Unit 3- Greedy algorithms

Srinidhi H
Dept of CSE, MSRIT

Topics
1. Interval Scheduling a. Problem. b. Natural approaches to solve the problem. c. Designing algorithm based on greedy approach. d. Analysis
2. Scheduling to minimize lateness: An Exchange argument a. Problem b. Natural approaches to solve the problem. c. Designing algorithm based on greedy approach. d. Analysis
3. Optimal Caching a. Problem b. Algorithm c. Example

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.

1. *Interval Scheduling problem*

- We have a set of requests {1, 2, . . . , *n*}; the *i*th request corresponds to an interval of time starting at *s(i)* and finishing at *f (i)*.
- We'll say that a subset of the requests is ***compatible*** if no two of them overlap in time, and our goal is to accept as large a compatible subset as possible. Compatible sets of maximum size will be called ***optimal***.

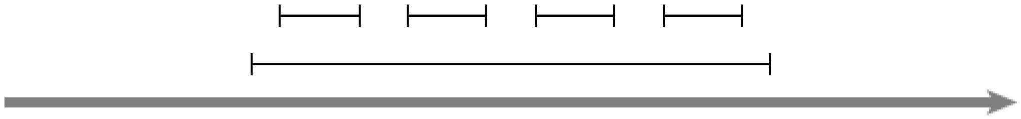
Designing a Greedy Algorithm using natural approaches

The basic idea in a greedy algorithm for interval scheduling is to use a simple rule to select a first request i_1. Once a request i_1 is accepted, we reject all requests that are not compatible with i_1. We then select the next request i_2 to be accepted, and again reject all requests that are not compatible with i_2 and so on.

Some of the most **natural rules** that can be used to solve the interval scheduling problem are:

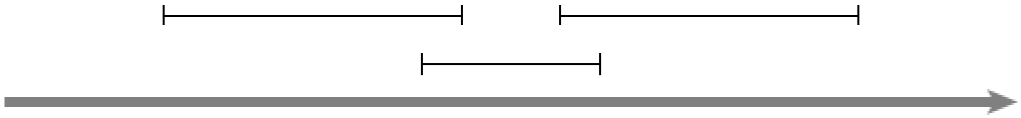
- The most obvious rule might be to always select the available request that starts earliest—that is, the one with **minimal start time *s(i)* earliest start time**. This way our resource starts being used as quickly as possible. This method does not yield an optimal solution.
    - ➢ If the earliest request *i* is for a very long interval, then by accepting request *i* we may have to reject a lot of requests for shorter time intervals. Since our

goal is to satisfy as many requests as possible, we will end up with a suboptimal solution.



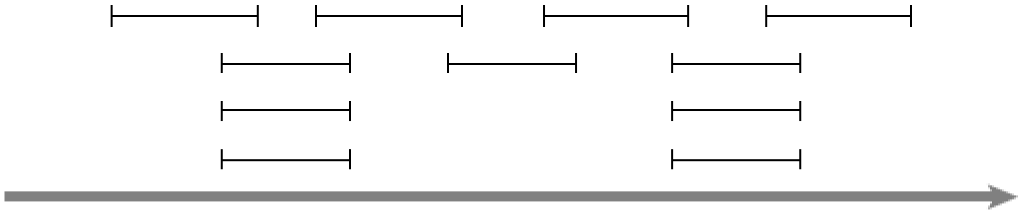
(a)

- By accepting the request that requires the **smallest interval of time**—namely, the request for which *f (i)* − *s(i)* is as small as possible. As it turns out, this is a somewhat better rule than the previous one, but it still can produce a suboptimal schedule. For example, in Figure (b), accepting the short interval in the middle would prevent us from accepting the other two, which form an optimal solution.



(b)

- We could design a greedy algorithm that is based on this idea:
  - for each request, we count the number of other requests that are not compatible, and accept the request that has the fewest number of noncompatible requests. (In other words, we select the interval with the fewest "conflicts.")
  - This greedy choice would lead to the optimum solution in the previous example
  - The unique optimal solution in this example shown in figure (c) is to accept the four requests in the top row. The greedy method suggested here accepts the middle request in the second row and thereby ensures a solution of size no greater than three.



