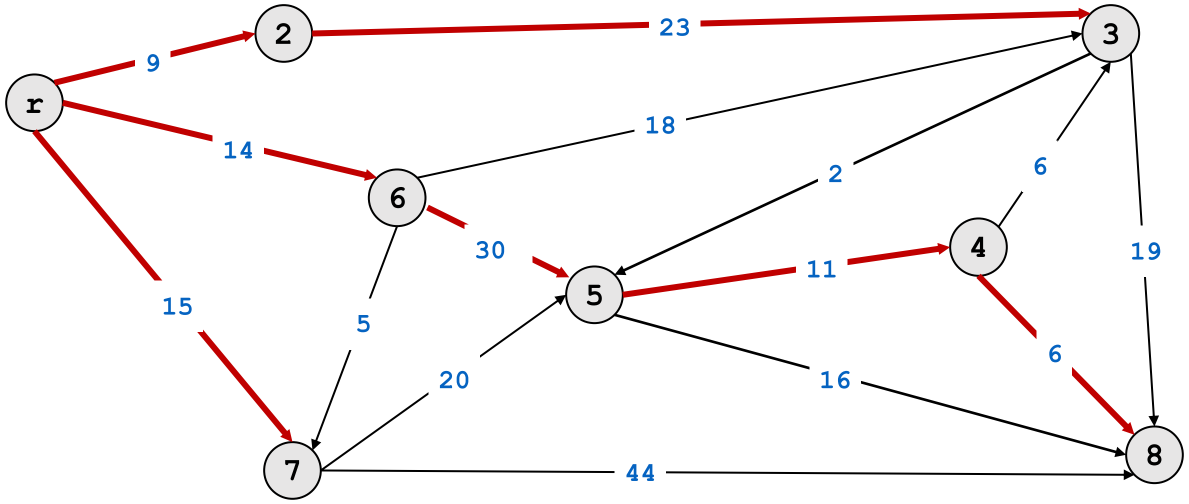
this is the edge Kruskal would have added next if not stopped

  - All edges on the $p_i - p_j$ path have length $\leq d^*$ since Kruskal already added them.
  - Spacing of C is $\leq d^*$ since u and v are in different clusters in C and $d(u, v) \leq d^*$. ▪

8. Min-Cost Arborescences

Arborescences

- **Def.** Given a directed graph $G = (V, E)$ and a root $r \in V$, an arborescence (rooted at r) is a subgraph $T = (V, F)$ such that
  - T is a spanning tree of G if we ignore the direction of edges.
  - There is a (unique) directed path in T from r to each other node $v \in V$.
- **Observation.** Arborescences are essentially directed spanning trees.



Arborescences

- **Claim.** A subgraph T of G is an arborescence rooted at r iff T has no directed cycles and each node $v \neq r$ has exactly one entering edge.
- **Pf.** ("if" + "only if")

  "only if" ⇒:

  ✓An arborescence (directed spanning tree) has no cycles.

  ✓Each node $v \neq r$ has only one entering edge: last edge on the unique r-v path.

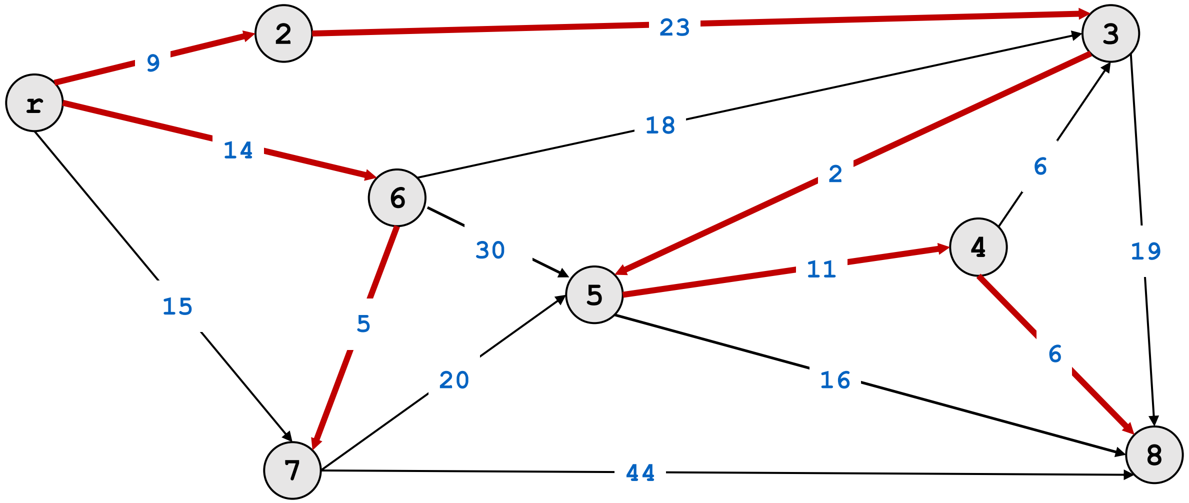
  "if" ⇐:

  ✓To construct an r-v path, start at v and follow edges in the backward direction. Since T has no directed cycles, the process must terminate. It must terminate at r since r is the only node with no entering edge.

