Unit **3: Greedy Method**

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.

Greedy – General Method

- The greedy approach suggests
  - constructing a solution through a sequence of steps
  - each expanding a partially constructed solution obtained so far,
  - until a complete solution to the problem is reached.

- On each step the choice made must be:
  - ***feasible***, i.e., it has to satisfy the problem's constraints
  - ***locally optimal***, i.e., it has to be the best local choice among all feasible choices available on that step
  - ***irrevocable***, i.e., once made, it cannot be changed on subsequent steps of the algorithm

General method

- Given n inputs choose a subset that satisfies some constraints.
- A subset that satisfies the constraints is called a feasible solution.
- A feasible solution that maximises or minimises a given (objective) function is said to be optimal.
- Often it is easy to find a feasible solution but difficult to find the optimal solution

Greedy algorithm

An **algorithm is greedy** if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion. One can often design many different greedy algorithms for the same problem, each one locally, incrementally optimizing some different measure on its way to a solution.

Greedy Method

```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
     solution := ∅; // Initialize the solution.
     for i := 1 to n do
     {
          x := Select(a);
          if Feasible(solution, x) then
               solution := Union(solution, x);
     }
     return solution;
}
```

Coin Change Problem

Problem Statement:

Given coins of several denominations find out a way to give a customer an amount with **fewest** number of coins.

Coin Change Problem

Example: if denominations are 1, 5, 10, 25 and 100 and the change required is 30, the solutions are,

Amount : 30 Solutions :

3 x 10 (3 coins)

6 x 5 (6 coins)

1 x 25 + 5 x 1 (6 coins)

1 x 25 + 1 x 5 (2 coins)

The last solution is the **optimal** one as it gives us change only with 2 coins.

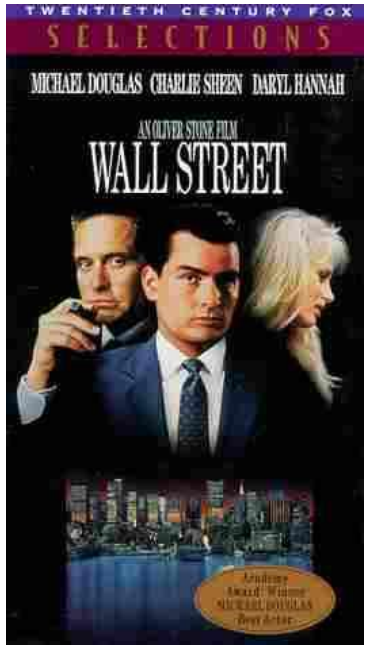
Coin Change Problem

- Solution for coin change problem using greedy algorithm is very intuitive and called as cashier's algorithm.
- Basic principle is:
  - At every iteration for search of a coin, take the largest coin which can fit into remain amount to be changed at that particular time.
  - At the end you will have optimal solution.

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.

   coins selected
  ↙
S ← ϕ
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. Greed 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.

Example: Your Train breaks down in a desert and you decide to walk to nearest town. You have a rucksack but which objects should you take with you ?

Feasible: Any set of objects is a feasible solution provided that they are not too heavy, fit in the rucksack and will help you survive (these are constraints).

An optimal solution is the one that maximises or minimises something

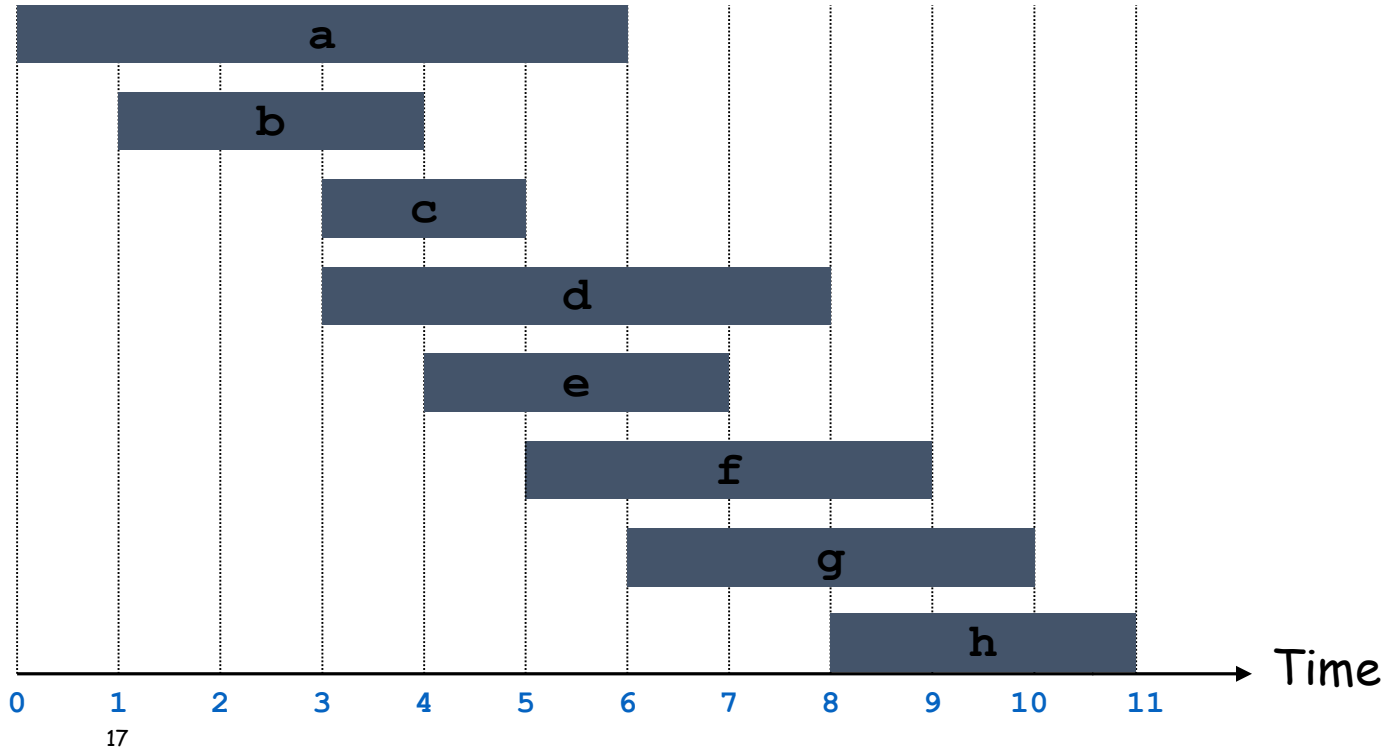
– One that minimises the weight carried

– One that fills the rucksack completely (maximise)

– One that ensures the most water is taken etc.

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 stays ahead

- 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

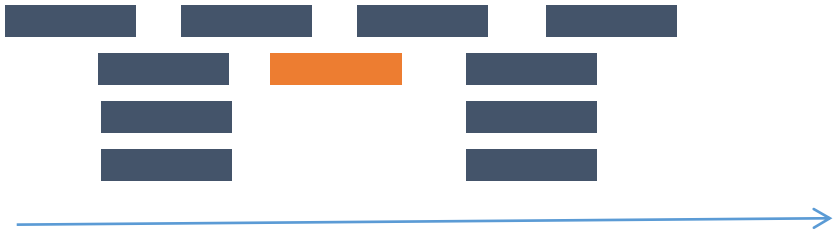
- Greedy template. Consider jobs in some natural order.
  Take each job provided it's compatible with the ones already taken.

counterexample for earliest start time

(a)

counterexample for shortest interval

(b)



counterexample for fewest conflicts

(c)

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.