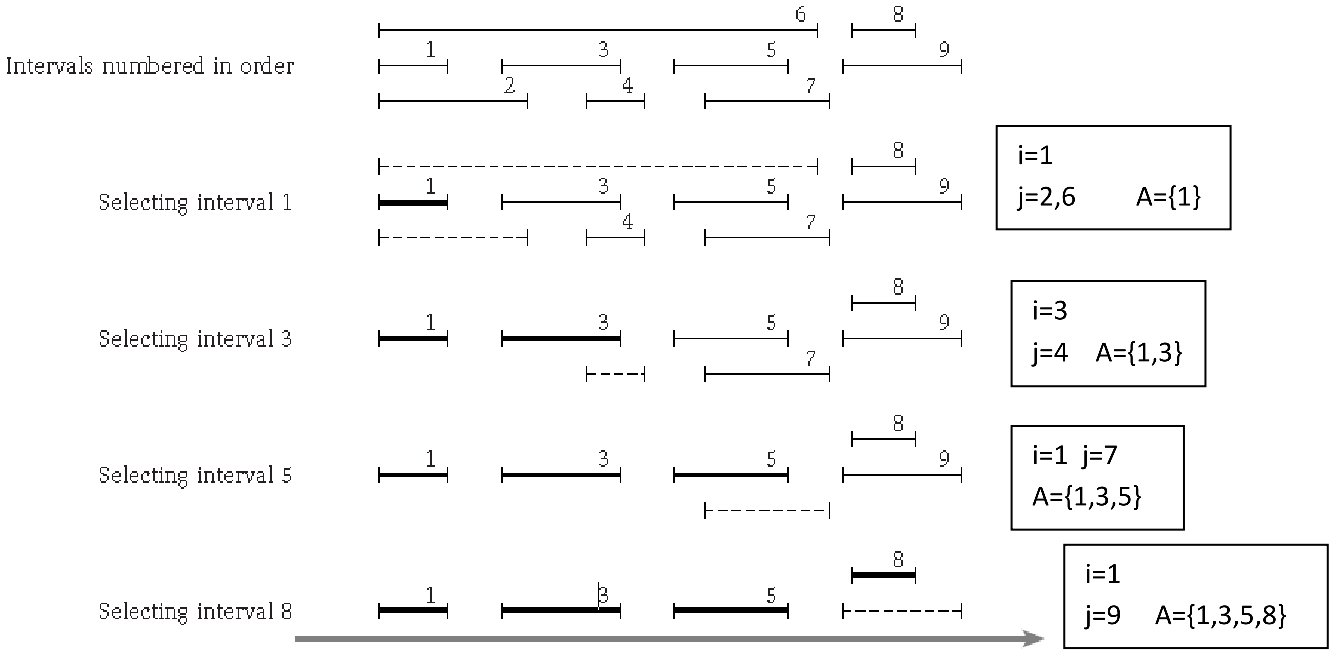
(c)

- **A greedy rule that does lead to the optimal solution is based on a fourth idea**:
  - we should accept first the request that finishes first, that is, the request *I* for which *f (i)* is as small as possible.

Interval Scheduling algorithm

```
Initially let $R$ be the set of all requests, and let $A$ be empty
    While $R$ is not yet empty
        Choose a request $i \in R$ that has the smallest finishing time
        Add request $i$ to $A$
        Delete all requests from $R$ that are not compatible with request $i$
    EndWhile
Return the set $A$ as the set of accepted requests
```

Interval scheduling example with smallest finish time (Greedy)



Implementation and Running Time of Interval Scheduling

The running time of the algorithm shown is *O(n* log *n)* as follows.