  ✓Since each node $v \neq r$ has exactly one entering edge, the above r-v paths are unique for every v and T must be a spanning tree when ignoring direction. ▪

Min-Cost Arborescence Problem

- **Problem.** Given a directed graph *G* with a root node *r* and nonnegative edge costs, find an arborescence rooted at *r* of minimum cost.
- **Observation.** Min-cost arborescences are essentially directed MSTs.
- **Assumptions.** (w.l.o.g.) All nodes are reachable from *r*. No edge enters *r*.



cost of min-cost arborescence:
9 + 14 + 5 + 23 + 2 + 11 + 6 = 70

A Sufficient Optimality Condition

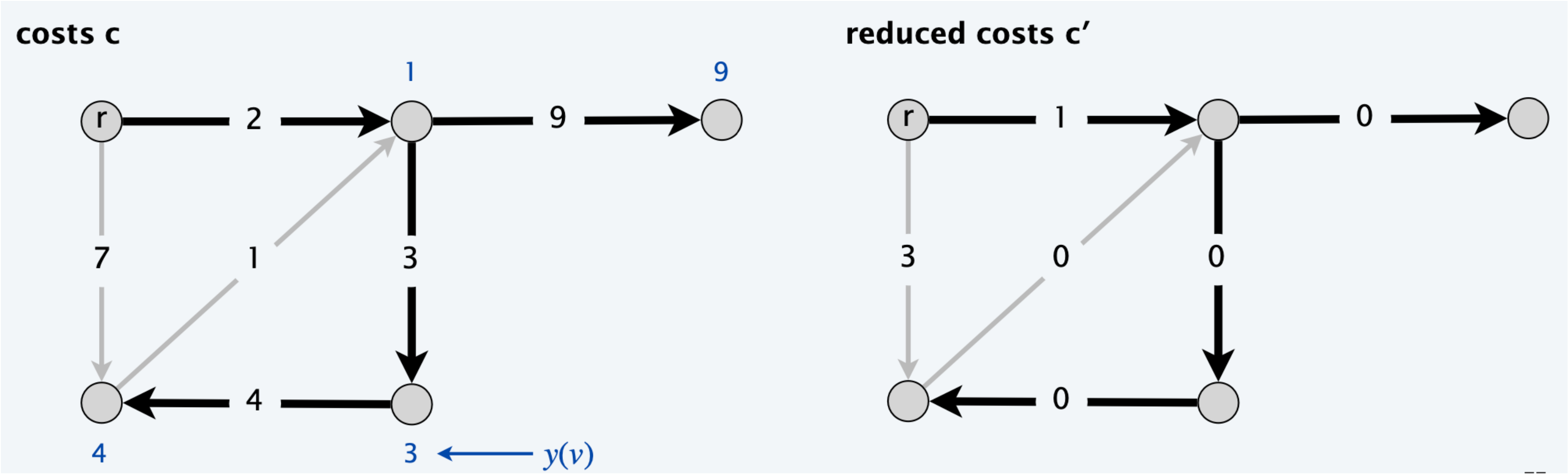
- **Property.** For each node $v \neq r$, choose a cheapest edge entering v. If such $n - 1$ edges form an arborescence, then it is a min-cost arborescence.
- **Pf.** An arborescence needs exactly one edge entering each node $v \neq r$ and the above is the cheapest way to make each of these choices. ▪

- **Q.** What would happen when it is not an arborescence?
- **A.** There are directed cycles.

- **Q.** How can we handle such directed cycles?

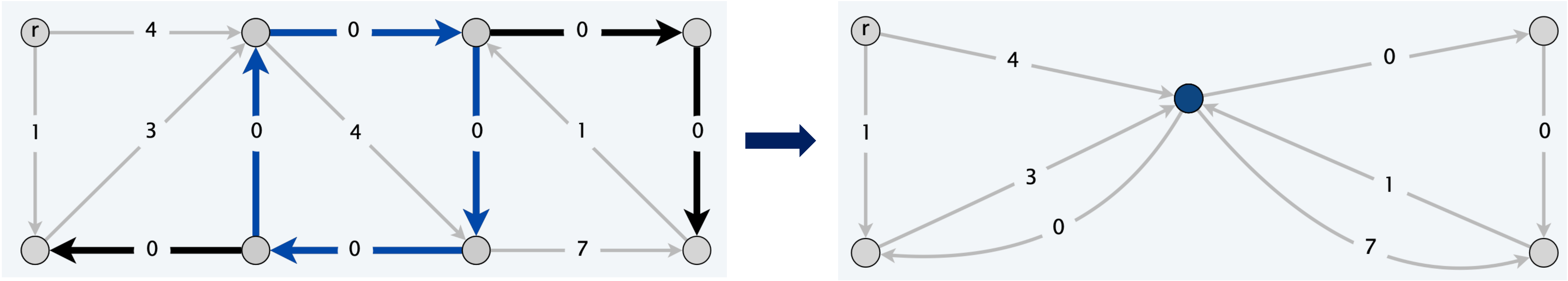
Reduced Costs

- **Def.** For each $v \neq r$, let $y(v)$ denote the min cost of any edge entering v. The reduced cost of an edge (u, v) is $c'(u, v) = c(u, v) - y(v) \geq 0$.
- **Claim.** T is a min-cost arborescence in G using costs c if and only if T is a min-cost arborescence in G using reduced costs c'.
- **Pf.** Recall that any arborescence T has exactly one edge entering $v \neq r$, so the cost difference in c and c' for any T is the same. ▪



Chu-Liu's Algorithm: Intuition

- **Intuition:**
  - For each $v \neq r$, choose a cheapest edge entering v and form an edge set E^*.
  - Then, all edges in E^* have 0 cost with respect to reduced costs $c'(u, v)$.
  - If E^* does not contain a cycle, then we find a min-cost arborescence.
  - If E^* contains a cycle C, can afford to use as many such 0-cost edges in C.
  - Therefore, we can contract C to a supernode (and remove self-loops).
  - Recursively solve problem in contracted graph G' with reduced costs $c'(u, v)$.