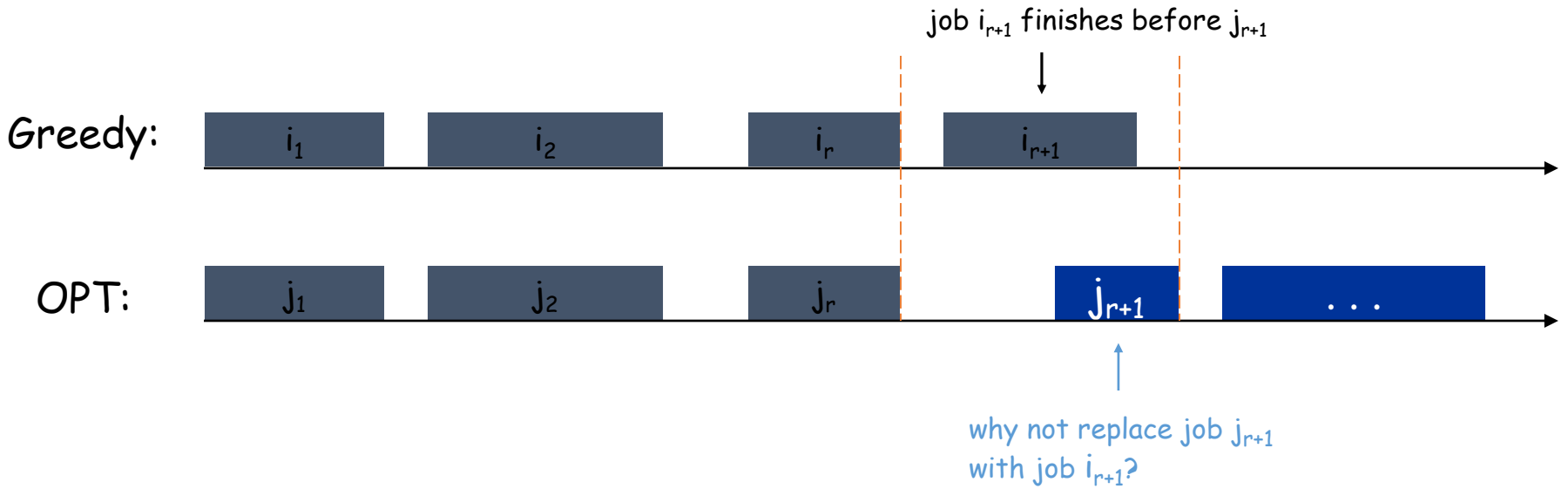
```
Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

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

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

Interval Scheduling: Analysis

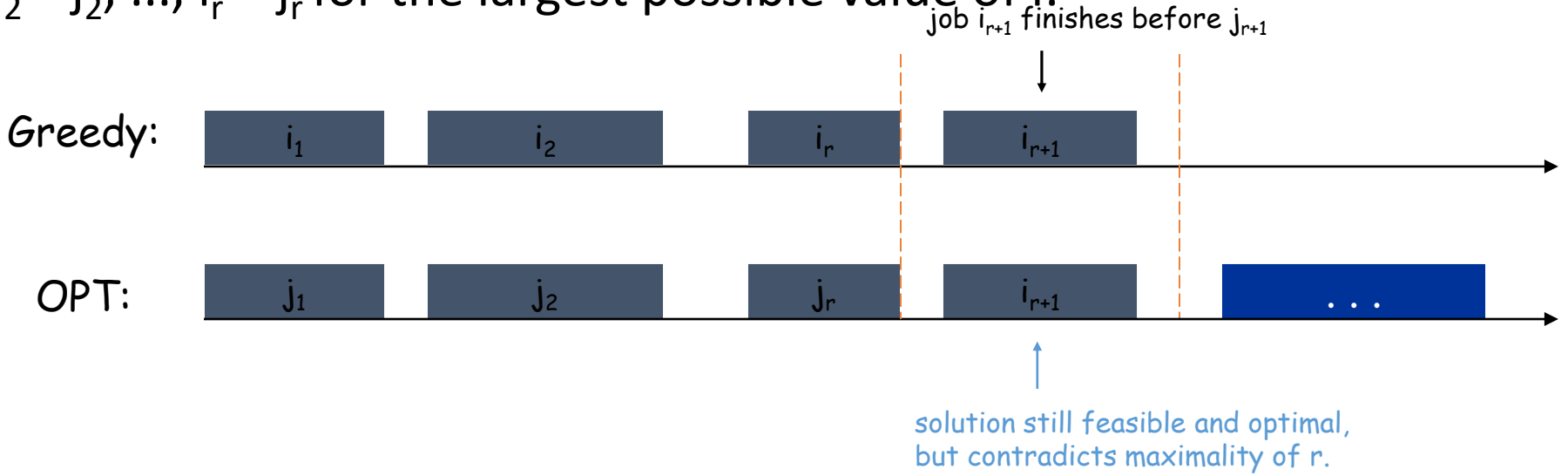
- Theorem. Greedy algorithm is optimal.
- Pf. (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots,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



Interval Scheduling: Analysis

- Theorem. Greedy algorithm is optimal.

- 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.

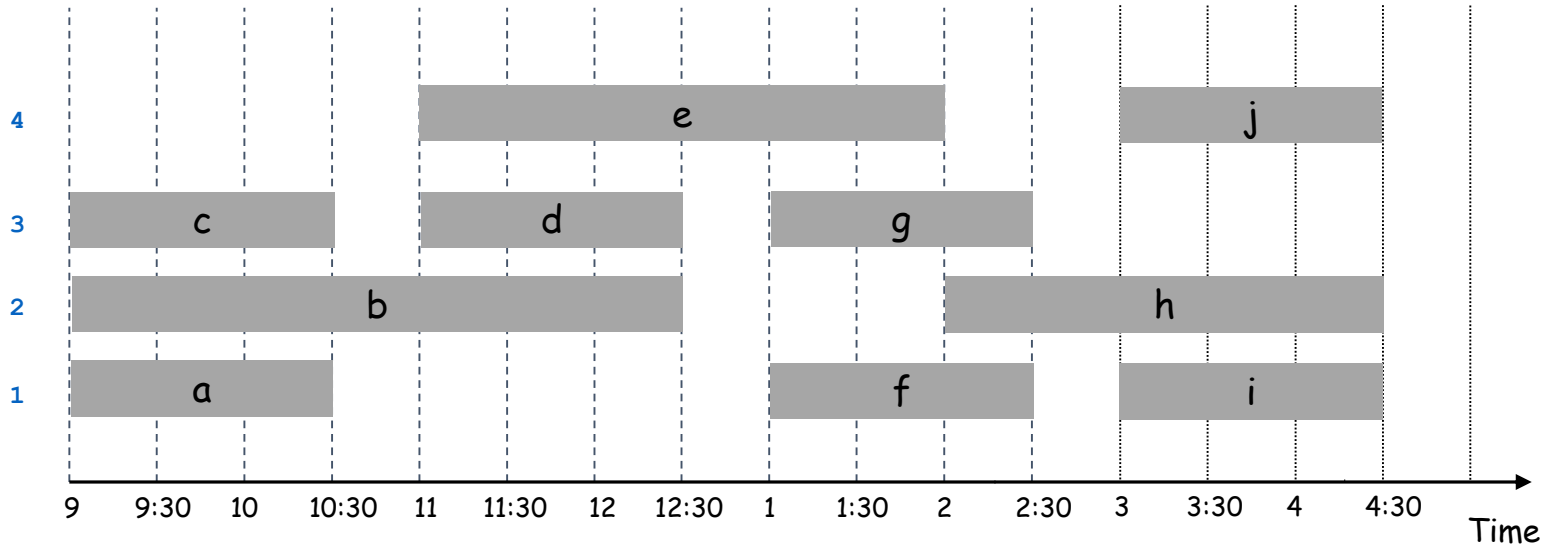


4.1 Interval Partitioning

Interval Partitioning :Scheduling all Intervals

- 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.

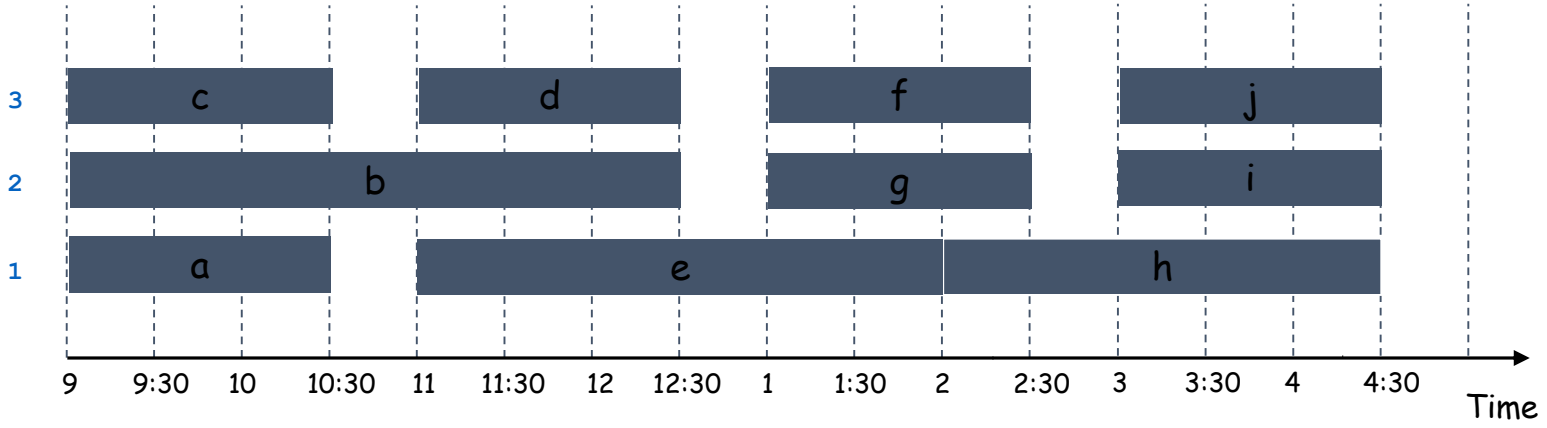
- Ex: This schedule uses 4 classrooms to schedule 10 lectures.



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.

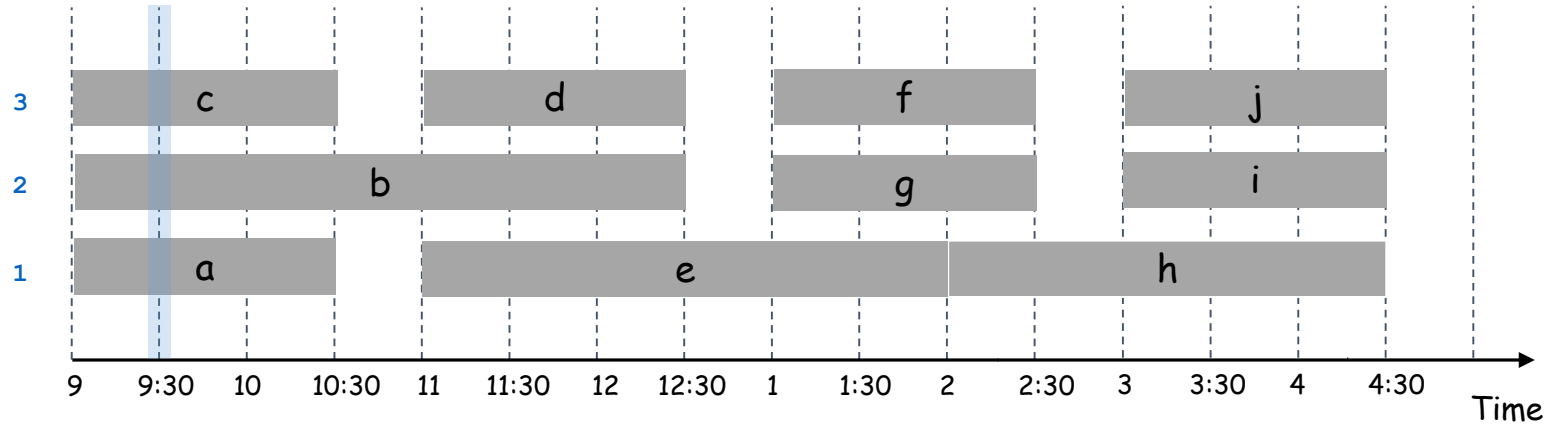
- Ex: This schedule uses only 3.



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

a, b, c all contain 9:30



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 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
}
```