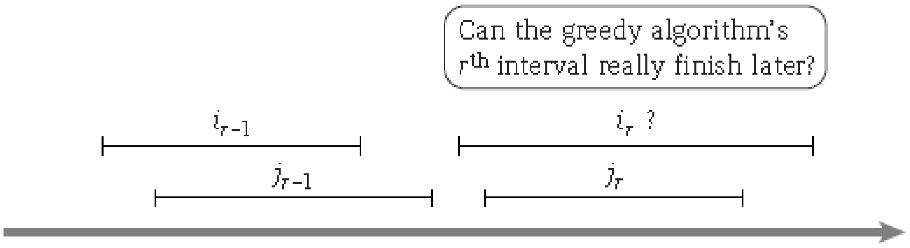
- We begin by sorting the *n* requests in order of finishing time and labeling them in this order; that is, we will assume that $f(i) \leq f(j)$ when $i < j$. This takes time *O(n* log *n).*
- In an additional *O(n)* time, we construct an array *S*[1. . . *n*] with the property that *S*[*i*] contains the value *s(i)*. We now select requests by processing the intervals in order of increasing *f(i)*. We always select the first interval; we then iterate through the intervals in order until reaching the first interval *j* for which $s(j) \geq f(1)$; we then select this one as well.
- More generally, if the most recent interval we've selected ends at time *f* , we continue iterating through subsequent intervals until we reach the first *j* for which $s(j) \geq f$.
- In this way, we implement the greedy algorithm analyzed above in one pass through the intervals, spending constant time per interval. Thus this part of the algorithm takes time *O(n).*

Analysis

1. ***The greedy algorithm returns an optimal set A.***

- In order to prove this we first have to prove that **set A contains a set of compatible requests.**
- Let $i_1, \ldots, i_k$ be the set of requests in A in the order they were added to A. Note that $|A| = k$. Similarly,
- Let the set of requests in optimal set O be denoted by $j_1, \ldots, j_m$. Our goal is to prove that $k = m$.
- Our greedy rule guarantees that $f(i_1) \le f(j_1)$. This is the sense in which we want to show that our greedy rule "stays ahead"—that each of its intervals finishes at least as soon as the corresponding interval in the set O. Thus we now prove that for each $r \ge 1$, the rth accepted request in the algorithm's schedule finishes no later than the rth request in the optimal schedule.
  i.e. *For all indices* $r \le k$ *we have* $f(i_r) \le f(j_r)$.

- For $r = 1$ the statement is clearly true: the algorithm starts by selecting the request i_1 with minimum finish time.
- Now let $r > 1$. We will assume as our induction hypothesis that the statement is true for $r - 1$, and we will try to prove it for r. As shown in Figure 4.3, the induction hypothesis lets us assume that $f(i_{r-1}) \le f(_{jr-1})$.


Can the greedy algorithm's r^{th} interval really finish later?
i_{r-1}
i_r ?
j_{r-1}
j_r

- the greedy algorithm always has the option (at worst) of choosing j_r and thus fulfilling the induction step. i.e. j_r is compatible with i_{r-1} and $j_{r-1.}$
- We know (since O consists of compatible intervals) that $f(j_{r-1}) \le s(j_r)$. Combining this with the induction hypothesis $\boldsymbol{f(i_{r-1}) \le f(j_{r-1})}$**, we get** $\boldsymbol{f(i_{r-1}) \le s(j_r)}$**.**
- Thus the interval j_r is in the set R of available intervals at the time when the greedy algorithm selects i_r . The greedy algorithm selects the available interval with *smallest* finish time; since interval j_r is one of these available intervals, we have $\boldsymbol{f(i_r) \le f(j_r)}$**.**
- Thus we have formalized the sense in which the greedy algorithm is remaining ahead of O: for each r, the r^{th} interval it selects finishes at least as soon as the r^{th} interval in O.

We will now prove the statement by contradiction.

- If A is not optimal, then an optimal set O must have more requests, that is, we must have $m > k$.
- With $r = k$, we get that $f(i_k) \le f(j_k)$. Since $m > k$, there is a request j_{k+1} in O.
- This request starts after request j_k ends, and hence after i_k ends.
- So after deleting all requests that are not compatible with requests $i_1, \ldots, i_k$, the set of possible requests R still contains j_{k+1}.
- But the greedy algorithm stops with request i_k, and it is only supposed to stop when R is empty—a contradiction. Hence, the set A returned is optimal

2. *Scheduling to minimize lateness: An Exchange argument*

Problem

- Consider again a situation in which we have a single resource and a set of *n* requests to use the resource for an interval of time. Assume that the resource is available starting at time *s*. In contrast to the previous problem, however, each request is now more flexible.
- Instead of a start time and finish time, the request *i* has a deadline d_i, and it requires a contiguous time interval of length t_i, but it is willing to be scheduled at any time before the deadline.

