Greedy Algorithms

Activity Selection, Shortest Path, Minimal Spanning Tree, Huffman Codes, Knapsack Problem, Clustering, Set Cover,

Acknowledgement: The notes have drawn heavily from the following books: (1) Éva Tardös and Jon Kleinberg, "Algorithm Design", Addison-Wesley, (2) Michael T. Goodrich and Roberto Tamassia,"Algorithm Design and Applications", Wiley, (3) Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms", The MIT Press.

1

What is a Greedy Algorithm?

- In Wall Street (iconic movie of the 1980s), Michael Douglas gets up in front of a room full of stockholders and proclaims, "Greed . . . is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit."
- Wikipedia defines greed "As secular psychological concept, greed is an inordinate desire to acquire or possess more than one needs". A game like chess can be won only by thinking ahead: a player who is focused entirely on immediate advantage, is easy to defeat. But in many other games, such as Scrabble, it is possible to do quite well by simply making whichever move seems best now and not worrying too much about future consequences.
- This sort of myopic behavior is easy and convenient, making it an attractive algorithmic strategy (faster). *Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit*. Although such an approach can be disastrous for some computational tasks, there are many for which it is optimal.
- Locally optimal decisions are called **greedy**
  - Short sighted strategy (e.g., thinking only one move ahead in a game), usually easy, faster to implement
  - Generally efficient, sometimes provably optimal, but …
  - Do not always lead to a globally optimal solution
    - Sometimes, close enough to globally optimal anyway (approximate or near optimal solution)

Greedy Technique

- It is hard, if not impossible, to define precisely what is meant by a greedy algorithm.
- Basic steps to finding efficient greedy algorithms:
  - Start by finding a dynamic programming style solution
  - Prove that at each step of the recursion, the min/max can be satisfied by a "greedy choice" (greedy substructure)
  - Show that only one recursive call needs to be made once the greedy choice is assumed. This is often natural when all the recursive calls are made by the min/max.
  - Find the recursive solution using the greedy choice
  - Convert to an iterative algorithm if possible
- More generally, taking the direct approach:
  - Show the problem is reduced to a sub problem via a greedy choice
  - Prove there is an optimal solution containing the greedy choice
  - Prove that combining the greedy choice with an optimal solution of the remaining sub problem yields an optimal solution.
- Our first example is Interval Scheduling, also known as Activity Selection (some activities that use a resource in a non-sharable way, e.g., two classes cannot be scheduled in a single classroom when the class interval overlaps – there are many variations and extensions to this problem)

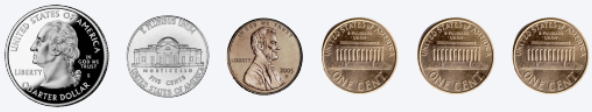
Characterization of Greedy Algorithms

- Greedy is a strategy that works well on optimization problems with the following characteristics:
  1. **Greedy-choice property**: A global optimum can be arrived at by selecting a local optimum.
  2. **Optimal substructure**: An optimal solution to the problem contains an optimal solution to subproblems.
- The second property *may* make greedy algorithms look like dynamic programming. However, the two techniques are quite different.

Coin Changing

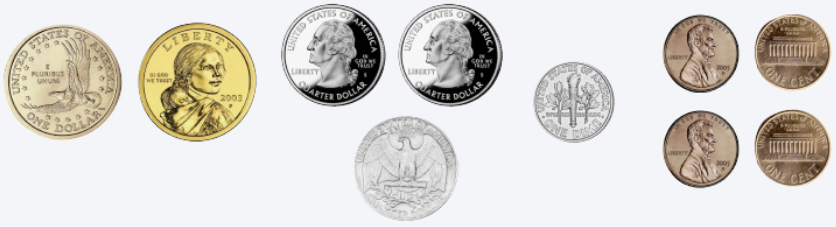
Goal. Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

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.



Cashier's algorithm

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

CASHIERS-ALGORITHM $(x, c_1, c_2, \ldots, c_n)$

SORT n coin denominations so that $c_1 < c_2 < \ldots < c_n$

$S \leftarrow \phi$ ⟵ set of coins selected

WHILE $x > 0$

  $k \leftarrow$ largest coin denomination c_k such that $c_k \leq x$

  IF no such k, RETURN "no solution"

  ELSE

    $x \leftarrow x - c_k$

    $S \leftarrow S \cup \{ k \}$

RETURN S

Q. Is cashier's algorithm optimal?

Properties of optimal solution

Property. Number of pennies $\leq$ 4.

Pf. Replace 5 pennies with 1 nickel.

Property. Number of nickels $\leq$ 1.

Property. Number of quarters $\leq$ 3.

Property. Number of nickels + number of dimes $\leq$ 2.

Pf.

- Replace 3 dimes and 0 nickels with 1 quarter and 1 nickel;
- Replace 2 dimes and 1 nickel with 1 quarter.
- Recall: at most 1 nickel.