- 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.

- 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. ▪

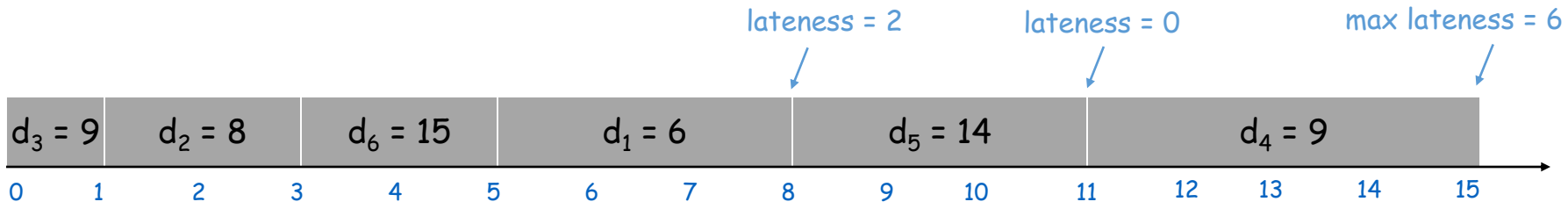
4.2 Scheduling to Minimize Lateness

Scheduling to Minimizing Lateness:An Exchange Argument

- 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: $l_j = \max \{ 0, f_j - d_j \}$.
  - Goal: schedule all jobs to minimize maximum lateness $L = \max_j l_i$

- 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.
    - Req:1,2
    - tj:1,3
    - dj:2,5

	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

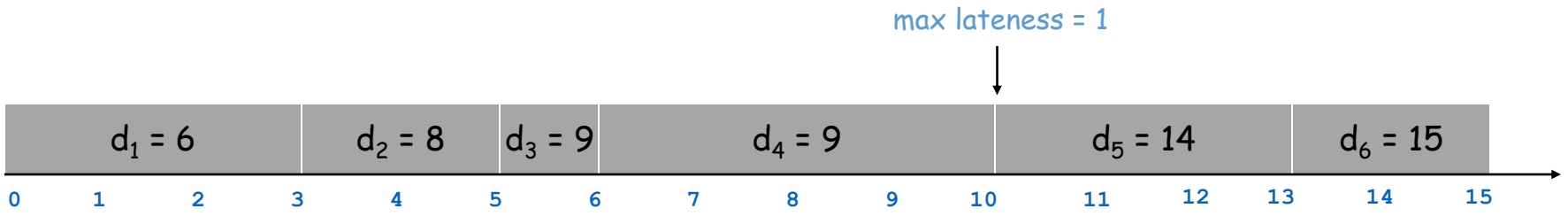
counterexample

Minimizing Lateness: Greedy Algorithm

- Greedy algorithm. Earliest deadline first.