Each accepted request must be assigned an interval of time of length t_i, and different requests must be assigned with no overlapping intervals.

Suppose that we plan to satisfy each request, but we are allowed to let certain requests run late. Thus, beginning at our overall start time *s*, we will assign each request *i* an interval of time of length t_i;

let us denote this interval by $[s(i), f(i)]$, with $f(i) = s(i) + t_i$.

where, s(i)= start time of the interval i

f(i)=finish time of the interval i

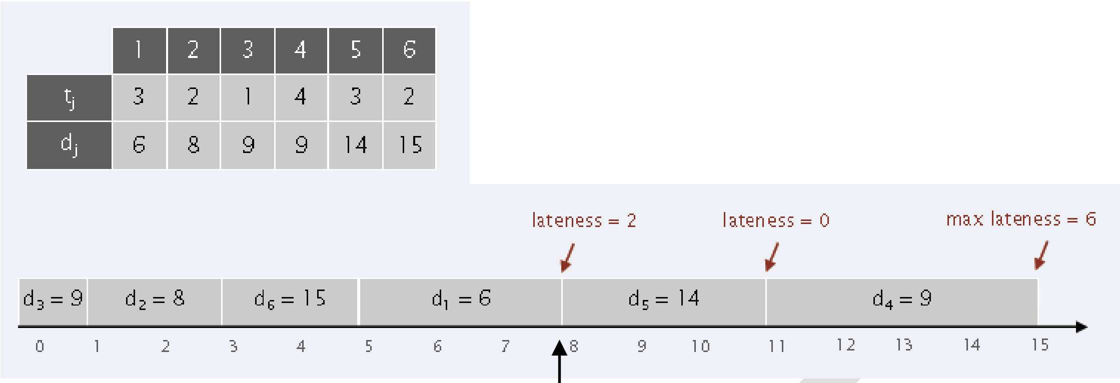
t_i= time length of the interval i

Unlike the previous problem, then, the algorithm must actually determine a start time (and hence a finish time) for each interval.

We say that a request *i* is *late* if it misses the deadline, that is, if $f(i) > d_i$.
The *lateness* of such a request *i* is defined to be $l_i = f(i) - d_i$. We will say that $li = 0$ if request *i* is not late.

The goal in our new optimization problem will be to schedule all requests, using nonoverlapping intervals, so as to minimize the *maximum lateness*, $L = \max_i l_i$.

Example



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

Compare finish time with deadline and then calculate lateness

f(i)=8 d(i)=6

l(i)=8-6=2

Natural approaches to the problem

- One approach would be to schedule the jobs in order of increasing length t_i, so as to get the short jobs out of the way quickly.

Example:

	1	2
t_j	1	3
d_j	2	5

t_1=1	t_2=3

0 1 3

Counter Example:

	1	2
t_j	1	10
d_j	100	10

If request 2 is scheduled first

	1	2
t_j	1	10
d_j	100	10

t_1=1, d_1=100	t_2=10, d_2=10

0 1 11

t_2=10 d_2=10	t_1=1 d_1=100

0 10 11

Lateness for the request 2 is 1

Lateness for the request 2 is 0

- ***slack time*** $d_i - t_i$ is very small—they're the ones that need to be started with minimal delay. So a more natural greedy algorithm would be to sort jobs in order of increasing slack $d_i - t_i$.

  Example:

	1	2
t_j	1	3
d_j	2	5
sl_i	1	2

sl_1=1 d_1=2	sl_2=2 d_2=5

0 1 4

Counter Example:

	1	2
t_j	1	10
d_j	2	10
sl_i	1	0

→

If request 1 is scheduled first

	1	2
t_j	1	10
d_j	2	10
sl_i	1	0

sl_2=0 d_2=10	sl_1=1 d_1=2

0 10 11

sl_1=1 d_1=2	s_2=0 d_2=10

0 1 11

Lateness for the request 1 is 9

Lateness for the request 2 is 1

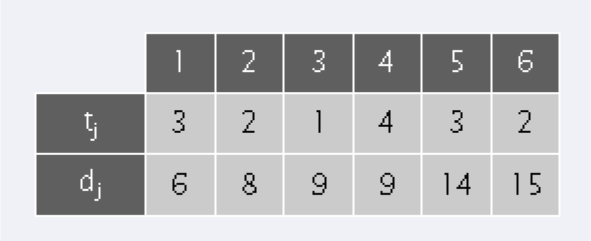
Designing greedy algorithm

In the above discussions, the right hand side of the interval scheduling shows the earliest deadline first strategies.