Analysis of cashier's algorithm

Theorem. Cashier's algorithm is optimal for U.S. coins: 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 cashier's algorithm. ▪

k	c_k	all optimal solutions must satisfy	max value of coins $c_1, c_2, \ldots, c_{k-1}$ 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

Cashier's algorithm for other denominations

Q. Is cashier's algorithm for any set of denominations?

A. No. Consider U.S. postage: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

- Cashier's algorithm: 140¢ = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1.
- Optimal: 140¢ = 70 + 70.

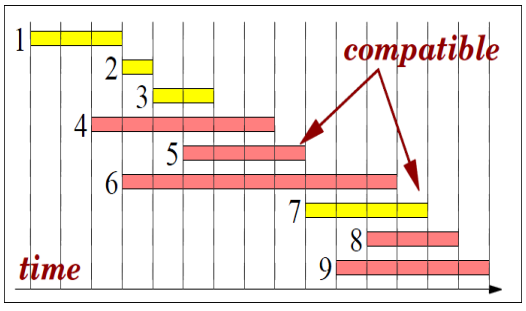


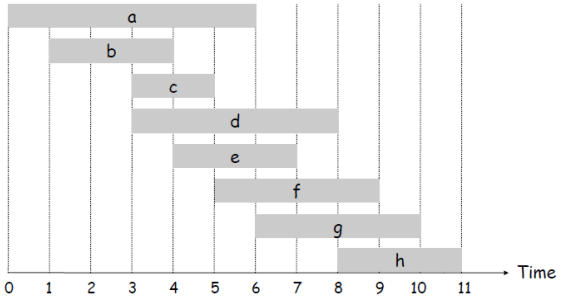
A. No. It may not even lead to a feasible solution if $c_1 > 1$: 7, 8, 9.

- Cashier's algorithm: 15¢ = 9 + ???.
- Optimal: 15¢ = 7 + 8.

Activity Selection (Interval Scheduling)

- **Problem Statement**: Let S = {1, 2, … n} be the set of activities that compete for a resource. Each activity i has its starting time s_i and finish time f_i with $s_i \leq f_i$, namely, if selected, i takes place during time $[s_i; f_i)$. No two activities can share the resource at any time point. We say that activities i and j are compatible if their time periods are disjoint. The activity-selection problem is the problem of selecting the largest set of mutually compatible activities *within a given time*.
  - Easy to see that this identical to scheduling intervals such that intervals do not overlap.





Greedy Algorithm, Interval Scheduling

- **Algorithm**: In this algorithm the activities are **first sorted according to their finishing time**, from the earliest to the latest, where a **tie can be broken arbitrarily**. Then the activities are **greedily** selected by going down the list and by picking whatever activity that is compatible with the current selection.
- **Run Time Analysis**:
  - Sorting takes O(n log n) time.
  - Second part takes O(n) time – scan a list sequentially, checking compatibility with the next job takes O(1) time, there are n jobs to scan.
- **Correctness Analysis**: Correctness means the output is a maximal set of mutually compatible activities. We use induction:
  - For the base case n = 1. Correctness trivially holds. For the induction case, let $n \geq 2$ and assume the claim holds for all values of n less than the correct one. Let p be the number of activities in each optimal solution for [1 … n – 1] and let q be the number for [1 … n]. Here $p \leq q$; why? It is because each optimal solution for [1 … n – 1] is a solution for [1 … n] [This is the greedy substructure.]
  - Is it possible $p = q - 2$? NO. Why? Let W be any optimal solution for [1 … n]. Let $W' = W - \{n\}$ if W contains {n} and $W' = W$ otherwise. Then W ′ does not contain {n} and is a solution for [1 … n – 1]. This contradicts the assumption that optimal solutions for [1 … n – 1] have p activities.

Optimality Proof

- We must first note that the greedy algorithm always finds some set of mutually compatible activities.
- (Case 1): Suppose p = q. Then each optimal solution for [1 … n – 1] is optimal for [1 … n]. By our induction hypothesis, when n – 1 has been examined an optimal solution for [1 … n – 1] has been constructed. So, there will be no addition after this; otherwise, there would be a solution of size > q. So, the algorithm will output a solution of size p, which is optimal.
- (Case 2): Suppose p = q – 1. Then each optimal solution for [1 … n] contains n. Let k be the largest integer i, $1 \leq i \leq n - 1$, such that $f_i \leq s_n$. Since $f_1 \leq ... \leq f_n$, for all i, $1 \leq i \leq k + 1$, i is compatible with n, and for all i, $k + 1 \leq i \leq n\text{-}1$, i is incompatible with n. This means that each optimal solution for [1 ... n] is the union of {n} and an optimal solution for [1 ... k]. So, each optimal solution for [1 ... k] has p activities. This implies that no optimal solutions for [1 ... k] are compatible with any of $k + 1$, ..., $n - 1$.

  Let W be the set of activities that the algorithm has when it has finished examining k. By our induction hypothesis, W is optimal for [1, …, k]. So, it has p activities. The algorithm will then add no activities between k +1 and n – 1 to W but will add {n} to W. The algorithm will then output $W \cup \{n\}$. This output has q = p+1 activities, and thus, is optimal for [1, …, n].