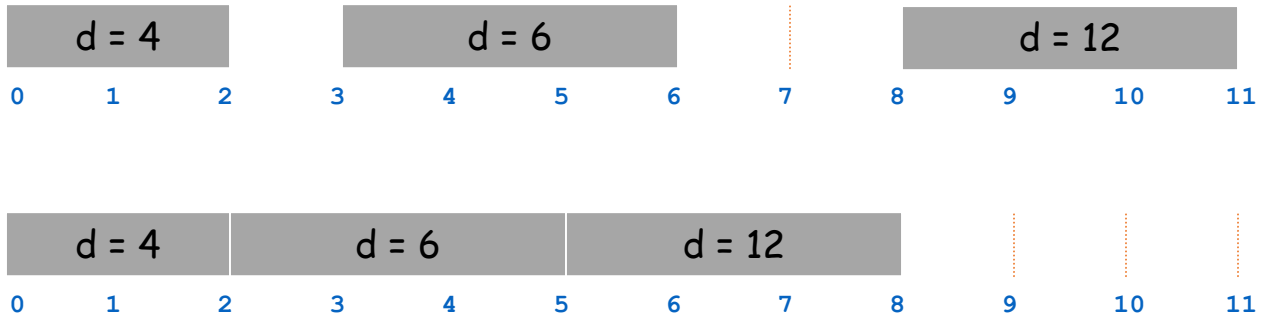
```
Sort n jobs by deadline so that d_1 ≤ d_2 ≤ …
≤ d_n

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

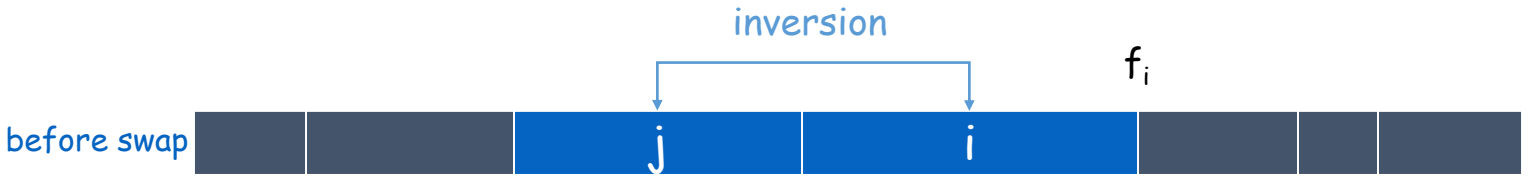
- Observation. There exists an optimal schedule with no idle time.



- 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: i < j but j scheduled before i.



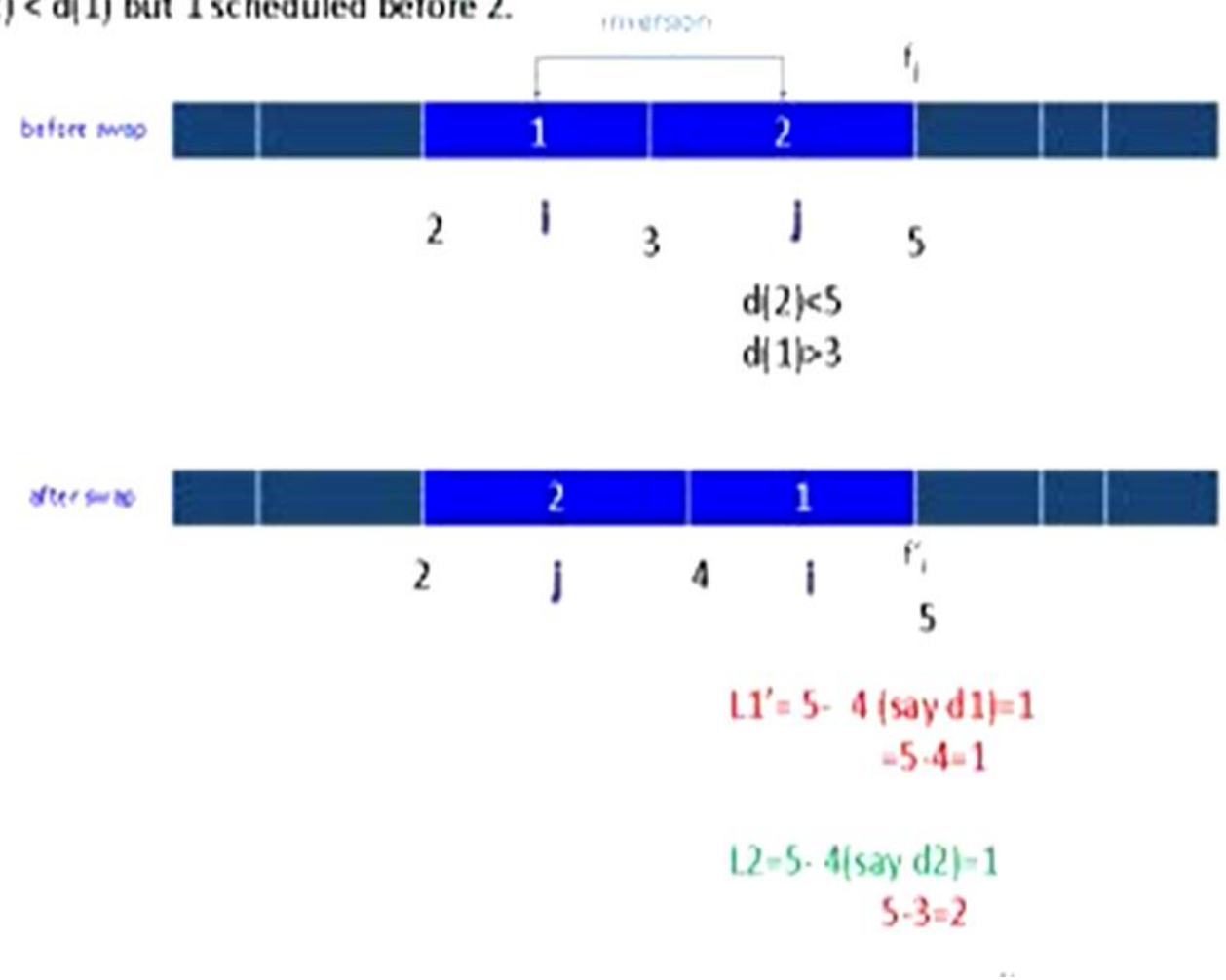
[as before, we assume jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$]

- Observation. Greedy schedule has no inversions.

- 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

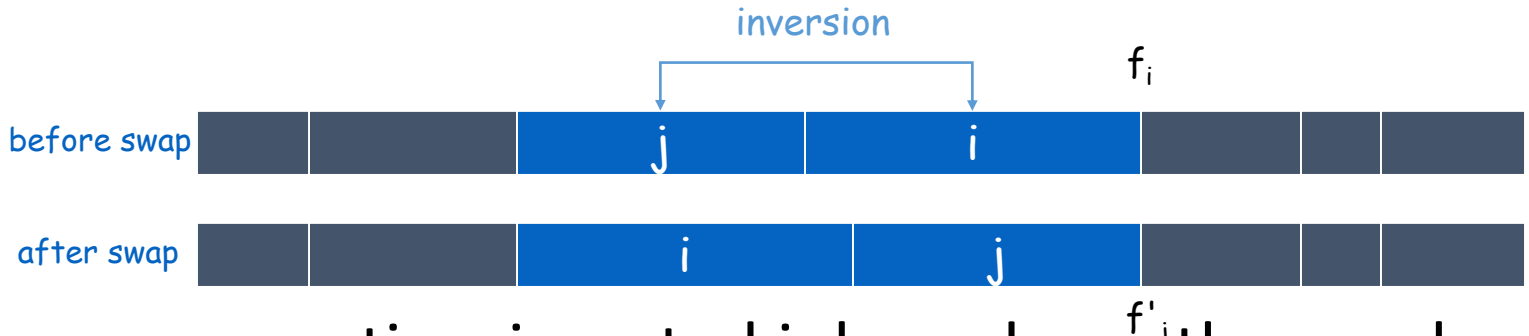
- **Def.** Given a schedule S, an inversion is a pair of jobs 1 and 2 such that: d(2) < d(1) but 1 scheduled before 2.



Lateness Calculation Before Swap:$L1$
L1' (lateness of job 1): Calculate the time when job 1 finishes and subtract its due date.
L1' = Finish time of job 1 - $d(1)$
Example: $L1$
L1' = 5 (finish time) - 4 (due date) = 1
L2' (lateness of job 2): Calculate the time when job 2 finishes and subtract its due date.
$L2$
L2' = Finish time of job 2 - $d(2)$
Example: $L2$
L2' = 5 (finish time) - 3 (due date) = 2

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 ℓl be the lateness before the swap, and let ℓl ' be it afterwards.
  - ℓ l'$_k$ = ℓl$_k$ for all k ≠ i, j
  - ℓl '$_j \leq \ell$l$_j$
  - 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*.
  - If S* has an inversion, let i-j be an adjacent inversion.
    - swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions
    - this contradicts definition of 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.

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

4.3 Optimal Caching

Optimal Offline Caching

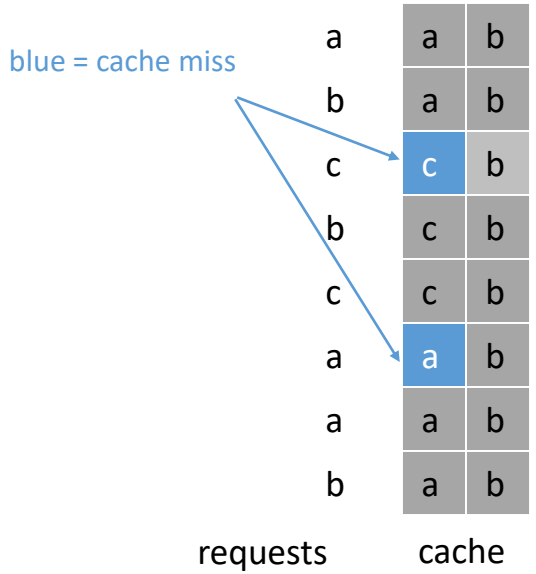
Caching.

- 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 = ab,
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.

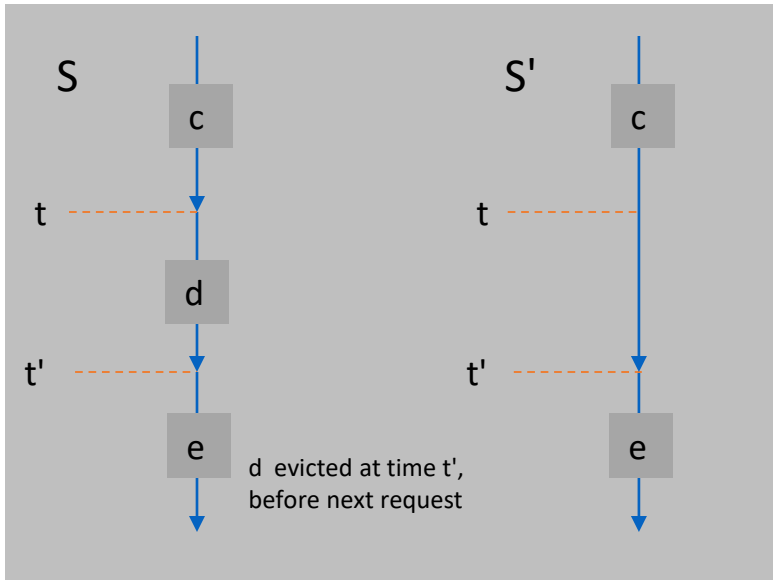
Reduced Eviction Schedules

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

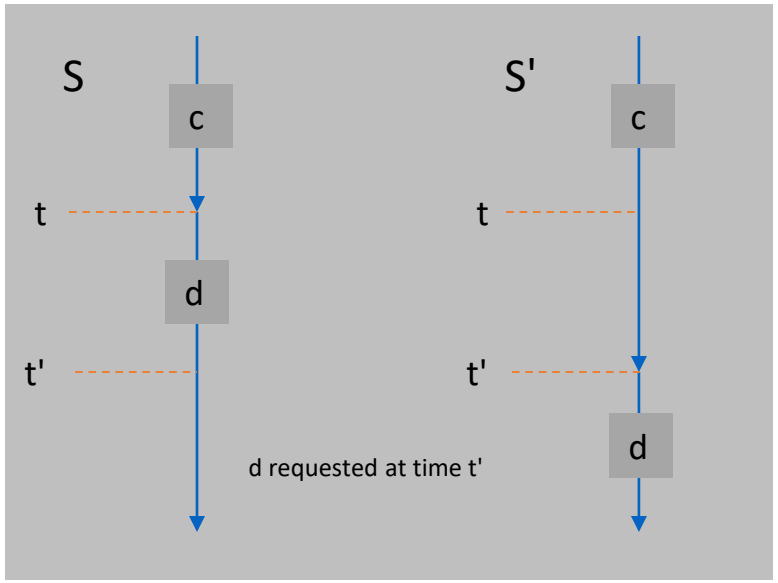
Pf. (by induction on number of unreduced items)

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

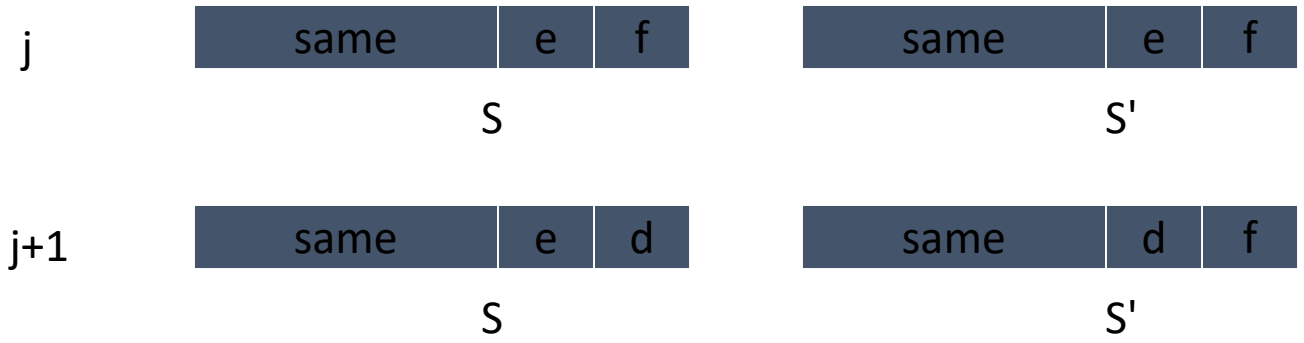
- Theorem. FF is optimal eviction algorithm.
- Pf. (by induction on number or requests j)

> 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

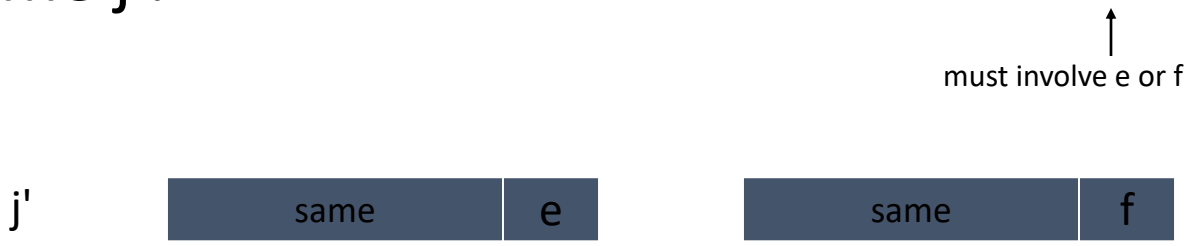
- Pf. (continued)
  - Case 3: (d is not in the cache; S_{FF} evicts e; S evicts $f \neq e$).
    - begin construction of S' from S by evicting e instead of f



    - 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'.



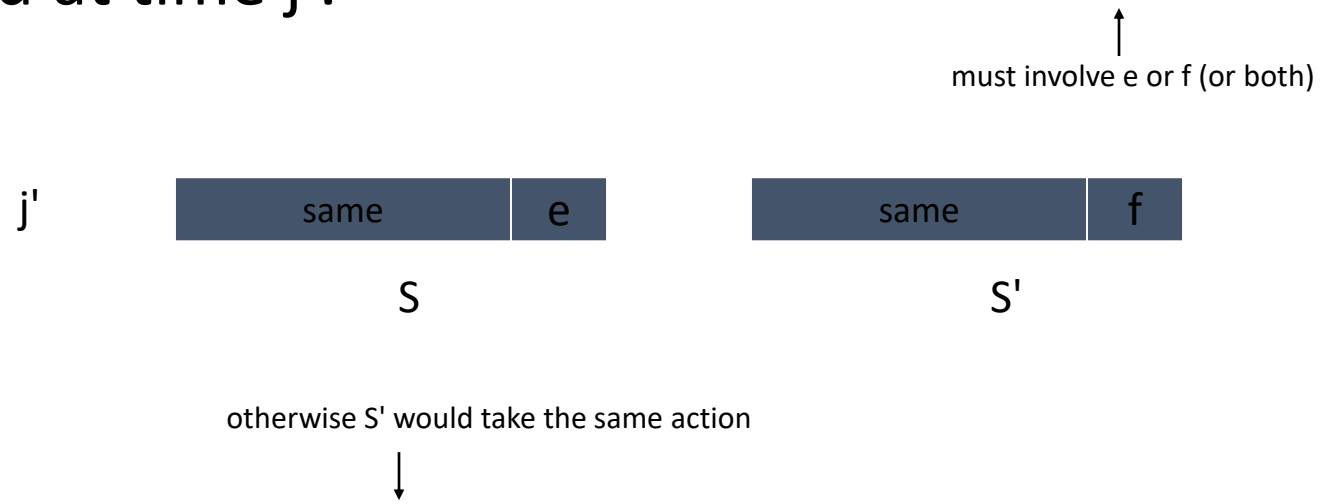
  - Case 3a: g = e. Can't happen with Farthest-In-Future since there must be a request for f before e.

  - 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'.



- 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.

FF with direction of time reversed!

Theorem. FF is optimal offline eviction algorithm.

