# EECS 660: FUNDAMENTALS OF COMPUTER ALGORITHMS

MODULE IV: GREEDY ALGORITHMS, PART II

## DISCLAIMER

## ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from ***Algorithm Design*** 4th edition, by Jon Kleinberg and Eva Tardos.

## GREEDY ALGORITHMS

- In this module, we will continue the discussion of greedy algorithms, by introducing three scheduling problems:
  - interval scheduling
  - scheduling to minimize maximum lateness
  - optimal caching

# INTERVAL SCHEDULING

- The interval scheduling problem: Given a single-threaded CPU server, and a set of jobs $J$ that need to be scheduled. Each job $j_i \in J$ has an associated start time $s_i$ and finishing time $f_i$. Find the maximal subset $J' \subseteq J$ such that for any two jobs $j_i, j_j \in J'$ either $f_i \leq s_j$ or $f_j \leq s_i$.

- In layman terms: Two jobs cannot be executed at the same time since we only have a single-threaded server. Schedule as many jobs as possible.

## INTERVAL SCHEDULING

- Intuitively, we can first sort the jobs based on their starting time (and break ties by their finishing time). Once the jobs are sorted by time, we can then schedule tasks one-by-one.

- Which math objective should we optimize at each step according to the greedy scheme?
  - at each given time point, we may need choose from a set of mutually-conflicting jobs
  - since the final math objective is to maximize $|J'|$, picking any job from the mutually-conflicting set will contribute equally to our math objective (+1 job scheduled)
  - in this case, we should probably pick the task that ends as early as possible, such that we can free up the server as early as possible for more jobs

## INTERVAL SCHEDULING

- So, we can come up with the following algorithm:

```
Let $J$ be the set of all jobs, and let $A$ be empty;
Sort jobs in $J$ with ascending finishing time;
foreach  $j_i \in J$ is not empty
        Add $j_i$ to $A$;
        Delete all request from $J$ that conflict with $j_i$
endwhile
Return $A$
```

## INTERVAL SCHEDULING

- We will try to prove that the simple algorithm actually works.
- We will use the same idea as illustrated in the shortest path problem and the minimum spanning tree problem:
  - we compare two solutions, one is produced by the greedy algorithm, another is an arbitrary one with at least one non-greedy choice
  - if we can show the greedy solution is always better than or as good as an arbitrary non-greedy solution, then we can prove the greedy algorithm is correct

## INTERVAL SCHEDULING

- Let $A = \{j_1, j_2, \dots\}$ be the set of jobs produced by the greedy algorithm, and let $B = \{j'_1, j'_2, \dots\}$ be an arbitrary solution that is non-greedy (i.e., with at least one task differs from $A$).

- <u>Lemma</u>: For each $i \leq |B|$, $f_{j_i} \leq f_{j'_i}$. (That is to say, the time that needs to finish the first $i$ jobs in $A$ is shorter than the time that needs to finish the first $i$ jobs in $B$.)

# INTERVAL SCHEDULING

- Proof: (We will prove this using math induction.)

(Base case): The statement is true when $i = 1$, this is because the greedy algorithm always selects the job that finishes the earliest.

(Induction hypothesis): We assume it is also true when $i = k$, where $1 \leq k < |B|$. That is, we assume $f_{j_k} \leq f_{j'_k}$. (Now we need to derive that $f_{j_{k+1}} \leq f_{j'_{k+1}}$.) Because $B$ is a valid schedule, we must have $f_{j_k} \leq f_{j'_k} \leq s_{j'_{k+1}}$. In this case, $j'_{k+1}$ is also a valid choice to be included in $A$ by the greedy algorithm. If the greedy algorithm picks $j'_{k+1}$ as its next job, then $f_{j_{k+1}} \leq f_{j'_{k+1}}$. If it does not pick $j'_{k+1}$, it means that there exists another valid valid $j_x$ such that $f_{j_x} \leq f_{j'_{k+1}}$. In either case, we have $f_{j_{k+1}} \leq f_{j'_{k+1}}$.

(Conclusion): Hence, for each $i \leq |B|$, $f_{j_i} \leq f_{j'_i}$.