Formulation using Dynamic Programming

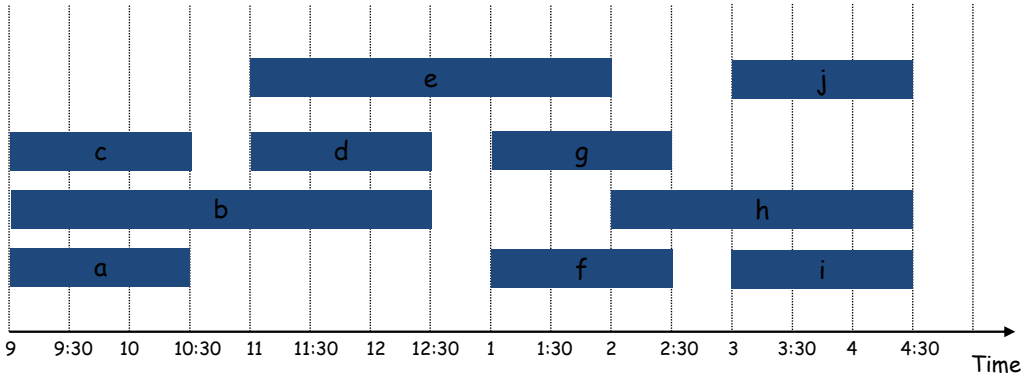
- Will do later.

Greedy Substructure

- Let us be a bit formal. The problem is: Given a set of activities $S = \{a_1, \ldots, a_n\}$, where a_i starts at time $s_i \geq 0$ and finishes at time $f_i > s_i$, find a maximal subset $A \subseteq S$ such that for distinct activities either $s_i \geq f_j$ or $s_j \geq f_i$.
- Convenience for notations:
  - Let a_0 be an imaginary activity finishing at time 0.
  - Let a_{n+1} be an imaginary activity starting at time ∞
  - $S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$
  - Observe that $S_{0,n+1}$ contains all activities.
- Let $f_m = \min\{f_k : a_k \in S_{ij}\}$, i.e., activity a_m has the earliest finish time in S_{ij}
  - Claim 1: a_m is used in some maximal solution for the activities in S_{ij} : Suppose a_k is the first activity in some maximal solution – it can be safely removed and replaced by a_m.
  - Claim 2: $S_{im} = \emptyset$: Nothing else starting after a_i finishes before a_m.
- Thus, it is always safe to include a_m, and solve the remaining problem for S_{mj} only.

Activity Selection (Multiple resources) or Interval Partitioning

- We extend the problem to multiple (identical) resources (to accommodate all activities)
- **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.

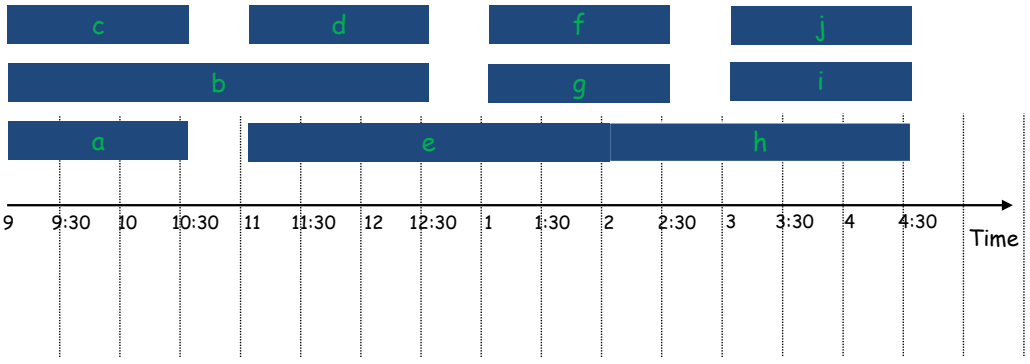


15

Interval Partitioning

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

- Ex: This schedule uses only 3.

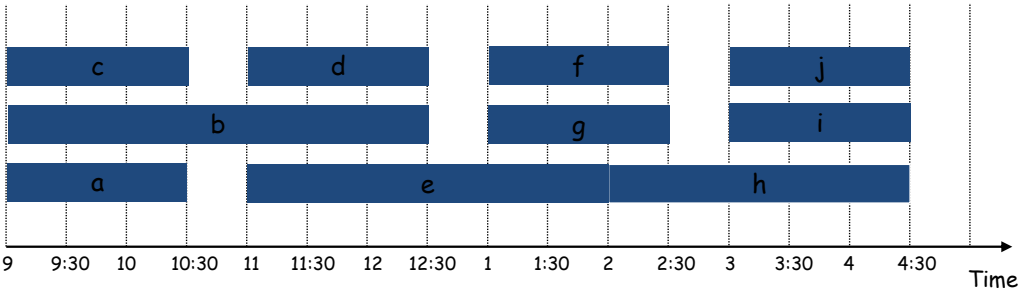


Interval Partitioning: Lower Bound on Optimal Solution

- Def. The depth of a set of open intervals is the maximum number that contain any given time.
- Key observation. Number of classrooms needed $\geq$ depth.
- Ex: Depth of schedule below = 3 $\Rightarrow$ schedule below is optimal.

a, b, c all contain 9:30

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



17

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.