Chu-Liu's Algorithm (or Edmonds' Algorithm)

- **Greedy algorithm.** Contract and expand.

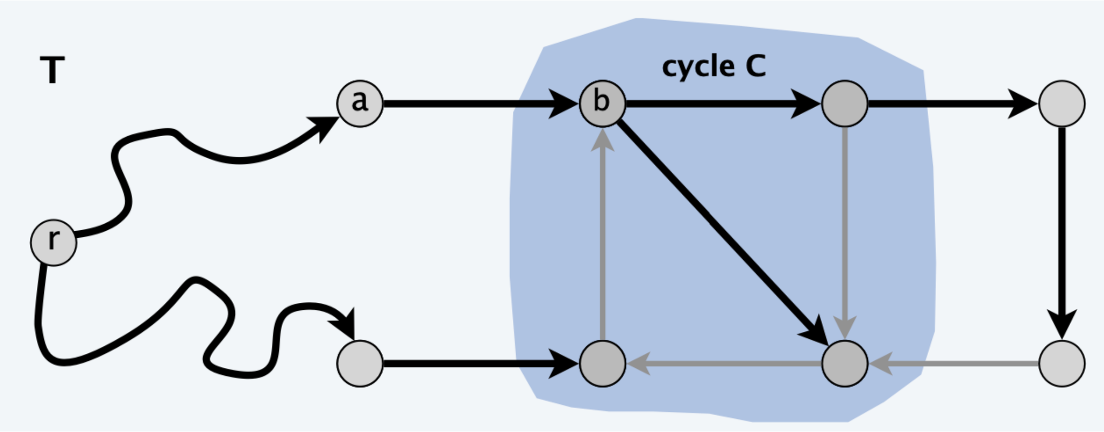
```
Chu-Liu(G, r, c) {
   for each v ≠ r {
      choose one min-cost edge entering v and add it to set E*
      y(v) ← min cost of any edge entering v
      c′(u, v) ← c(u, v) - y(v) for each edge (u, v) entering v
   }
   if (E* forms an arborescence)
      return T = (V, E*)
   else
      C ← directed cycle in E*
      contract C to a single supernode and get G′ = (V′, E′)
      T′ ← Chu-Liu(G′, r, c′)
      expand T′ to an arborescence T in G by adding all but one edge of C
      return T
}
```

one node in C already had an entering edge in T′

Chu-Liu's Algorithm: Greedy Analysis

- **Q.** What could go wrong for Chu-Liu's algorithm?
- **A.** Contracting cycle *C* places extra constraint on arborescence.
  - Min-cost arborescence in *G'* must have exactly one edge entering a node in *C* (since *C* is contracted to a single supernode)
  - But a min-cost arborescence in *G* might have multiple edges entering *C*.

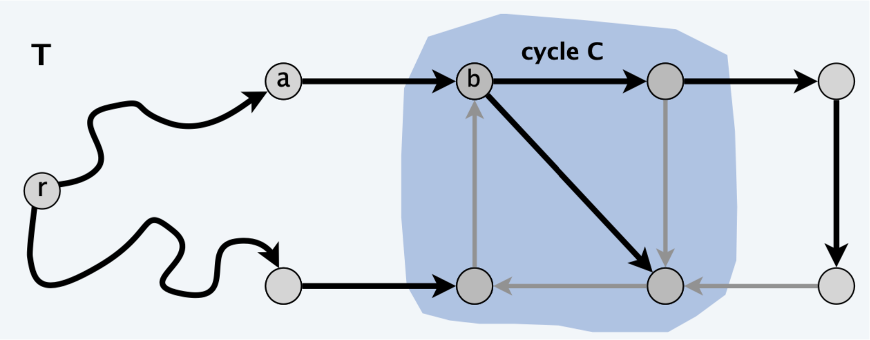
sufficient to prove the existence of a min-cost arborescence in G with only one edge entering C



Chu-Liu's Algorithm: Greedy Analysis

- **Lemma.** Let C be a cycle in G containing only 0-cost edges. There exists a min-cost arborescence T rooted at r with exactly one edge entering C.
- **Pf.** (by cases)
  - Case 0: T has no edges entering C. Impossible!

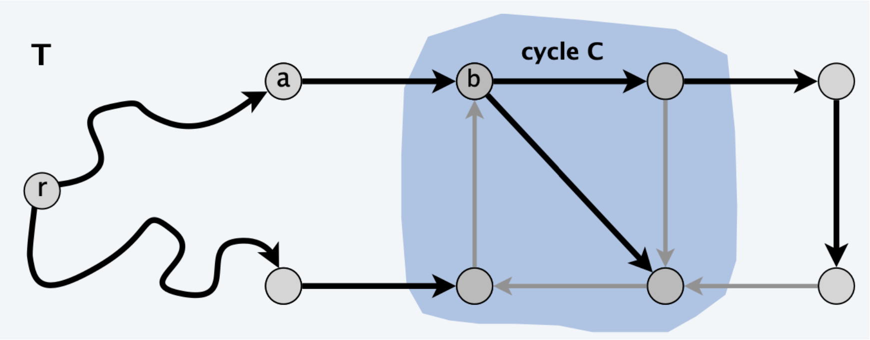
    ✓Arborescence T has an r-v path for each node v ⇒ at least one edge enters C.
  - Case 1: T has exactly one edge entering C. Nothing to prove.
  - Case 2: T has two (or more) edges entering C. We construct another min-cost arborescence T^* that has exactly one edge entering C.