Algorithm

```
Order the jobs in order of their deadlines
Assume for simplicity of notation that $d_1 \leq . . . \leq d_n$
Initially, $f = s$
Consider the jobs $i = 1, . . . , n$ in this order
      Assign job $i$ to the time interval from $s(i) = f$ to $f(i) = f + t_i$
      Let $f = f + t_i$
End
Return the set of scheduled intervals $[s(i), f(i)]$ for $i = 1, . . . , n$
```

Example



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

d1<d2<d3≤d4<d5<d6, f=0

1. i=1 Add Job J1 to the schedule.

J1 d1=6

0 3

s(1) = f =0
f(1)= s(1)+t(1)=0+3=3
f= f + t(1)=0+3=3

2. i=2 Add Job J2 to the schedule.

J1 d1=6	J2 d2=8

0 3 5

s(2) = f =3
f(2)= s(2)+t(2)=3+2=5
f= f + t(2)=3+2=5

3. i=3 Add Job J3 to the schedule.

J1 d1=6	J2 d2=8	J3 d3=9

0 3 5 6

s(3) = f =5
f(3)= s(3)+t(3)=5+1=6
f= f + t(3)=5+1=6

4. i=4 Add Job J4 to the schedule.

s(4) = f =6
f(4)= s(4)+t(4)=6+4=10
f= f + t(4)=6+4=10

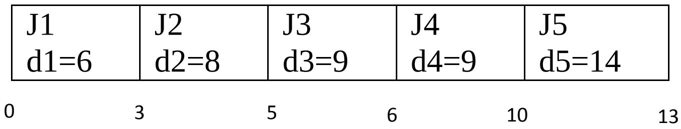
J1 d1=6	J2 d2=8	J3 d3=9	J4 d4=9

0 3 5 6 10

← Lateness=1

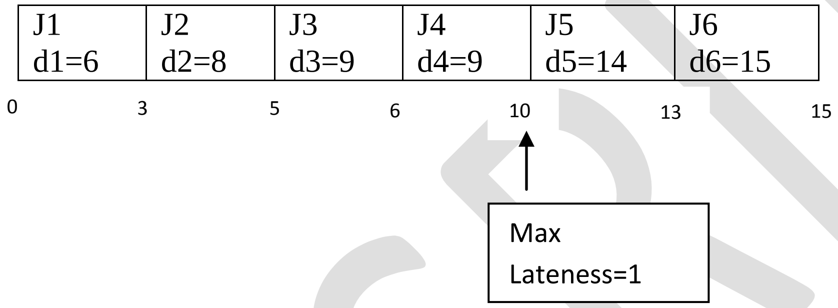
5. i=5 Add Job J5 to the schedule.

s(5) = f =10
f(5)= s(5)+t(5)=10+3=13
f= f + t(5)=10+3=13



6. i=6 Add Job J6 to the schedule.

s(6) = f =13
f(6)= s(6)+t(6)=13+2=15
f= f + t(6)=13+2=15



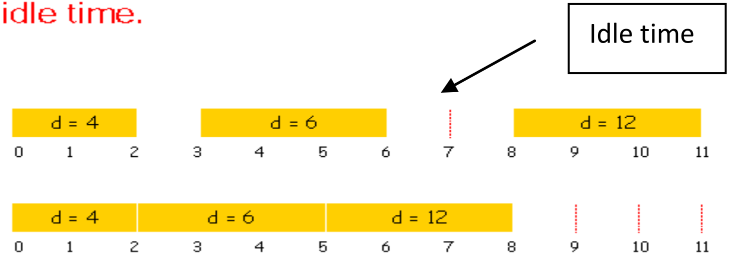
Analysis

(4.9) *There is an optimal schedule that has no inversions and no idle time.*

The main step in showing the optimality of our algorithm is to establish that there is an optimal schedule that has no inversions and no idle time.
To do this, **we will start with any optimal schedule having no idle time**; we will then **convert it into a schedule with no inversions** without increasing its maximum lateness. Thus the resulting scheduling after this conversion will be optimal as well. This method is known as “Exchange argument” where the optimal schedule is converted to the greedy schedule by exchanging.