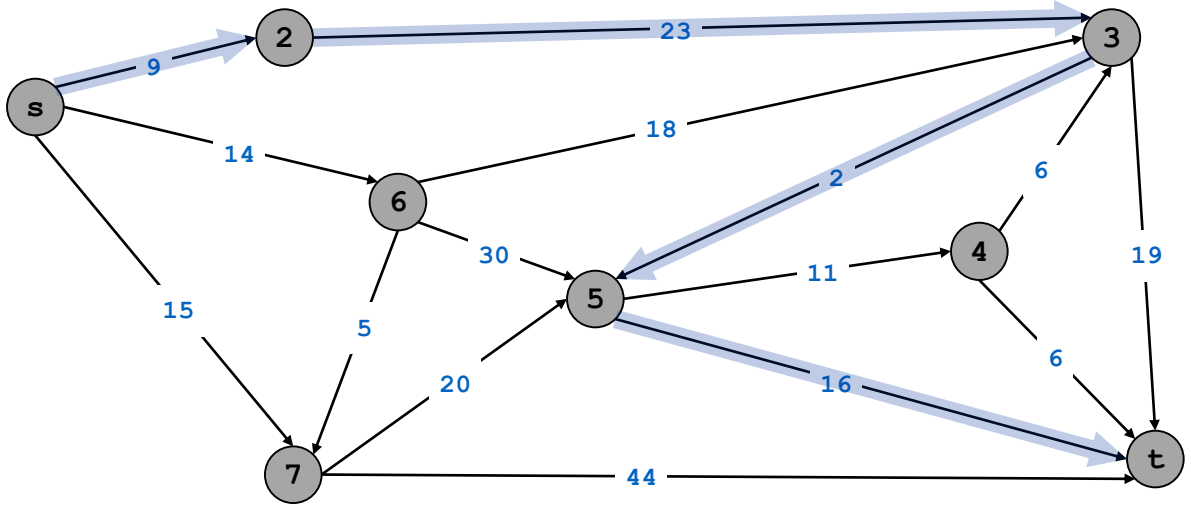
- Provides basis for understanding and analyzing online algorithms.
- LRU is k-competitive. [Section 13.8]
- LIFO is arbitrarily bad.

Part 2 Greedy algorithms on Graphs

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.

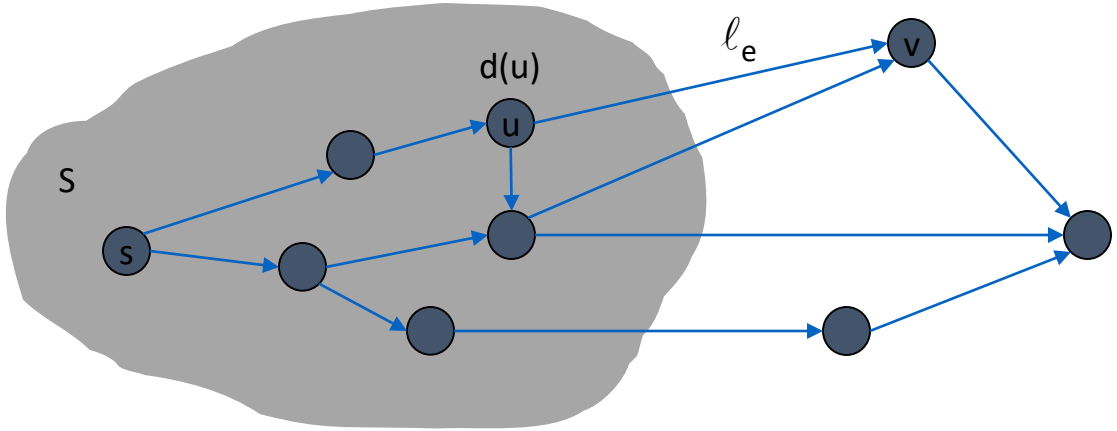
Dijkstra's Algorithm

- 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

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

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

add v to S, and set d(v) = π(v).



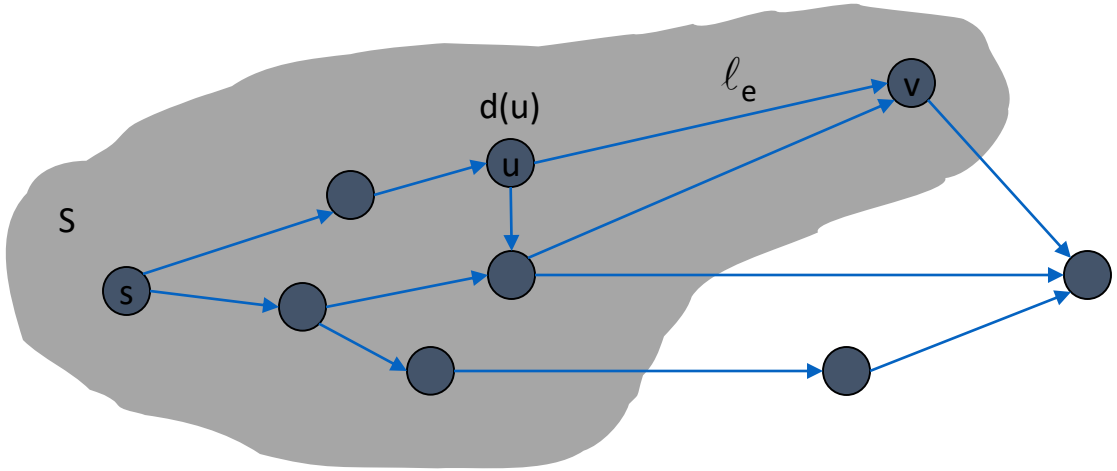
Dijkstra's Algorithm

- 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

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

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

add v to S, and set d(v) = π(v).



Dijkstra's Algorithm: Proof of Correctness

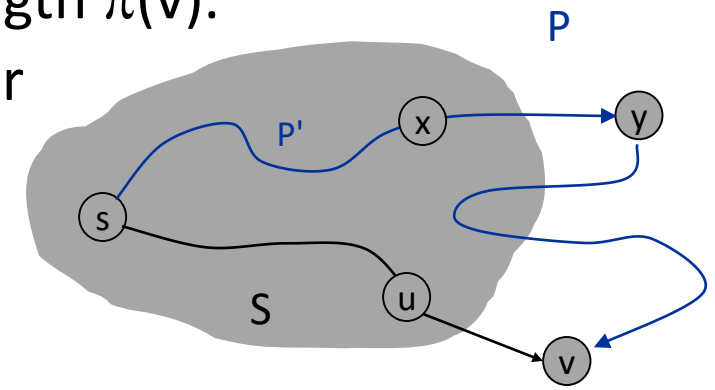
Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.

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.
- 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.



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

- $\ell(P) \geq \ell(P')$: nonnegative weights
- $\ell(P') + \ell(x,y) \geq d(x) + \ell(x,y)$: inductive hypothesis
- $d(x) + \ell(x,y) \geq \pi(y)$: defn of $\pi(y)$
- $\pi(y) \geq \pi(v)$: Dijkstra chose v instead of y

Dijkstra's Algorithm: Implementation

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

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

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