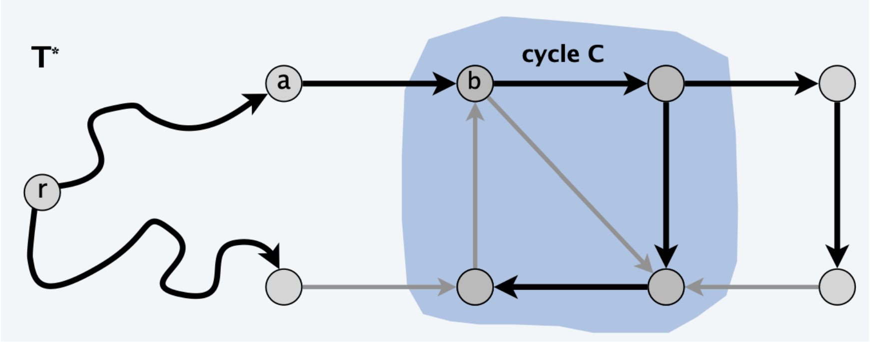


Chu-Liu's Algorithm: Greedy Analysis

- **Lemma.** Let *C* be a cycle in *G* containing only *0*-cost edges. There exists a min-cost arborescence *T* rooted at *r* with exactly one edge entering *C*.
- **Pf.** (by cases)
  - ➢ Case 2. *T* has two (or more) edges entering *C*. We construct another min-cost arborescence *T** that has exactly one edge entering *C*.
    - ✓Let *(a, b)* be an edge in *T* entering *C* that lies on a shortest path from *r*.
    - ✓We delete all edges of *T* that ends at a node in *C* except *(a, b)*.
    - ✓We add in all edges of *C* except the one that enters *b*.

this path uses only one node in C





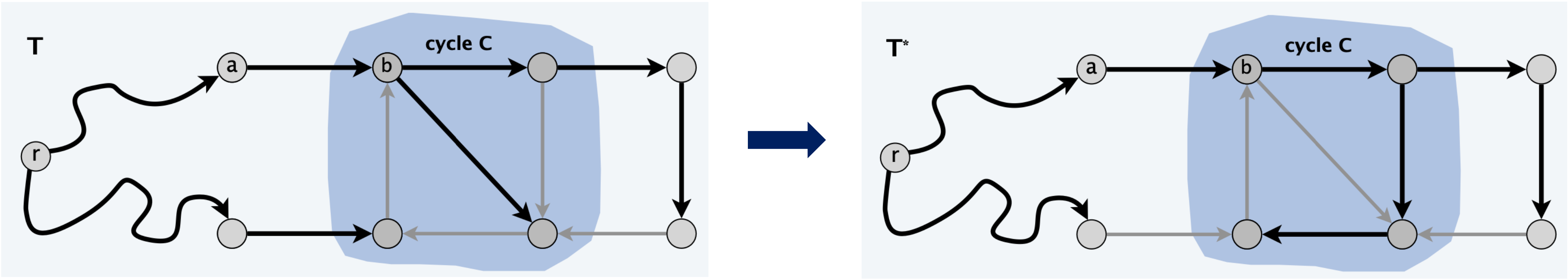
Chu-Liu's Algorithm: Greedy Analysis

- **Lemma.** Let C be a cycle in G containing only 0-cost edges. There exists a min-cost arborescence T rooted at r with exactly one edge entering C.
- **Pf.** (by cases)
  - Case 2. **Claim.** T^* is a min-cost arborescence.

    ✓The cost of T^* is no more than that of T since we add only 0-cost edges.

    ✓T^* is an arborescence, i.e., it has exactly one edge entering each node $v \neq r$ and has no directed cycles.

    - T had no cycles before, now only (a, b) enters C and no cycle exists in C.



Chu-Liu's Algorithm: Greedy Analysis

- **Theorem.** [Chu–Liu 1965, Edmonds 1967] The greedy algorithm finds a min-cost arborescence.
- **Pf.** (by strong induction on number of nodes $|V|$)
  - If the edges of E^* form an arborescence, then min-cost arborescence.
  - Otherwise, we use reduced costs, which is equivalent.
  - After contracting a 0-cost cycle C to obtain a smaller graph G', the algorithm finds a min-cost arborescence T' in G' (by inductive hypothesis).
  - **Lemma:** There exists a min-cost arborescence T in G that corresponds to T'. ▪
- **Time complexity.** $O(mn)$
  - At most n contractions (since each reduces the number of nodes).
  - Finding and contracting cycle C (with reduced costs) takes $O(m)$ time.
  - Transforming T' back into T takes $O(m)$ time. ▪

More on Min-Cost Arborescences

- **Remark.** Chu-Liu's algorithm can be implemented in $O(m \log n)$ time and can be extended to find directed minimum spanning forests.

- **Fact.** [Gabow–Galil–Spencer–Tarjan 1986] There exists an $O(m + n \log n)$ time algorithm to compute a min-cost arborescence.

Announcement

- **Assignment 2 has been released and the deadline is April 2.**

9. Huffman Codes

Encoding

- **Q.** Why do we need encoding?
- **A.** Encoding transforms data of human language to numbers such that they can be processed by digital computers.
- **Example.** Postcode, character codes (Unicode), etc.

- **Q.** Given a text that uses *32* symbols (*26* different letters, space, and some punctuation characters), how can we encode this text in bits?
- **A.** We can encode 2^5 different symbols using a fixed length of *5* bits per symbol. This is called fixed length encoding.

Fixed Length Encoding: Example

- **Q.** Given *64* samples with *1* poison in them. How many guinea pigs do we need to find the poison?
  - ➢ How many possible test results (dead/alive) for *n* guinea pigs?