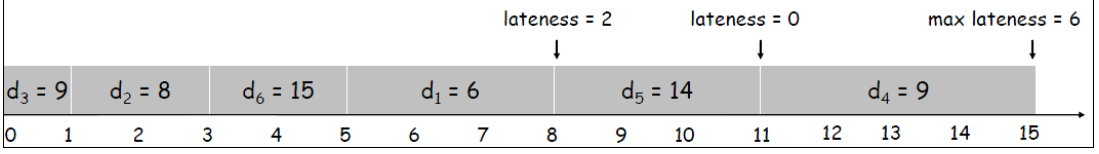
- **Theorem**. Greedy algorithm is optimal.
- **Proof.**
  - Let d = number of classrooms that the greedy algorithm allocates.
  - Classroom d is opened because we needed to schedule a job, say j, that is incompatible with all d – 1 other classrooms.
  - Since we sorted by start time, all these incompatibilities 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.

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. [No restriction on start time]
  - If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
  - Lateness: $\lambda_j = \max \{ 0, f_j - d_j \}$.
  - Goal: schedule all jobs to minimize maximum lateness $L = \max_j \lambda_j$.

Ex:

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



Minimizing Lateness: Greedy Algorithms

- Greedy template. Consider jobs in some order.
  - [Shortest processing time first] Consider jobs in ascending order of processing time t_j.
  - [Earliest deadline first] Consider jobs in ascending order of deadline d_j.
  - [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

Minimizing Lateness: Greedy Algorithms

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

	1	2
t_j	1	10
d_j	100	10

counterexample

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

	1	2
t_j	1	10
d_j	2	10

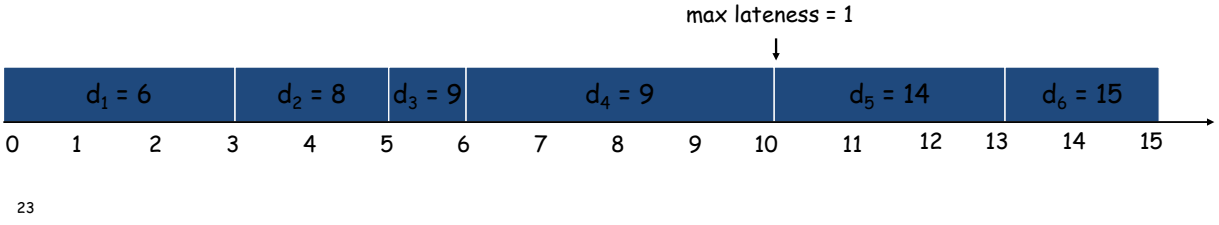
counterexample

Minimizing Lateness: Greedy Algorithm

- Greedy algorithm. Earliest deadline first.