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

PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap †
Insert	n	n	log n	$d \log_d n$	1
ExtractMin	n	n	log n	$d \log_d n$	log n
ChangeKey	m	1	log n	$\log_d n$	1
IsEmpty	n	1	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

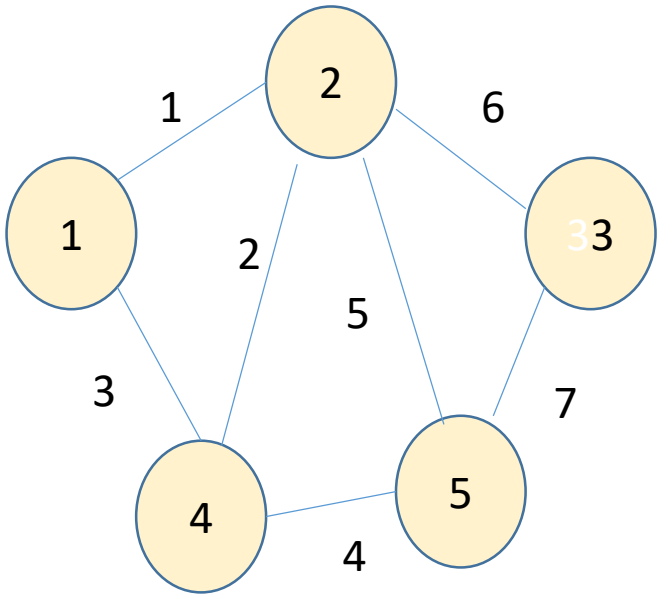
- **PROBLEM**
- Often when we traverse from one point to another or from one place to another place,we want to know the shortest path.
- Given a graph G=(V,E) which models the shortest path problems ,we may want to determine-
  - Given node u,v what is the shortest u-v path
  - Given a start node 's' ,what is the shortest path from s to each other node

SET UP OR INPUT INSTANCE SET UP

- The edges in the graph has a length l_e >=0

l_e ->indicate time it takes to traverse e

->usually associated as cost to travel



Take u=1,v=2, l_e =1

u=1,v=5

l_e ={1,4}+{4,5}=3+4=7

={1,2}+{2,5}=1+5=6

={1,2}+{2,4}+{4,5}

=1+2+4=7

ALGORITHM

//Algorithm:Dijkstra's Algorithm

Let S be the set of explored nodes

For each $u \in S$, we store a distance $d(u)$ 1

Initially $S = \{s\}$ and $d(s) = 0$

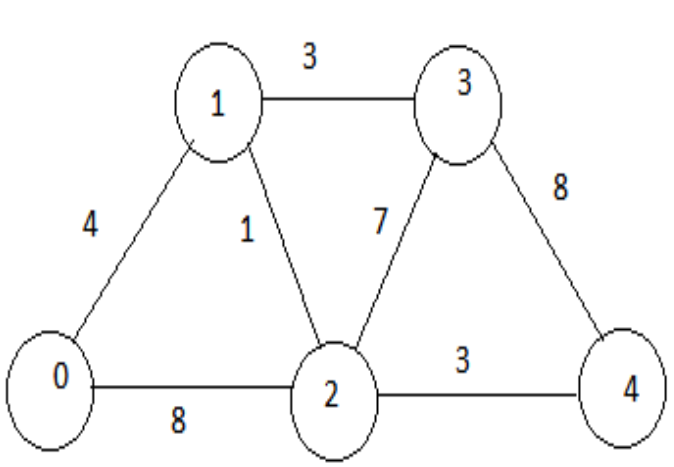
While $S \neq V$

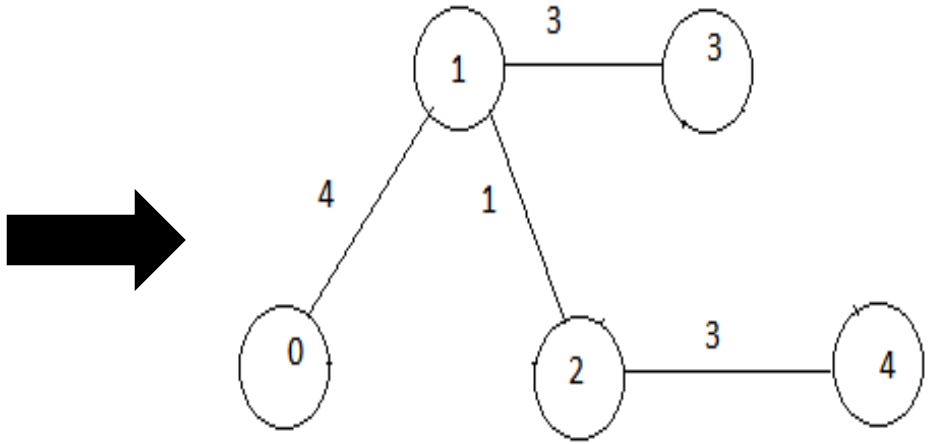
Select a node $v \notin S$ with at least one edge from S for which $d'(v) = \min_{e=(u,v)\, u \in S} d(u) + \ell_e$ is as small as possible 2

Add v to S and define $d(v) = d'(v)$

EndWhile

Example

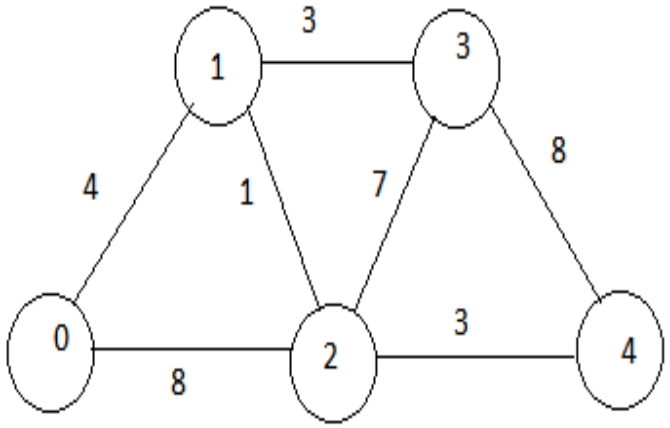




0	1	2	3	4