# INTERVAL SCHEDULING

- Proposition: the job set $A$ returned by the greedy algorithm is optimal.
- Proof: (We can prove this by contradiction.) If $A$ is not an optimal set, then it means $|A| < |B|$ for some conflict-free job set $B$. Let $|A| = n$. It means that there exists a job $j'_{n+1} \in B$. By the previous lemma, we know that $f_{j_n} \leq f_{j'_n} \leq s_{j'_{n+1}}$. Therefore, $j'_{n+1}$ does not conflict with $j_n \in A$ and should have been included in $A$, which contradicts with the fact that $|A| = n$. Therefore, $A$ must be an optimal set.

# INTERVAL SCHEDULING

- Time complexity analysis:

```
Let $J$ be the set of all jobs, and let $A$ be empty;          // $O(1)$ time
Sort jobs in $J$ with ascending finishing time;                // $O(nlogn)$ time
foreach  $j_i \in J$ is not empty                              // $O(n)$ iterations
        Add $j_i$ to $A$;                                      // $O(1)$ time
        Delete all request from $J$ that conflict with $j_i$   // more detailed analysis?
Return $A$
```

## INTERVAL SCHEDULING

- A simple strategy is to check each subsequent job (recall that we have sorted the tasks based on their finishing time).
  - in the worst-case scenario, we may need to check all subsequent jobs (imagine we have a job $j$ where $s_j = 0$ and $f_j = \infty$; it will be sorted last but conflict with every other job)
  - so, this operation will require $O(n)$ time, and the overall time complexity will become $O(n^2)$
  - can we do better? (ideally anything below $O(\log n)$, since sorting already takes $O(n \log n)$)

# INTERVAL SCHEDULING

- We can circumvent the problem (that we have the last task conflicting with every other task) by constructing another sorted list by ascending start time. In this case, the job $j$ in the previous example will be sorted first instead of last.

- Given a job $j_i$ that has just been added to $A$, we will mask all its subsequent jobs (in the start-time sorted array) as invalid, until we reach a job whose start time is behind $f_{j_i}$.
  - we will spend an extra of $O(n \log n)$ time to sort the jobs based on start time (but within the original time complexity)
  - for each task, it is either included or masked; since either operation only takes $O(1)$ time, we will only need $O(n)$ time throughout the entire algorithm

- So, the overall time complexity is $O(n \log n)$.

# MINIMIZED MAXIMUM LATENESS

- The scheduling to minimized lateness problem: Given a single-threaded CPU server, and a set of jobs $J$ that need to be scheduled. Each job $j_i \in J$ has an execution time $t_i$ and a deadline $d_i$. Given a conflict-free schedule represented by a set of start time $S = \{s_1, s_2, \dots\}$, define the lateness for $j_i$ as $l_i = s_i + t_i - d_i$. Find the schedule $S$ to minimize $\max_i \{l_i\}$. (We want the schedule to be as fair as possible.)

- The maximized lateness problem differs from the interval scheduling problem in terms of:
  - we have to schedule all tasks, rather than a subset of them
  - the scheduling of each job is more flexible: it does not need to be started at a specific time
  - the math object is to minimize the lateness, i.e., how long a task gets delayed in the schedule

## MINIMIZED MAXIMUM LATENESS

- Intuitively, we wish to minimize the maximal lateness among all jobs. A job will suffer from high lateness if it has an early deadline but is schedule to execute late.

- So, a simple greedy algorithm is to schedule the tasks according to their deadlines.
  - sort the jobs according to their deadlines
  - schedule one-by-one
  - we will skip the time complexity analysis since it is obviously $O(nlogn)$

## MINIMIZED MAXIMUM LATENESS

- We will focus on proving its correctness.
- We will adopt a similar proving technique: we will compare the schedule $A$ produced by the greedy algorithm with another hypothetical optimal non-greedy schedule $\mathcal{O}$ and show that $A$ is always better than or as good as $\mathcal{O}$.

## MINIMIZED MAXIMUM LATENESS

- We shall first define **inversion**: two scheduled jobs are called an inversion if the two jobs have different deadlines and the job with the later deadline gets executed earlier.