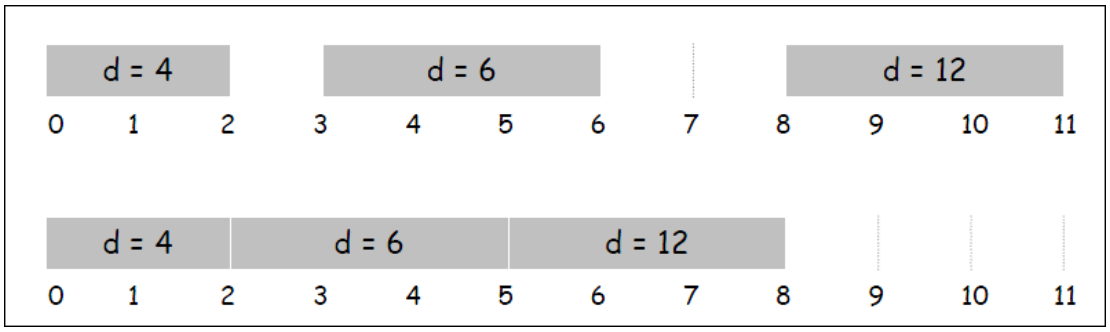
```
Sort n jobs by deadline so that d_1 ≤ d_2 ≤ … ≤ d_n

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



Scheduling to Minimize Lateness

- **Observation**. There exists an optimal schedule with no idle time. [Remember, No restriction on start time]

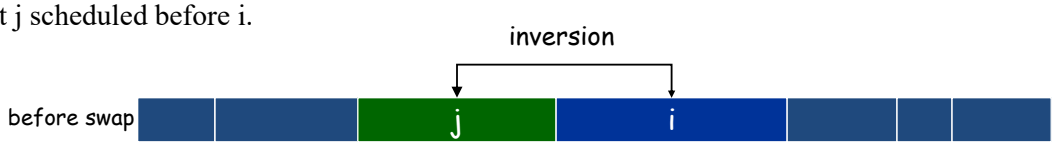


- **Observation**. The greedy schedule has no idle time. Why? Obvious

- We want to prove that earliest-deadline-first greedy algorithm is optimal. How? We use exchange argument – Change into greedy schedule without losing optimality. Things to ponder:
  - What do we know about the greedy schedule?
  - How can we change the optimal to be more like that without losing optimality?

Minimizing Lateness: Inversions

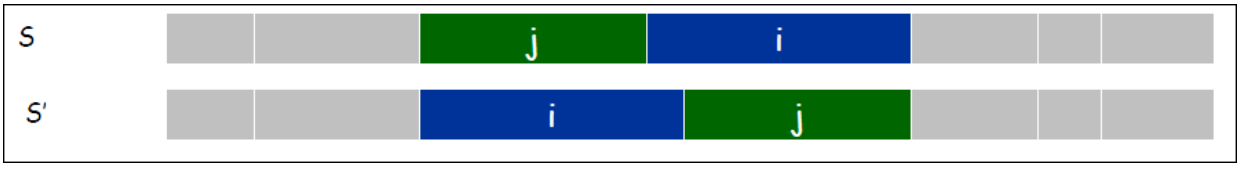
- **Definition**. An inversion in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i.



- **Observation**. Greedy schedule has no inversions. Why

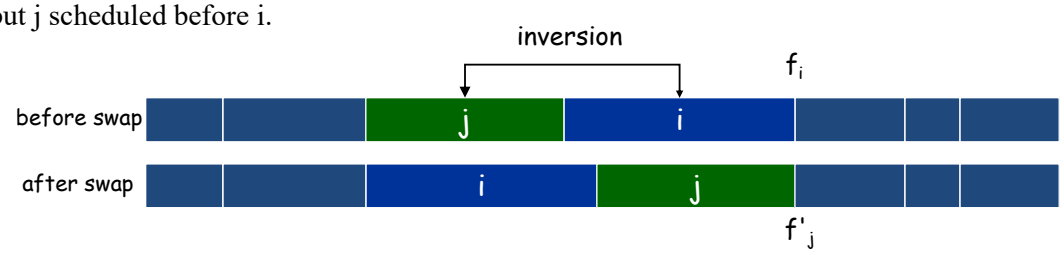
- **Question**. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively. Why? [Recall: jobs sorted in ascending order of due dates]
  - Only difference is an "inversion" of i and j with equal deadline (d_i=d_j). Maximum lateness of i and j is only influenced by last job (f_i - d_i). Maximum lateness of i and j is the same if i and j are swapped.
- **Observation**: All schedules without inversions have same lateness.
- **Question:** If a schedule (with no idle time) has an inversion, how can we find it?

going through schedule, at some point deadline decreases



Minimizing Lateness: Inversions

- **Definition**. An inversion in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i.



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

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

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

Minimizing Lateness: Analysis of Greedy Algorithm

- Theorem. Greedy schedule S is optimal.
- Proof. (by contradiction)
- Suppose S is not optimal.
- Define S* to be an optimal schedule that has the fewest number of inversions (of all optimal schedules) and has no idle time.
- Clearly S≠S*.
  - If S* has no inversions, then maxlate(S) = maxlate(S*). Contradiction.
  - 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*
- So S is an optimal schedule.



Greedy has no inversions. All schedules without inversions have same lateness.

*This proof can be found on pages 128 – 131.

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

  - Greed is myopic; at every iteration, the algorithm chooses the best morsel it can swallow, without worrying about the future.
  - Objective function does not explicitly appear in greedy algorithm! Hard, if not impossible, to precisely define "greedy algorithm."

Example Exercise

Planning a mini-triathlon:

1. swim 20 laps (one at a time)
2. bike 10 km (can be done simultaneously)
3. run 3 km (can be done simultaneously)

expected times are given for each contestant

Def. The completion time is the earliest time all contestants are finished.

Q. In what order should they start to minimize the completion time?

Q. Proof that this order is optimal (minimal).

Ex.

	1	2	3	← contestant
s_j	3	2	4	← time required for swimming
b_j	5	4	6	← time required for biking
r_j	3	3	3	← time required for running

Variant: Scheduling to Minimizing Total Lateness

Outside our scope

Minimizing total lateness problem.

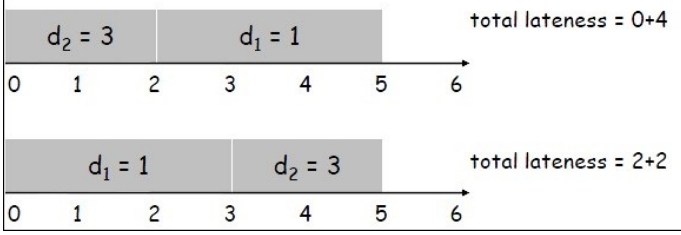
- Single resource processes one job at a time.
- Job j requires t_j units of processing time and is due at time d_j.

- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
- Lateness: $\ell_j = \max \{ 0, f_j - d_j \}$.
- Goal: schedule all jobs to minimize total lateness $L = \sum_j \ell_j$.

Ex:

	1	2	
t_j	3	2	← time required
d_j	1	3	← deadline

job number



No polynomial algorithm can compute optimal schedule (unless P=NP)

The Knapsack Problem and Greedy Algorithms

The Knapsack Problem is a central optimization problem in the study of computational complexity. We are presented with a set of n items, each having a value and weight, and we seek to take as many items as possible to maximize the total value, but with respect to a constraint that the total weight cannot exceed a pre-defined maximum weight. More formally, the Knapsack Problem is defined as:

> **Input:** Set S of n items $x_1, x_2, \ldots, x_n$, where items x_i has weight w_i and value v_i, for $1 \leq i \leq n$, and a number W.
>
> **Output:** Subset $S' \subseteq S$ such that (1) $\sum_{x_i \in S'} w_i \leq W$ and (2) $\sum_{x_i \in S'} v_i$ is maximum over all subsets that satisfy (1).

Condition (1) in the problem statement concerns feasibility, while Condition (2) concerns optimality over all feasible solutions.