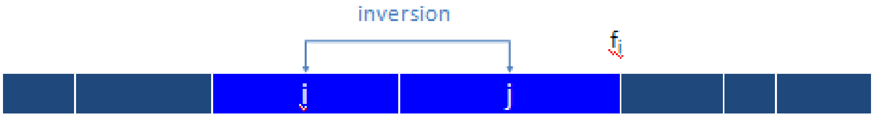
- *There is an optimal schedule with no idle time.*

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



Observation. The greedy schedule has no idle time.

- *There is an optimal schedule that has no inversions*
  - We first try characterizing schedules in the following way. We say that a schedule *A* has an ***inversion*** if a job *i* with deadline d_i is scheduled before another job *j* with earlier deadline $d_j < d_i$ as shown below.



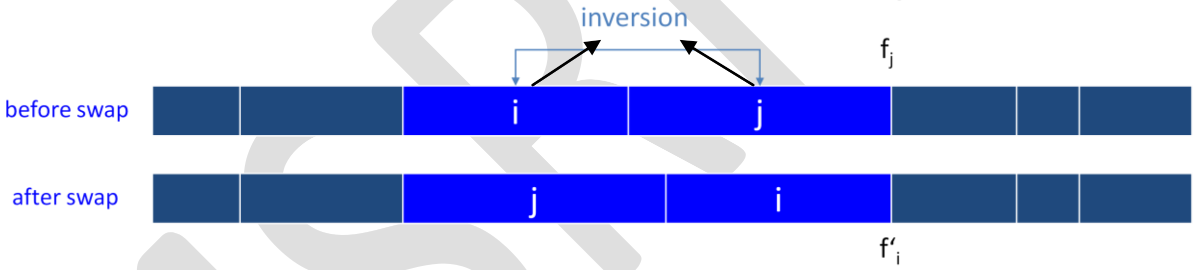
  - Let us consider Optimal schedule O has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has dj < d_i.
  - The pair *(i, j)* formed an inversion in O, this inversion can be eliminated by the swap, and no new inversions are created. Thus we have

(b) After swapping i and j we get a schedule with one less inversion.

(c) The new swapped schedule has a maximum lateness no larger than that of O.

It is clear that if we can prove (c), then we are done.

  - Consider the jobs i and j in the optimal schedule before swap and after swap as shown below.



  - Let ℓ be the lateness before the swap, and let ℓ' be it afterwards.
    - $\ell'_k = \ell_k$ for all $k \neq i, j$.
    - Also $l_j' \leq l_j$
    - If job i is late:

$$\begin{aligned} \ell'_i &= f'_i - d_i && \text{(definitio n)} \\ &= f_j - d_i && (i \text{ finishes at time } f_j) \\ &\leq f_j - d_j && (j < i) \\ &\leq \ell_j && \text{(definitio n)} \end{aligned}$$

Hence proved that "There is an optimal schedule with no idle time and no inversions".

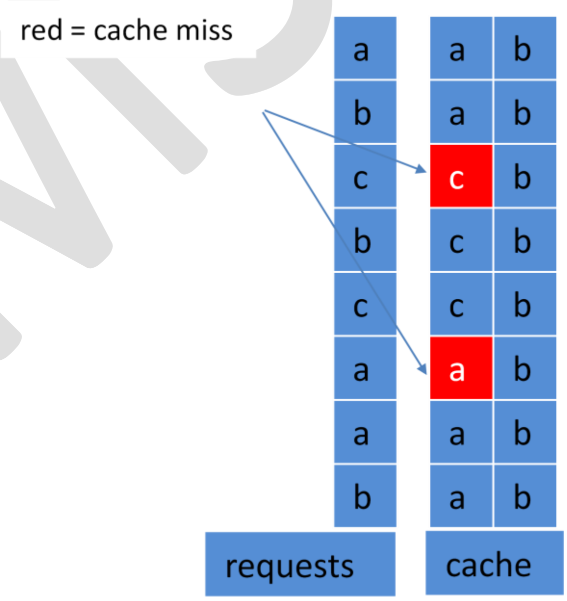
3. Optimal Caching

Caching is a general term for the process of storing a small amount of data in a fast memory so as to reduce the amount of time spent interacting with a slow memory.
For caching to be as effective as possible, it should generally be the case that when you go to access a piece of data, it is already in the cache. To achieve this, a *cache maintenance* algorithm determines what to keep in the cache and what to evict from the cache when new data needs to be brought in.