- Note that we did not specify whether the inversion is adjacent or long-range. However, we do note that these two types of inversions are equivalent, as any long-range inversion can be modeled by a series of adjacent inversions.

## MINIMIZED MAXIMUM LATENESS

- Lemma: A finite list of jobs can be sorted via a finite number of adjacent swaps.
- Proof: (We will prove it using math induction.)

(Base case): The lemma is true when we have 2 jobs. If the two jobs are an inversion, we can convert them into a sorted list with on adjacent swap.

(Induction hypothesis): We assume that we can sort $k \geq 2$ jobs with only adjacent swaps. Given $k + 1$ jobs, we will first sort the first $k$ jobs with only adjacent swaps. Then, we will keep performing adjacent swaps on the last job until we find its correct position among the first $k$ jobs. The number of adjacent swaps we need is clearly bounded by $k$.

(Conclusion): Hence, any job can be sorted via a finite number of adjacent swaps

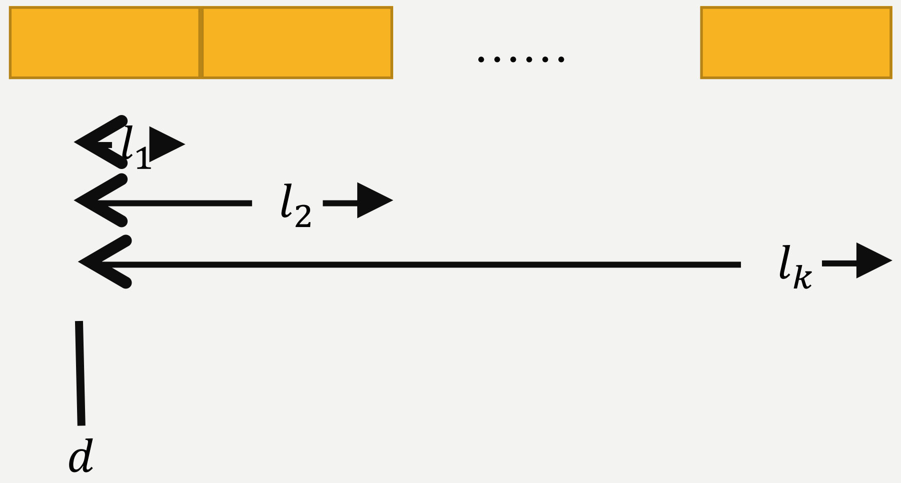
## MINIMIZED MAXIMUM LATENESS

- Lemma: There is an optimal solution with no idle time.
- Proof: Assume that there is an optimal solution with idle time. We can remove the idle time by scheduling the rest of the jobs earlier. No job will be finished later than the original optimal schedule. Hence, the resulted schedule without idle time is also an optimal solution.

## MINIMIZED MAXIMUM LATENESS

- Lemma: All schedules with no inversions and no idle time have the same maximized lateness.
- Proof: For two schedules with no inversions and no idle time, they may only differ from each other in the execution order of jobs with the same deadline. Because no inversion is allowed, all jobs with the same deadline will be executed consecutively. The maximum lateness of these jobs thus only depends on the finish time of the last job and is independent of the ordering of these jobs. In this case, all ordering of these jobs will result in the same maximized lateness.