Unfortunately, there is no known polynomial-time algorithm for the Knapsack Problem (it is NP-hard).

However, if we make a seemingly simple relaxation to one of the problem's constraints, we obtain a problem that has a polynomial time solution. Notice that in the Knapsack Problem, each item x_i is either not taken to be part of the solution, in which case it contributes 0 to the total weight and total value, or it is taken in its entirety, in which case it contributes w_i to the total weight and v_i to the total value. But what if we can take fractions of items? That is, what if are allowed to take 0.5 amount of item x_i and 0.37 amount of item x_j, and so on? This is known as the Fractional Knapsack Problem, and defined as follows.

> **Input:** Set S of n items $x_1, x_2, \ldots, x_n$, where items x_i has weight w_i and per-unit value v_i, for $1 \leq i \leq n$, and a number W.
>
> **Output:** $y_1, y_2, \ldots, y_n$ such that (1) for $1 \leq i \leq n$, $0 \leq y_i \leq w_i$, (2) $\sum_{1 \leq i \leq n} y_i \leq W$ and (3) $\sum_{1 \leq i \leq n} (y_i \cdot v_i)$ is maximum over all subsets that satisfy (1) and (2).

The Knapsack Problem and Greedy Algorithms

Please note here that v_i is now the per-unit value. This is why the problem seeks to maximize the sum of $y_i \cdot v_i$ terms (for example, if $y_3 = 2.4$, this means the solution includes 2.4 units of item 3, contribution a value of $2.4 \cdot v_3$).

This simple relaxation (from either taking the item in its entirety or not taking it at all to taking a fraction of it) turns the problem into a polynomially solvable one by a greedy algorithm as follows:

1. Sort the items in non-increasing order of their per-unit value v_i. Let the sorted list be $x_1, x_2, \ldots, x_n$;
2. Initialize W'—the total weight of the items included so far—to 0;
3. For $i \leftarrow 1$ to n
   (a) If $W' = W$
      i. $y_i \leftarrow 0$; // item i cannot be taken
      ii. $W' \leftarrow W' + y_i$;
   (b) Else
      i. If $(W - W') \geq w_i$
         A. $y_i \leftarrow w_i$; // all of item i is taken
         B. $W' \leftarrow W' + y_i$;
      ii. Else
         A. $y_i \leftarrow W - W'$; // a fraction of item i is taken
         B. $W' \leftarrow W' + y_i$;

32

The Knapsack Problem and Greedy Algorithms

This is an $O(n \log n)$ greedy algorithm. We now prove that it is correct; that is, that the algorithm above yields an optimal solution to the Fractional Knapsack Problem.

Proof: Assume the items sorted in non-increasing per-unit values are $x_1, x_2, \ldots, x_n$, and let $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be the solution computed by the algorithm. Consider now an optimal solution to the problem (not necessarily computed by the algorithm above): $O = \langle o_1, o_2, \ldots, o_n \rangle$, where o_i is the weight units of item i according to this optimal solution. The proof will now proceed as follows: If $y_i = o_i$ for every $1 \leq i \leq n$, then we are done, since the solution computed by the algorithm is optimal. If there exists at least one $1 \leq j \leq n$ such that $y_j \neq o_j$, we will then show that we can convert the optimal solution O into Y without affecting the total value, thus establishing that Y is also an optimal solution.

Assume without loss of generality that the knapsack can be filled with the available items (that is, a solution to the problem has a total weight the equals W; otherwise, we can "adjust" the value of W by subtracting from it the total weight of all elements). We know that

$$\sum_{i=1}^{n} o_i = W$$

since the o_i's are an optimal solution, and

$$\sum_{i=1}^{n} y_i = W$$

by construction of the algorithm (that algorithm fills the knapsack). If $y_i = o_i$ for every $1 \leq i \leq n$, then the solution computed by the algorithm is optimal, and the proof is established. Now, assume that the solution computed by the algorithm differs from the optimal solution we are considering, and let j be the smallest index ($1 \leq j \leq n$) such that $y_j \neq o_j$. Since the algorithm is designed to take for every item the maximum amount possible, it follows that $y_j > o_j$. Let $d = y_j - o_j$. Consider the solution $Q = \langle q_1, q_2, \ldots, q_n \rangle$ that we construct as follows:

The Knapsack Problem and Greedy Algorithms

- for every $1 \leq i < j$: $q_i = o_i$ (also $q_i = y_i$);
- $q_j = y_j$;
- For $i = j + 1$ to n
  - $d' \leftarrow \min(q_i, d)$;
  - $q_i \leftarrow q_i - d'$;
  - $d \leftarrow d - d'$;

While this might look complicated, the idea of constructing Q is simple: leave all elements up to the $j-1$-st item the same as those in O, replace the j-th item by y_j, and then decrease from the total weight of the remaining $n - j$ items the weight d that was added to item j in Q.