	🐁	🐁	🐁	🐁	🐁	🐁
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
…						
63	1	1	1	1	1	1

Data Compression

- **Q.** Some symbols *(e, t, a, o, i, n)* are used far more often than others. How can we use this to reduce our encoding size?
- **A.** Encode such characters with fewer bits and the others with more bits.

- **Q.** How do we know when the next symbol begins?
- **A.** Use a separation symbol (like the pause in Morse), or make sure that there is no ambiguity by ensuring that no code is a prefix of another one.
- **Example.** If $c(a) = 01$, $c(b) = 010$, $c(e) = 1$, where function $c(x)$ denotes the code for x, what is 0101?

Prefix Codes

- **Def.** A prefix code for a set S is a function γ that maps each $x \in S$ to a bit string such that for distinct $x, y \in S$, $\gamma(x)$ is not a prefix of $\gamma(y)$.

- **Example.** Consider $c(a) = 11$, $c(e) = 01$, $c(k) = 001$, $c(l) = 10$, $c(u) = 000$.
- **Q.** What is the meaning of *1001000001*?
- **A.** *leuk*
- **Q.** Consider *1GB* text that consists of only the above *5* letters. If the letter frequencies in text are $f_a = 0.4$, $f_e = 0.2$, $f_k = 0.2$, $f_l = 0.1$, $f_u = 0.1$, what is the size of the encoded text?
- **A.** $2f_a + 2f_e + 3f_k + 2f_l + 3f_u = 2.3GB$ ⟵ can we encode to smaller size?

Optimal Prefix Codes

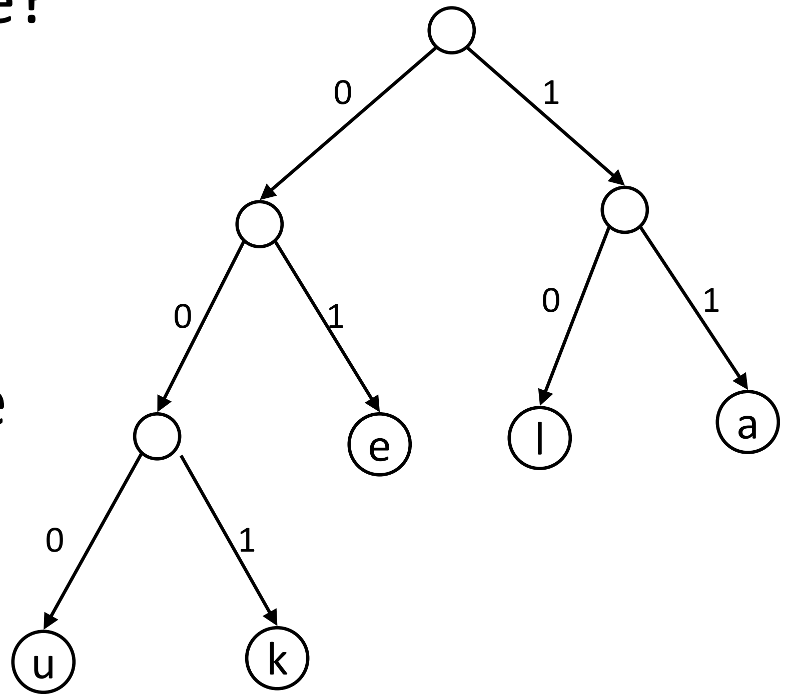
- **Def.** The average number of bits required per letter (ABL) of an encoding γ is the sum, over all symbols $x \in S$, of the symbol's frequency f_x times the number of bits of its encoded string $\gamma(x)$:

$$ABL(\gamma) = \sum_{x \in S} f_x |\gamma(x)|$$

- **Q.** Can we construct a prefix code that has the minimum ABL?

Representing Prefix Codes with Binary Trees

- **Observation.** A prefix code can be modeled with a binary tree.
  - Actually any binary code can be modeled with a binary tree.
- **Example.** Consider *c(a) = 11, c(e) = 01, c(k) = 001, c(l) = 10, c(u) = 000.*
- **Q.** How does the tree of a prefix code look like?