# MINIMIZED MAXIMUM LATENESS

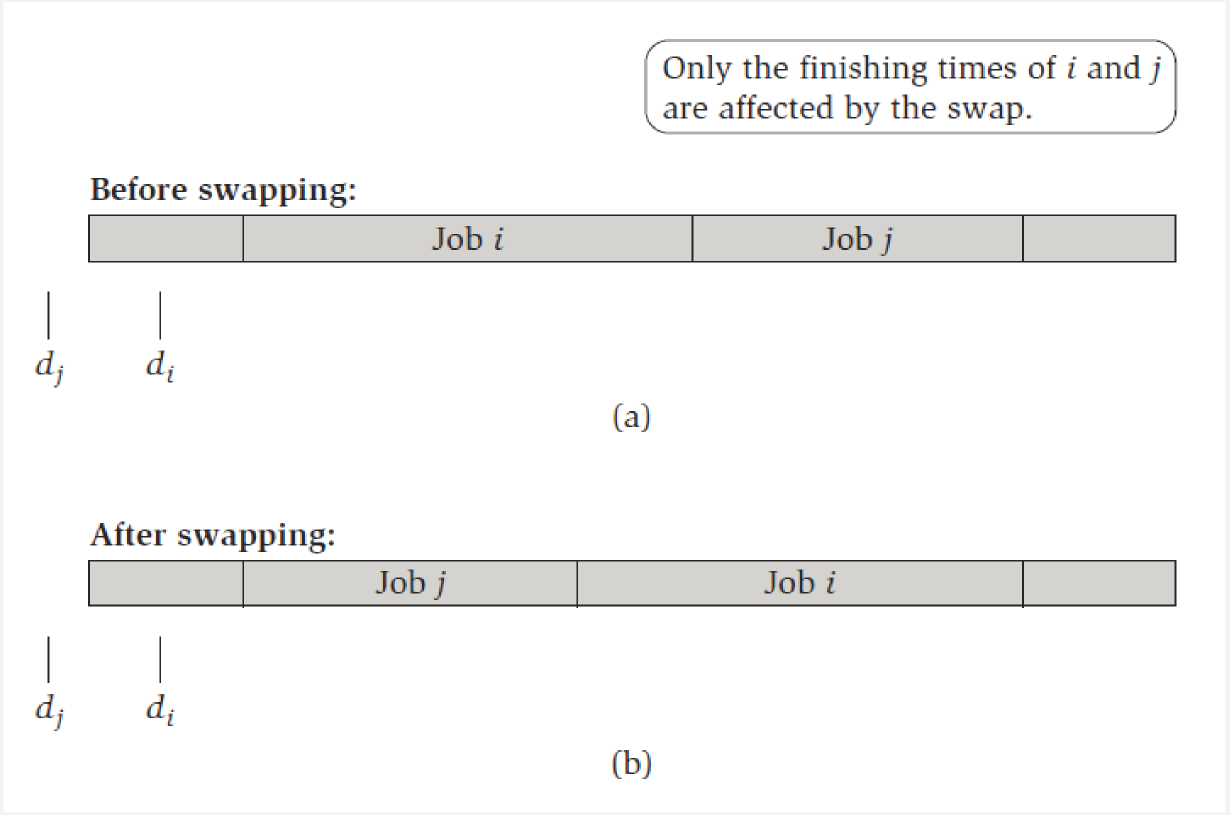


## MINIMIZED MAXIMUM LATENESS

- Lemma: There exists an optimal schedule that has no inversions and no idle time.
- Proof: (We will use prove by contradiction.) Let's assume that all optimal schedules have either some inversions or idle time. Per the previous lemma, we can simply remove the idle time for any schedule without increasing its associated maximum lateness. Hence, we only discuss optimal schedules that contain only inversions. Let $\mathcal{O}$ be an optimal schedule with at least one inversion. We can clearly sort $\mathcal{O}$ into a schedule $A$ with no inversion with only swapping adjacent inversions (per a previously proven lemma).

# MINIMIZED MAXIMUM LATENESS

- Proof (cont.): Now, we will prove that swapping adjacent inversions does no increase the maximum lateness. Consider the following scenario:



# MINIMIZED MAXIMUM LATENESS

- Proof (cont.): Since $i, j$ are an adjacent inversion, we have $d_i > d_j$ per definition. With the swap, $j$ is scheduled earlier and we will not increase the maximum lateness if it is attributed by $j$. The only concern is that job $i$ is postponed and the new schedule may increase its lateness. Let the new lateness of $i$ be $l'_i$. Note that in the new schedule, job $i$ will be finished the same time as $j$ in the old schedule. Now, we have $l'_i = f'_i - d_i = f_j - d_i < f_j - d_j = l_j$. It means that the lateness of job $i$ in the new schedule remains smaller than the lateness of job $j$ in $\mathcal{O}$. In this case, after swapping, neither the lateness of jobs $i$ nor $j$ will surpass $l_j$. So, the swapping will not increase the maximal lateness of $\mathcal{O}$. It further means that the schedule $A$ converted from $\mathcal{O}$ will not have a higher maximum lateness than $\mathcal{O}$, and $A$ should also be an optimal schedule. Clearly, $A$ is sorted and will not contain any inversion, hence $A$ will be an optimal schedule with no inversion and no idle time.