After this procedure is done, the total value of the solution Q is equal to the total value of O (make sure you understand why!). Therefore, solution Q' is also optimal, and it now agrees with the solution Y computed by the algorithm all the way to at least index j. If $Q = Y$, then we are done. If $Q \neq Y$, then we repeat the same process as above, but with Q taking the role of O (that is, find the smallest j such that $q_j \neq y_j$, and do the procedure above). After doing this, we obtain a solution that is optimal and identical to Y. Therefore, Y is optimal. □

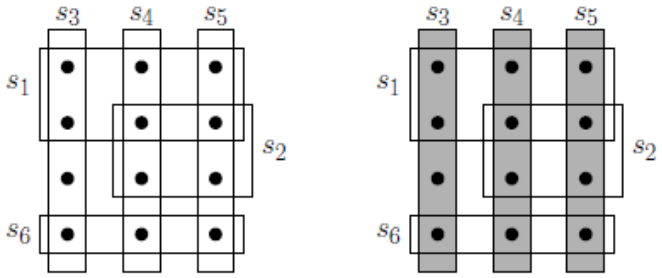
Think! The Knapsack Problem does not have a polynomial-time greedy algorithm (we stated above that it is NP-hard). But suppose you were not convinced and wanted to prove, similar to the proof above, that a greedy algorithm (e.g., take items in non-increasing order of their values) would solve the problem. Where does the proof break down?

Greedy Approximation: Set Cover

- **Set Cover**: An important class of optimization problems involves covering a certain domain, with sets of a certain characteristics. Many of these problems can be expressed abstractly as the set cover problem. We are given a pair $\sum = (X, S)$, called a set system, where $X = \{x_1, \ldots, x_m\}$ is a finite set of objects, called the universe, and $S = \{s_1, \ldots, s_n\}$ is a collection of subsets of X, such that every element of X belongs to at least one set of S. Set systems arise in many applications of science and engineering.
  - **Undirected Graph**: An undirected graph G = (V, E) is a set system where V constitutes the universe, and the edges E are subsets of cardinality two.
  - **Geometric set systems**: The universe consists of n points in space, and the sets are the subsets of points that are contained within some specified geometric shape (balls, cubes, rectangles, triangles, etc.)
  - **Wireless Coverage**: The universe consists all the locations on a college campus, and for each possible location of a wireless transmitter there is an associated region of the campus that is covered by placing a wireless transmitter at this location.

Set Cover (contd.)

- A fundamental question involving set systems is determining the smallest number of sets needed to cover the entire universe. A cover of S is defined to be a subcollection of sets whose union covers X. For example, in Fig. 1, the elements of X are the black circles, and the sets s_1; : : : ; s_6 are indicated by rectangles. In this case there exists a cover of size three, consisting of s_3, s_4, and s_5 (see figure 2). (The sets of this cover do not overlap, which is sometimes called an exact cover. In general, the sets of the cover are allowed to overlap.)



- Notice that the output of set cover is not a set, but rather a set of sets. If we think of the sets of S as being indexed by the integers from 1 to n, then we can think of a cover C more conveniently as a subset of {1, …, n}. This suggests the following definition.

Set Cover (contd.)

- Set Cover Problem: Given a set system $S = (X, S)$, where $S = \{s_1, \ldots, s_n\}$, compute a set $C \not\subset \{1, \ldots, n\}$ of minimum cardinality such that

$$X = \bigcup_{i \in C} s_i$$

- The set cover problem is a very important and powerful optimization problem. It arises in a vast number of applications. Determining the fewest locations to place wireless transmitters to cover the entire campus is an example. Unfortunately, the set cover problem is known to be NP-hard. We will present a simple greedy heuristic for this problem.
- Given an input instance $\Sigma = (X, S)$, let $O(\Sigma)$ denote an optimum cover (of minimum cardinality) and let $G(\Sigma)$ denote the cover produced by our greedy heuristic. Clearly, $G(\Sigma) \geq O(\Sigma)$. We say that G achieves an approximation ratio of ρ if $|G(\Sigma)| \leq \rho|O(\Sigma)|$, say, for any input Σ. Ideally, we would like ρ to be as small as possible.
- Unfortunately, the best that we can show for set cover is that ρ is a slowly growing function of $m = |X|$, and in particular $\rho = \ln m$.

Set Cover (contd.)

- Greedy Set Cover: A simple greedy approach to set cover works by at each stage selecting the set that covers the greatest number of uncovered elements. The set C contains the indices of the sets of the cover, and the set U stores the elements of X that are still uncovered. Initially, C is empty and U ← X. We repeatedly select the set of S that covers the greatest number of elements of U and add it to the cover.