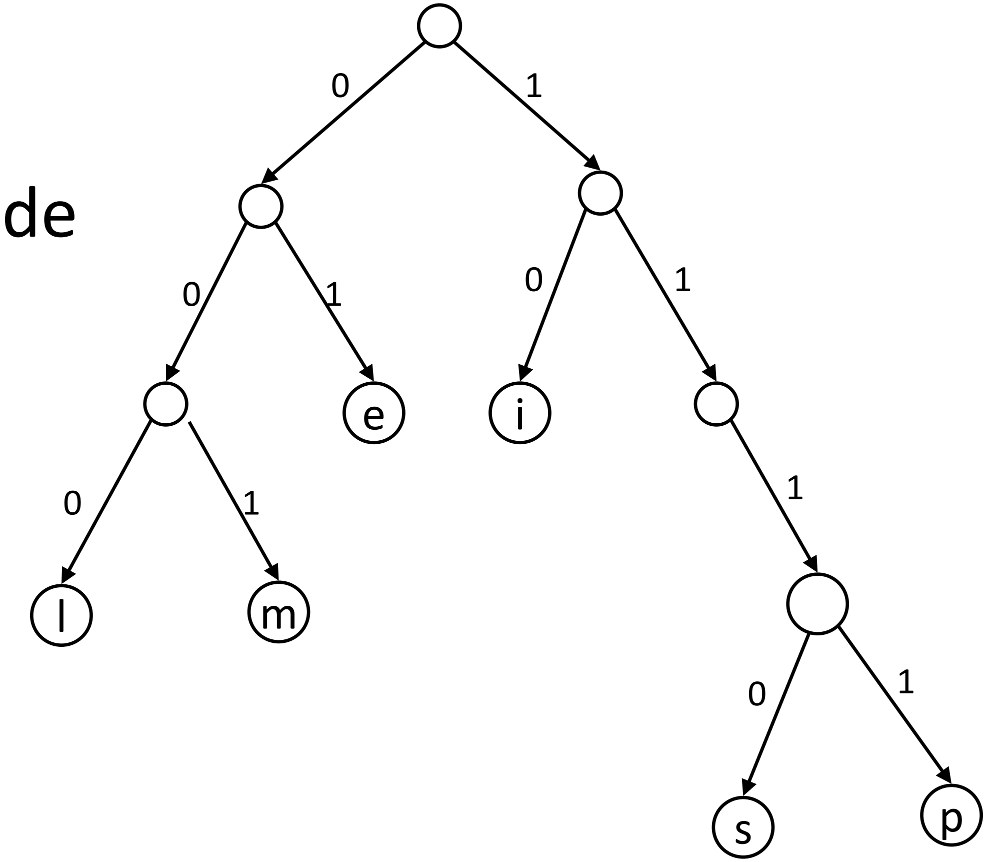


- **Property.** Only the leaves have a label.
- **Pf.** An encoding of *x* is a prefix of an encoding of *y* if and only if the path of *x* is a prefix of the path of *y*.

Representing Prefix Codes with Binary Trees

- **Q.** What is the meaning of *111010001111101000*?
- **A.** *simpel*

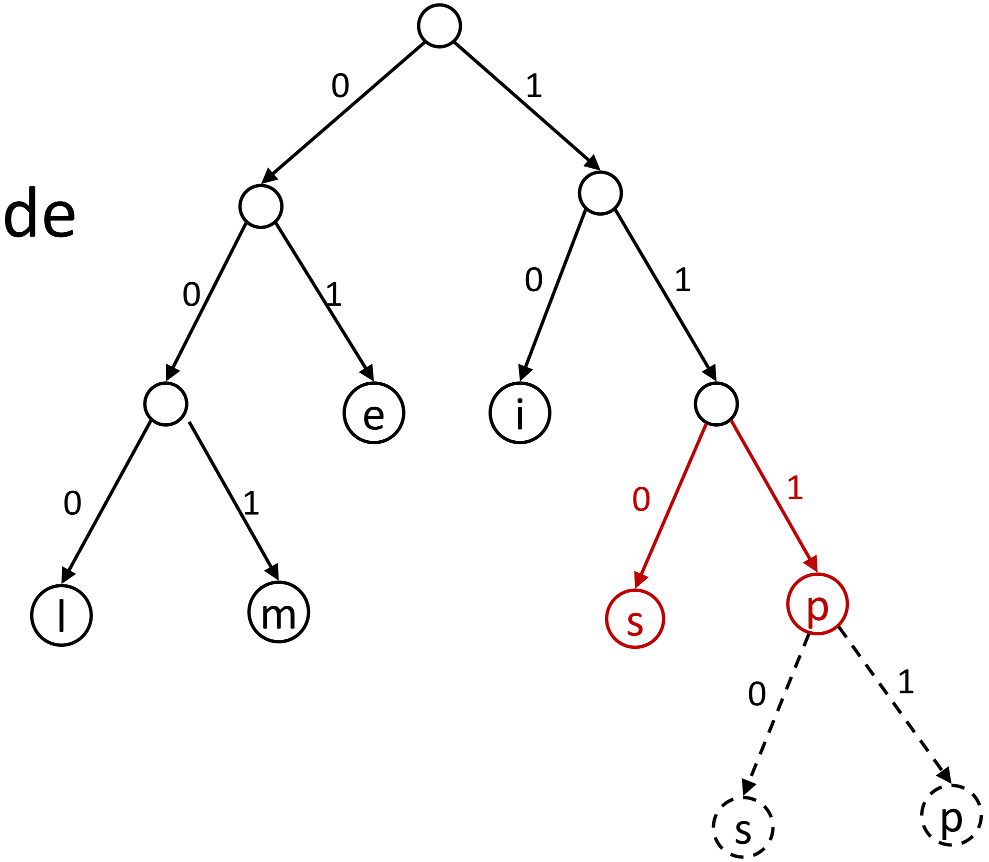
- **Q.** How can we make this prefix code more efficient?



Representing Prefix Codes with Binary Trees

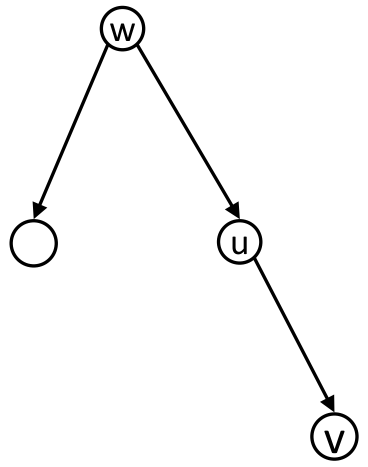
- **Q.** What is the meaning of *111010001111101000*?
- **A.** *simpel*

- **Q.** How can we make this prefix code more efficient?
- **A.** Change encoding of *s* and *p* to a shorter one.



Representing Prefix Codes with Binary Trees

- **Def.** A tree is full if every node that is not a leaf has two children.
- **Claim.** The binary tree corresponding to an optimal prefix code is full.
- **Pf.** (by contradiction)
  - ➢ Suppose *T* is binary tree of optimal prefix code and is not full.
  - ➢ This means there is an internal node *u* with only one child *v*.
  - ➢ Case 1: *u* is the root.
    - ✓Delete *u* and use *v* as the root.
  - ➢ Case 2: *u* is not the root.
    - ✓Delete *u* and make *v* be a child of *w* in place of *u*. (*w* is the parent of *u*.)
  - ➢ In both cases the number of bits needed to encode any leaf in the subtree of *v* is decreased. The rest of the tree is not affected.
  - ➢ Clearly the above new tree has a smaller ABL than *T*. Contradiction!