## MINIMIZED MAXIMUM LATENESS

- Proposition: The greedy algorithm finds an optimal schedule that has the minimized maximum lateness.
- Proof: Since there exists an optimal schedule with no inversion nor idle time, and all schedules with no inversion nor idle time have the same maximized lateness, all schedules with no inversion nor idle time have the minimized maximum lateness. The greedy algorithm produces a schedule with no inversion and no idle time, which also has the minimized maximum lateness..

## OPTIMAL CACHING

- Modern computer architecture: CPU, main memory, hard disk, internet (cloud)
  - the closer to the CPU (higher hierarchy), the faster we can access the data
  - the farther to the CPU (lower hierarchy), the more data we can store

- In modern computer architecture, caching is a popular scheme to exploit data locality (consecutive data used by consecutive instructions).
  - we will guess what data will be used in the future, and pre-load data from lower to higher hierarchy
  - and we will maintain the data in cache, hoping the data will be use by near-future instructions
  - in this way we can speedup the overall running time of a program

## OPTIMAL CACHING

- Because the cache size is rather limited, and in many cases a program may require more data than the cache can hold. In other words, we will be needing data that is not preloaded in cache; this called a **cache miss**.
  - a cache miss is problematic as it will require access of lower storage hierarchy, which is often very time consuming (in a practical sense, not in a time-complexity sense)
- A problem of interest is how to **minimize the number of cache misses**.
  - we will determine which data should be retained in cache
  - in other words, when the cache is full and new data needs to be brought in, we will determine which data to **evict** from cache

# OPTIMAL CACHING

- In an ideal scenario, we will know the what data is needed throughout the execution of a program (perspective)
  - in practical it is not possible, we only know what data has been accessed (retrospective)
  - but developing a caching strategy under the perspective scenario can bring insight for real-world applications
  - in real world, the perspective data access is predicted via machine learning model using retrospective data access as input features

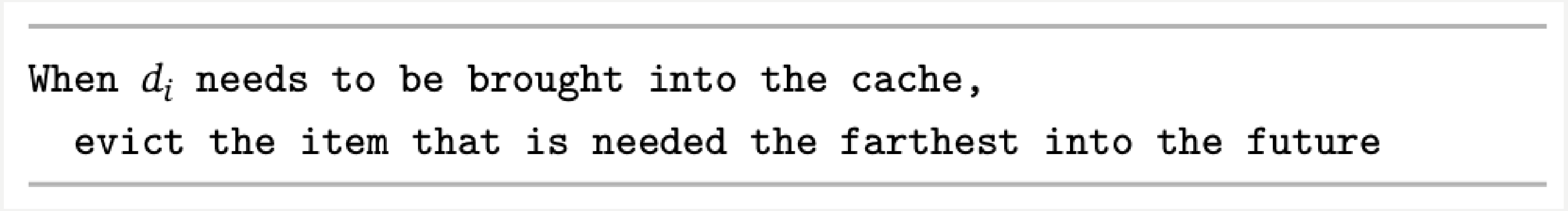
## OPTIMAL CACHING

- Optimal caching problem: Let $U$ be the set of all the data stored in the main memory, and $|U| = n$. Also, let $D = \{d_1, d_2, \ldots . d_m\}$ be the sequence of data access required by the program. Given a cache with capacity $k < n$, determine a data eviction plan that minimizes the number of cache misses.

## OPTIMAL CACHING

- Intuitively, our math objective is to minimize cache misses. So, we should try to keep the data that is needed in cache.

- If we have to evict one, we evict the one that is not going to be accessed in the near future. In other words, we shall evict the farthest-in-future (FF) data. We refer this greedy strategy as the FF strategy.