0	4	5	7	8

Dijkstra’s algorithm example

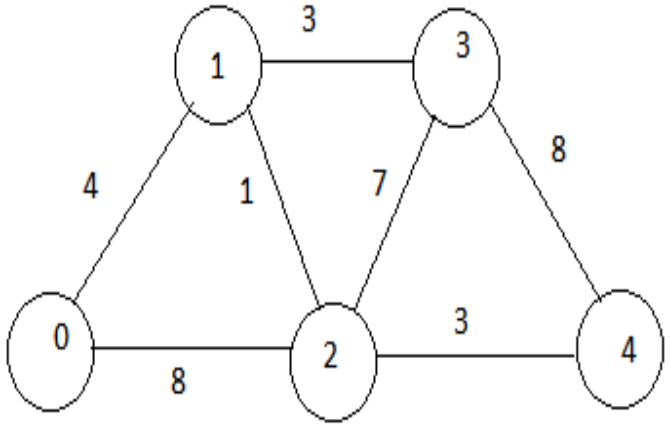
- Source =0



S	Distance[] d[]	Min(d[u]+le)
{0}	0 1 2 3 4 0 | 999 | 999 | 999 | 999	

Dijkstra's algorithm example

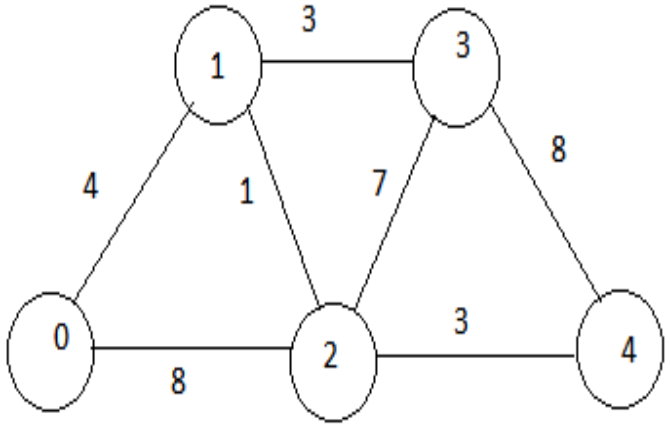
- Source =0



S	**Distance[]** **d[]**	**Min(d[u]+le)**
{0,1}	0 1 2 3 4 0 4	u=0 or u=0 v=1 v=2 min(0+4)=4

Dijkstra’s algorithm example

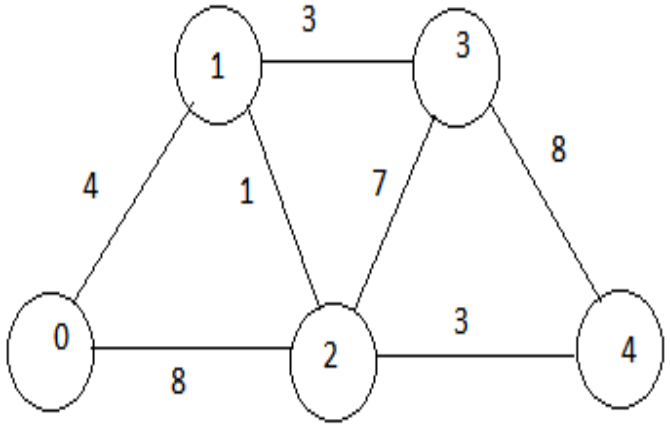
- Source =0



S	**Distance[]** **d[]**	**Min(d[u]+le)**
{0,1,2}	0 1 2 3 4 0 4 5	u=0 or u=1 v=2 v=2 min(0+8)=8 and (4+1)=5

Dijkstra’s algorithm example

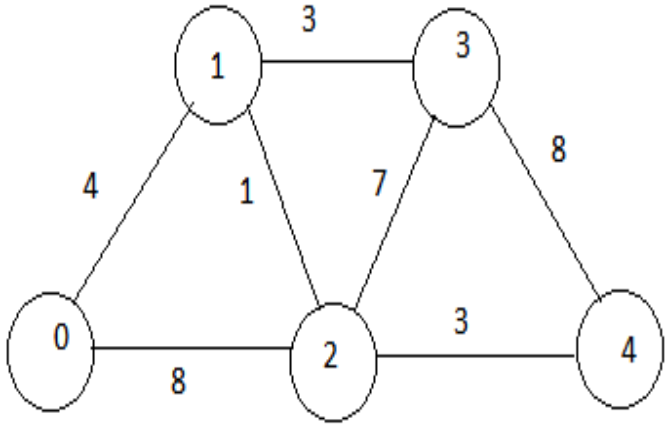
- Source =0



S	**Distance[]** **d[]**	**Min(d[u]+le)**
{0,1,2,3}	0 1 2 3 4 0 | 4 | 5 | 7 |	u=1 or u=2 v=3 v=3 min(4+3)=7 and (4+7)=11

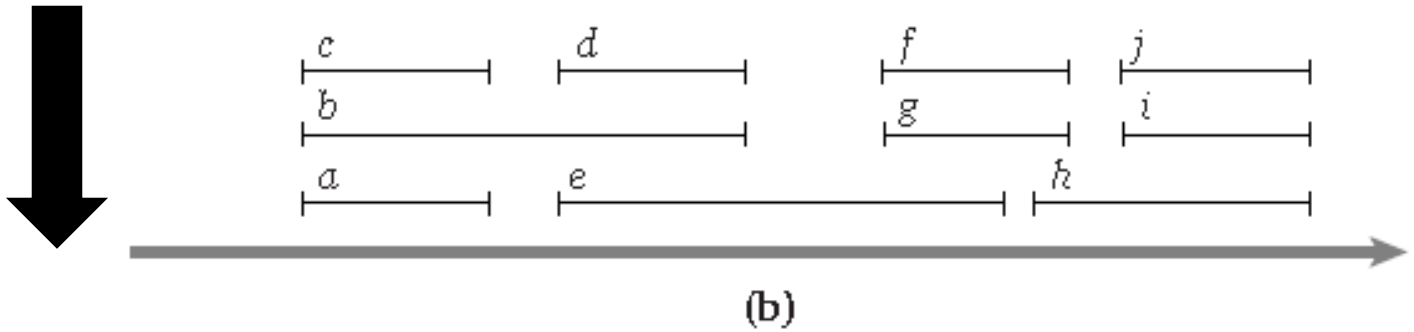
Dijkstra's algorithm example

- Source =0



S	**Distance[]** **d[]**	**Min(d[u]+le)**
{0,1,2,4}	0 1 2 3 4 0 | 4 | 5 | 7 | 8	u=3 or u=2 v=4 v=4 min(3+8)=11 min(5+3)=8

Depth of intervals



- the ***depth*** *of a set of intervals to be the maximum number that pass* over any single point on the time-line.

Analysis

- *In any instance of **Interval Partitioning**, the number of resources needed is at least the depth of the set of intervals.*

- **Proof. Suppose a set of intervals has depth *d, and let I1, . . . , Id all pass over a*** common point on the time-line.
- Then each of these intervals must be scheduled
- on a different resource,
- so the whole instance needs at least *d resources.*