- We consider a set *U* of *n* pieces of data stored in *main memory*.
- We also have a faster memory, the *cache*, that can hold $k < n$ pieces of data at any one time.
- We will assume that the cache initially holds some set of *k* items.
- A sequence of data items *D* = *d*1, *d*2, . . . , *dm* drawn from *U* is presented to us—this is the sequence of memory references we must process— and in processing them we must decide at all times which *k* items to keep in the cache.
- When item *di* is presented, we can access it very quickly if it is already in the cache; otherwise, we are required to bring it from main memory into the cache and, if the cache is full, to *evict* some other piece of data that is currently in the cache to make room for d_i.
- This is called a *cache miss*, and we want to have as few of these as possible.
- *Goal is to produce Eviction schedule that minimizes number of cache misses.*

Example

- k = 2, initial cache = ab,
  requests: a, b, c, b, c, a, a, b.
- Optimal eviction schedule: 2 cache misses.



red = cache miss

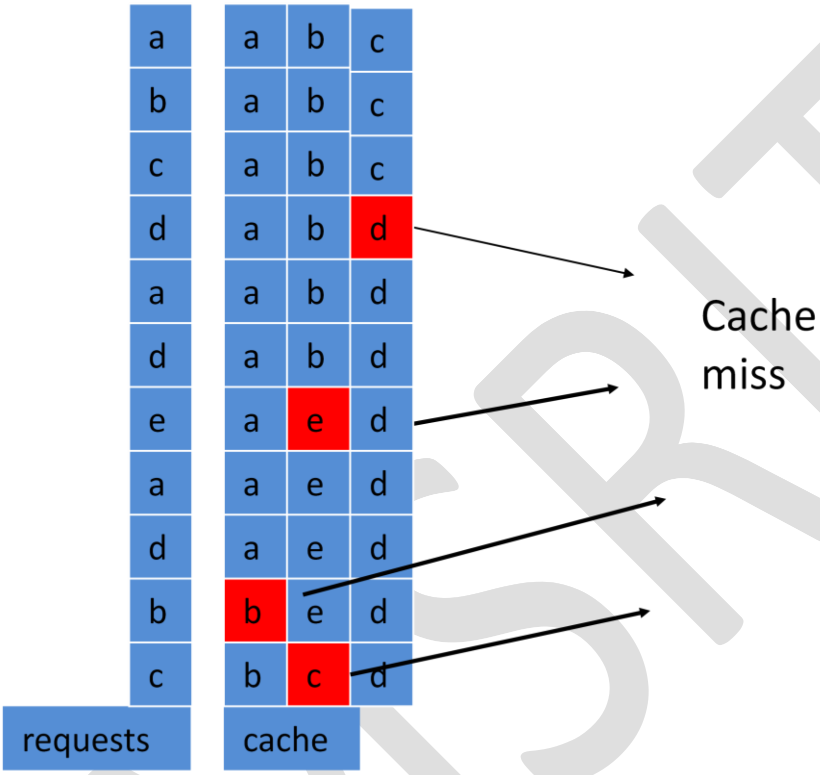
requests	cache	
a	a	b
b	a	b
c	🟥 c	b
b	c	b
c	c	b
a	🟥 a	b
a	a	b
b	a	b

Algorithm

```
When di needs to be brought into the cache,
      evict the item that is needed the farthest into the future
```

Example

a, b, c, d, a, d, e, a, d, b, c
k=3



a	a	b	c
b	a	b	c
c	a	b	c
d	a	b	d
a	a	b	d
d	a	b	d
e	a	e	d
a	a	e	d
d	a	e	d
b	b	e	d
c	b	c	d
requests	cache		

Cache miss