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

## OPTIMAL CACHING

- Now, we should prove that the simple algorithm actually work.
- We will again compare the greedy strategy with a hypothetical optimal solution.
- Note that we assume that the cache contains a fixed set of data $\{d_1, d_2, \ldots . d_k\}$ at the beginning no matter which eviction strategy we will apply.
  - if the cache is not full, we will not evict any data and simply bring in the data that is needed
  - we will focus on discussing our eviction strategy after $k$ data blocks being loaded

# OPTIMAL CACHING

- Another intuition we can apply when designing the eviction strategy is that, although we can evict data when no data needs to be brought in, we shall only evict data when a new data needs to be loaded into cache.
  - if we evict a data without the need for bringing a new data, the evicted data could be required and we would cause an unnecessary cache miss
  - we call this strategy **reduced schedule**
  - we will prove it more formally that any non-reduced schedules can be reduced without increasing the number of cache misses

# OPTIMAL CACHING

- For an arbitrary schedule $S$, we can reduce it by postponing the evictions until new data is brought in. Let the reduced schedule be $\bar{S}$.

- Note the difference between reduced schedule and greedy schedule:
  - reduced schedule discusses <u>when</u> should we evict data
  - greedy schedule discusses <u>which</u> data should we evict
  - and they are not conflicting (you can have a schedule that is both reduced and greedy)

## OPTIMAL CACHING

- Lemma: For any schedule $S$, its reduced version $\bar{S}$ will not cause more cache misses than it.
- Proof: Note that the $\bar{S}$ will evict a data when a new data is needed. So, no new data is needed between the corresponding eviction time in $S$ and the eviction time of $\bar{S}$. Therefore $\bar{S}$ will not cause more cache misses than $S$.

# OPTIMAL CACHING

- Lemma: Let $S$ be an arbitrary reduced schedule, and $S_{FF}$ be a farthest-in-future schedule. Also let $S$ and $S_{FF}$ have the same eviction decision through the first $j$ data blocks in the sequence. Then there exists a reduced schedule $S'$ that makes the same eviction decision as $S_{FF}$ for the $j + 1$ data blocks, and will incur no more cache misses than $S$ does (for the entire data access sequence).

# OPTIMAL CACHING

- First, note that the condition "let $S$ and $S_{FF}$ have the same eviction decision through the first $j$ data blocks" is feasible, when $j \leq k$ (where the cache is not full). And it serves as the base case for future math induction.

- We can view $S$ as a hypothetical non-greedy optimal schedule (since we did not specify any requirement for it), and $S'$ be the greedy solution. We are going to show that the greedy solution always outperforms the non-greedy optimal one.

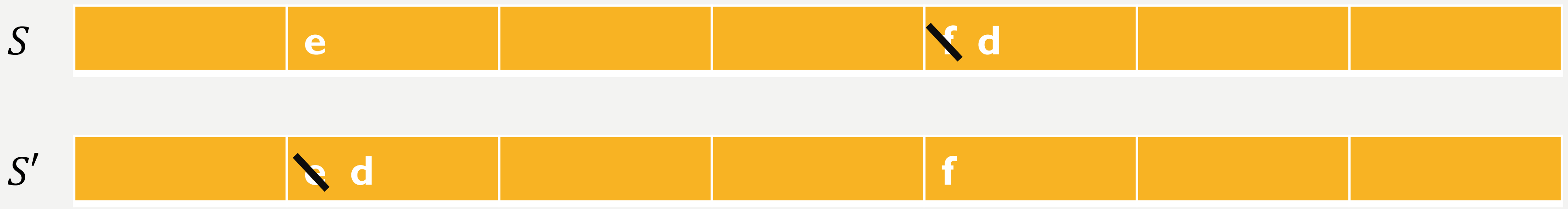
- The tricky part to show that "$S'$ will incur no more cache misses than $S$ does (for the entire data access sequence)" is through proving that, after making the decision for the $j + 1$th data, we can always convert $S'$ back to $S$ without incurring any cache misses. And the base case "let $S$ and $S_{FF}$ have the same eviction decision through the first $j$ data blocks" will be true again to continue the induction.