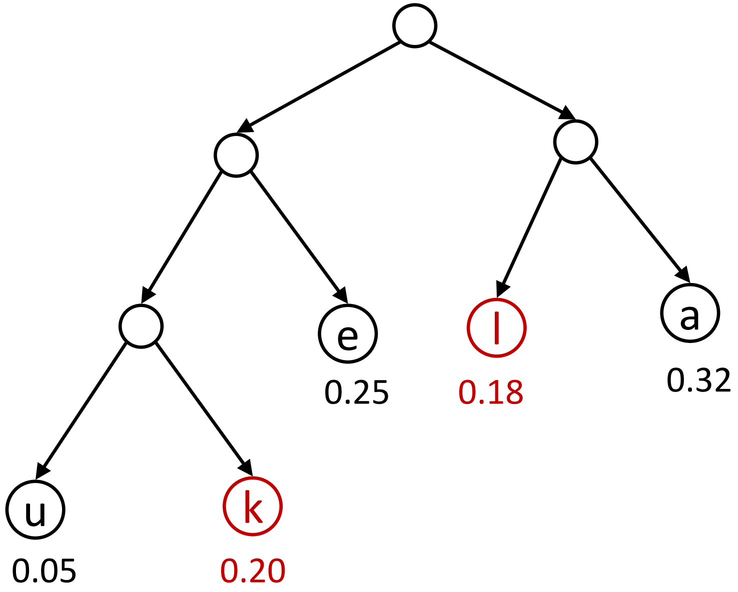
Optimal Prefix Codes: A First Attempt

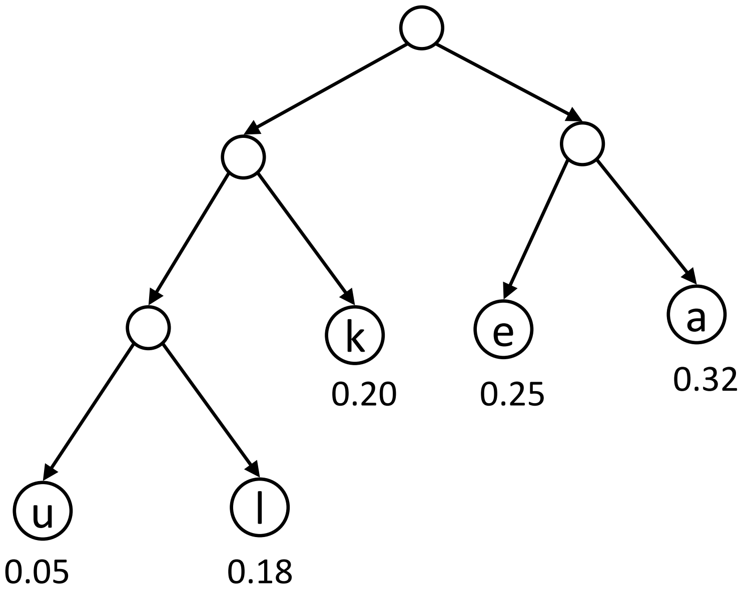
- **Q.** Where should we place symbols with high frequencies in the tree of an optimal prefix code?
- **A.** Near the top.

- **Greedy approach:** [Shannon-Fano 1949] Create tree top-down.
  - Split S into two sets S_1 and S_2 with (almost) equal frequencies.
  - Recursively build trees for S_1 and S_2 and add a parent node to connect them.

Optimal Prefix Codes: A First Attempt

- **Greedy approach:** [Shannon-Fano 1949] Create tree top-down.
  - Split S into two sets S_1 and S_2 with (almost) equal frequencies.
  - Recursively build trees for S_1 and S_2 and add a parent node to connect them.
- **Q.** Does this approach output an optimal prefix code?
- **A.** No! Counterexample: $f_a = 0.32, f_e = 0.25, f_k = 0.20, f_l = 0.18, f_u = 0.05$

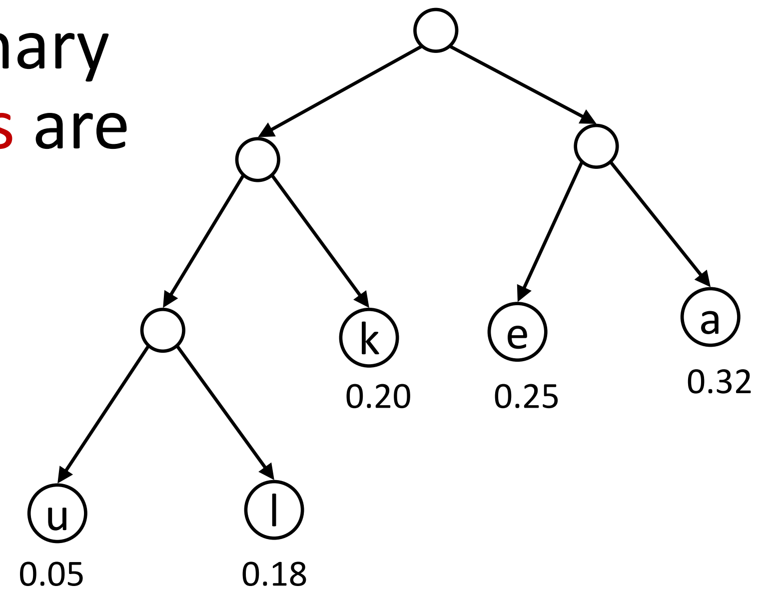




Optimal Prefix Codes: Properties

- **Property 1.** Lowest-frequency letters are at the lowest level.
- **Property 2.** For $n > 1$, the lowest level always contains ≥ 2 leaves.
- **Property 3.** The order letters appear in the same level does not matter.