# OPTIMAL CACHING

- Proof: (Note that the constraint for $S'$ is that is should always make the same decision as $S_{FF}$ for the $j+1$th data.) Because both $S$ and $S_{FF}$ have the same eviction decision through the first $j$ data blocks, they have the same cache content. For the $j+1$th data (say $d$), if it is not a cache miss, nothing needs to be done and $S$ and $S_{FF}$ will still have the same cache content. We can thus make $S'$ the same as $S_{FF}$ (making $S'$ the same as $S_{FF}$). Otherwise, if $d$ incurs a cache miss but $S$ and $S_{FF}$ make the same eviction decision, we can also make $S'$ the same as $S_{FF}$ (making $S'$ the same as $S_{FF}$).

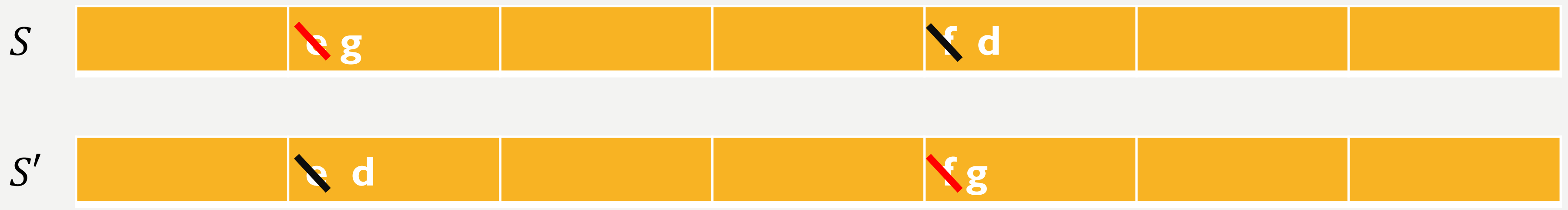
# OPTIMAL CACHING

- Proof: Now, consider case where $d$ incurs a cache miss but $S$ and $S_{FF}$ do not make the same eviction decision. Let $S$ evicts $f$ and $S_{FF}$ evicts $e \neq f$. (We will show that even $S'$ makes a different decision than $S$, we can always convert the cache content of $S'$ back to the same as $S$, without incurring any cache miss.) From the $j + 2$th data and onward, $S'$ will behave exactly like $S$ until one of the following two situations happen.



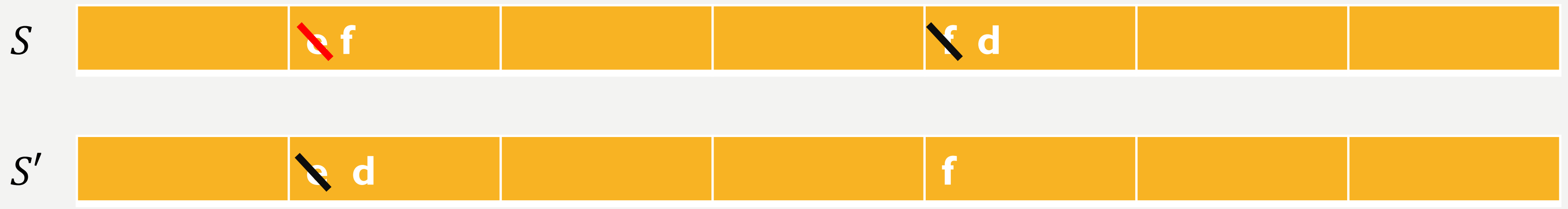
# OPTIMAL CACHING

- Proof: (case 1) There is a request of data $g \neq e, f$, and $S$ decides to evict $e$ to make room for it, we will let $S'$ evict $f$. Then the cache content of $S'$ and $S$ will become identical again. (Note that if $S$ decides to evict $h \neq e$, then we will make $S'$ evict $h$ as well; it remains the case that $S'$ and $S$ only differ in $e, f$). Both $S'$ and $S$ incur one eviction.



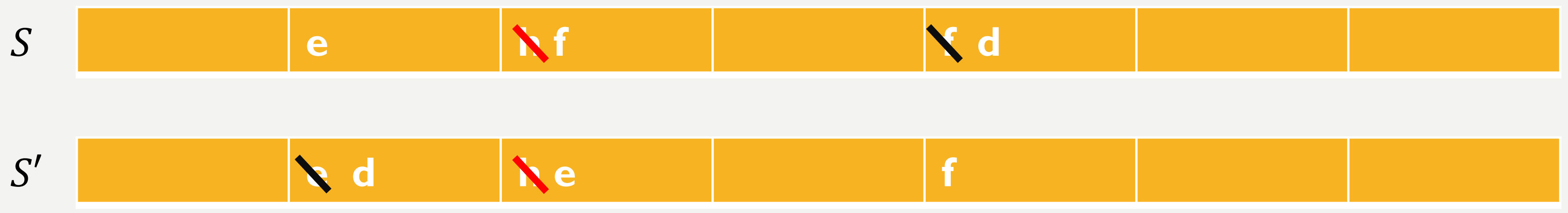
# OPTIMAL CACHING

- Proof: (case 2) There is a request for data $f$ and $S$ evicts $h$. If $h = e$ then we do not need to do anything. $S$ incurs one eviction and $S'$ incurs no eviction.



# OPTIMAL CACHING

- Proof: (case 2 cont.) There is a request for data $f$ and $S$ evicts $h$. If $h \neq e$ then we will have $S'$ to evict $h$ and bring in $e$. Note that such an eviction will make $S'$ a non-reduced schedule, but we can always postpone the eviction until needed without incurring additional cache misses (based on the already proved lemma). Both $S$ and $S'$ incur one eviction.



# OPTIMAL CACHING

- Proof: Because $S'$ evict $e$, which is the same as $S_{FF}$'s decision, based on the algorithm (farthest-in-future) we know that the request of $e$ will come after the request of $f$. So, either at least one of the above two cases will happen before the request of $f$ and allow us to convert the cache content of $S'$ into the cache content of $S$, or none will happen and no cache miss will be incurred. In either case, $S'$ will not incur more cache misses than $S$.

# OPTIMAL CACHING

- Proposition: The farthest-in-future strategy produces the optimal caching schedule.
- Proof: (We will prove it using math induction.)

(Base case): Let $S$ be a non-greedy optimal schedule, and let $S'$ be a reduced greedy FF schedule produced by our algorithm. It is clear that $S$ and $S'$ have the same cache content for the first $k$ data requests and $S'$ does not incur more cache misses than $S$ (when the cache is not full).

(Induction): Assume that $S$ and $S'$ have the same cache content and $S'$ does not incur more cache misses than $S$ until the $g \geq k$ data requests. Per the previous proven lemma, the schedule $S'$ which adopts the FF strategy will not incur more cache misses than $S$.

(Conclusion): Since $S$ is an optimal schedule, and the reduced FF schedule $S'$ will incur no more cache misses than $S$, thus $S'$ is also an optimal schedule.

# OPTIMAL CACHING

- Implementation:
  - we will parse the sequence $D$ first, and record for each data the times when it is requested.
  - for the cache, we will maintain a priority queue to effectively select the farthest-in-future data to evict

| $D$ | | | | | | |
|---|---|---|---|---|---|---|
| a | b | c | c | a | b | a |

| | | | |
|---|---|---|---|
| a | 0 | 4 | 6 |
| b | 1 | 5 | |
| c | 2 | 3 | |

## OPTIMAL CACHING

- We will need $O(m)$ time to parse the sequence $D$ and construct the table.
- Then, for each cache miss, we will spend $O(\log k)$ time to insert the new data into the priority queue.
- In the worst-case scenario, every data request is a cache miss, then the overall time complexity is $O(m) + O(m \log k) = O(m \log k)$.

## SUMMARY

- Greedy algorithm attempts to optimize the math objective in each step, with the consideration of only local optimality.
  - it is often very easy to design and implement, and it is also often efficient
  - in many cases it does not work
  - in some cases it does, but it needs to be proved
  - it is equivalent to show that for some problems, local optimality implies global optimality

## SUMMARY

- Classical algorithms studied:
  - shortest path
  - minimum spanning tree
  - interval scheduling
  - minimized maximum lateness
  - optimal caching