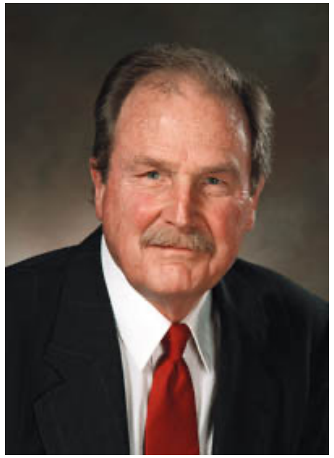
- **Claim.** There is an optimal prefix code with binary tree T^* where the two lowest-frequency letters are assigned to leaves that are siblings in T^*.
- **Pf.** Follows from the above properties.



Optimal Prefix Codes: Greedy Algorithm

- **Claim.** There is an optimal prefix code with binary tree T^* where the two lowest-frequency letters are assigned to leaves that are siblings in T^*.

- **Greedy approach:** [Huffman 1952] Create tree bottom-up.
  - Make two sibling leaves for two lowest-frequency letters y and z.
  - View their common parent as a "meta-letter" ω whose frequency is the sum of frequencies of y and z.
  - Recursively build tree by replacing y and z with ω.
  - Finally, "open up" the meta-letter back to y and z.



Huffman's Algorithm

- **Huffman's algorithm:** (produces a Huffman code)

```
Huffman(S) {
   if (|S| == 2)
      return tree with root and 2 leaves
   else
      let y and z be lowest-frequency letters in S
      let S′ be S with y and z removed
      insert new letter ω in S′ with fω = fy + fz
      T′= Huffman(S′)
      T = add two children y and z to leaf ω from T′
      return T
}
```

- **Q.** What is the time complexity?

delete-min from priority queue S

- **A.** *O(n log n)*. Solve $T(n) = T(n-1) + O(\log n)$ for $T(n)$.

Huffman Codes: Greedy Analysis

- **Lemma.** ABL(T) = ABL(T') + f_ω (ABL: average number of bits per letter)
- **Pf.** Recall that $T = T'$ with two children x, y added to leaf ω.

$$
\begin{aligned}
\text{ABL}(T) &= \sum_{x \in S} f_x \cdot \text{depth}_T(x) \\
&= f_{y^*} \cdot \text{depth}_T(y^*) + f_{z^*} \cdot \text{depth}_T(z^*) + \sum_{x \neq y^*, z^*} f_x \cdot \text{depth}_T(x) \\
&= (f_{y^*} + f_{z^*}) \cdot (1 + \text{depth}_{T'}(\omega)) + \sum_{x \neq y^*, z^*} f_x \cdot \text{depth}_{T'}(x) \\
&= f_\omega \cdot (1 + \text{depth}_{T'}(\omega)) + \sum_{x \neq y^*, z^*} f_x \cdot \text{depth}_{T'}(x) \\
&= f_\omega + f_\omega \cdot \text{depth}_{T'}(\omega) + \sum_{x \neq y^*, z^*} f_x \cdot \text{depth}_{T'}(x) \\
&= f_\omega + \sum_{x \in S'} f_x \cdot \text{depth}_{T'}(x) \\
&= f_\omega + \text{ABL}(T'). \quad \blacksquare
\end{aligned}
$$

Huffman Codes: Greedy Analysis

- **Theorem.** The Huffman code for a given alphabet S achieves the minimum ABL of any prefix code.
- **Pf.** (by induction on $n = |S|$)
  - **Basis Step:** For $n = 2$ there is no shorter code than root and two leaves.
  - Inductive hypothesis (IH): Suppose the Huffman tree T' for S' of size $n - 1$, with letter ω replacing the lowest-frequency letters y and z, is optimal.
  - **Inductive Step:** (by contradiction)
    - ✓Let T be the Huffman tree. Suppose there is tree Z such that ABL(Z) < ABL(T).
    - ✓**Claim:** There is such a Z that has lowest-frequency leaves y and z as siblings.
    - ✓Let Z' be Z with y and z deleted and their former parent labeled ω'.
    - ✓**Lemma:** $\text{ABL}(Z) = \text{ABL}(Z') + f_{\omega'}$ and $\text{ABL}(T) = \text{ABL}(T') + f_{\omega}$.
    - ✓Recall ABL(Z) < ABL(T) and $f_{\omega'} = f_{\omega}$, so ABL(Z') < ABL(T'). Contradiction with IH!

Huffman Codes: Closing Remarks

- **Greedy approach.** Shrink the size of the problem instance, so that a smaller problem can then be solved by recursion.
  - The greedy operation is proved to be "safe": solving the smaller problem still leads to an optimal solution for the original problem.
- **Application.** ZIP file format that supports lossless data compression.
  - Its most common compression algorithm DEFLATE combines LZ77 with Huffman.
  - https://pkwaredownloads.blob.core.windows.net/pem/APPNOTE.txt
- **Drawbacks.** Huffman is not optimal to compress data in all cases.
  - Cannot adapt to letter frequency changes in midstream. ← adaptive compression
  - No "fraction of a bit" per symbol: black-white images with mostly white pixels

can be handled by, e.g., arithmetic coding