```
Greedy-Set-Cover(X, S) {
    U = X                              // U stores the uncovered items
    C = empty                          // C stores the sets of the cover
    while (U is nonempty) {
        select s[i] in S that covers the most elements of U
        add i to C
        remove the elements of s[i] from U
    }
    return C
}
```

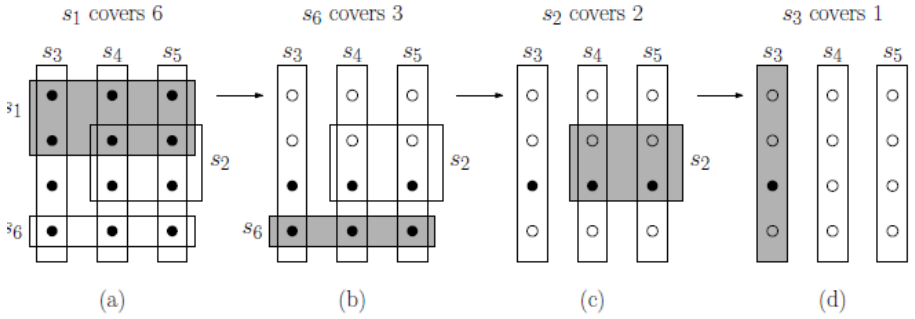
- We will not worry about implementing this algorithm in the most efficient manner. If we assume that U and the sets s_i are each represented as a simple list of elements of X (each of length at most m), then we can perform each iteration of the main while loop in time O(mn), for a total running time of $O(mn^2)$.

38

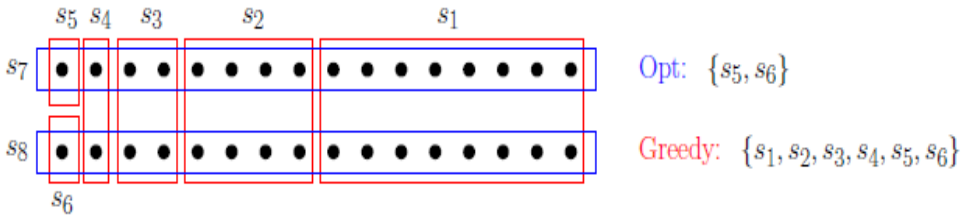
38

Set Cover (contd.) Example

- For the example given earlier the greedy-set cover algorithm would select s1 (see Fig. (a)), then s_6 (see Fig. (b)), then s_2 (see Fig. (c)) and finally s_3 (see Fig. (d)). Thus, it would return a set cover of size four, whereas the optimal set cover has size three.



Set Cover (contd.) Approximation Factor



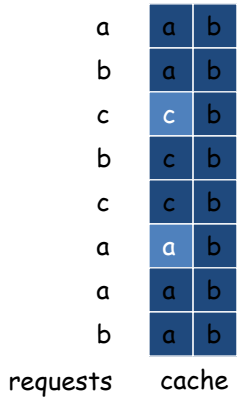
- Problem with the greedy heuristic is that it can be "fooled" into picking the wrong set, repeatedly. Consider the example shown here involving a universe of 32 elements. The optimal set cover consists of sets s_7 and s_8, each of size 16. Initially all three sets s_1, s_7, and s_8 have 16 elements. If ties are broken in the worst possible way, the greedy algorithm will first select the set s_1. We remove all the covered elements. Now s_2, s_7 and s_8 all cover eight of the remaining elements. Again, if we choose poorly, s_2 is chosen. The pattern repeats, choosing s_3 (covering four of the remainder), s_4 (covering two) and finally s_5 and s_6 (each covering one). Although there are ties for the greedy choice in this example, it is easy to modify the example so that the greedy choice is unique. Try to do that as an exercise.

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**.

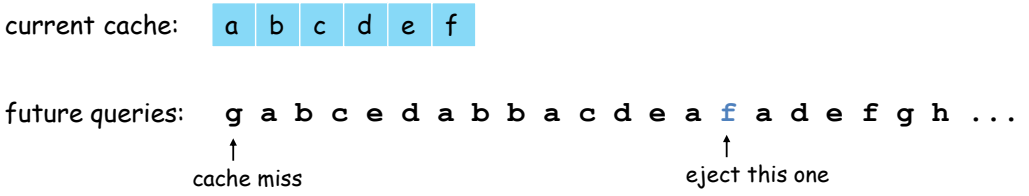
- Goal. Eviction schedule that minimizes number of cache misses.

- 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

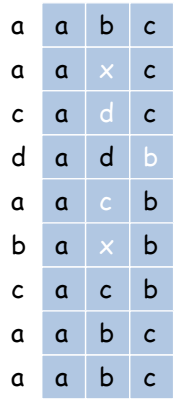
- Farthest-in-future. Evict item in the cache that is not requested until farthest in the future.



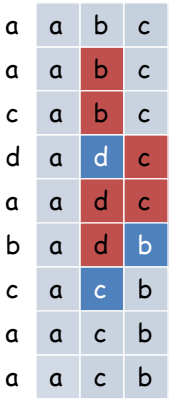
- Theorem. [Bellady, 1960s] FF is optimal eviction schedule.
- Pf. Algorithm and theorem are intuitive; proof is subtle.

Reduced Eviction Schedules

- Def. A reduced schedule is a schedule that only inserts an item into the cache in a step in which that item is requested.
- Intuition. Can transform an unreduced schedule into a reduced one with no more cache misses.



an unreduced schedule

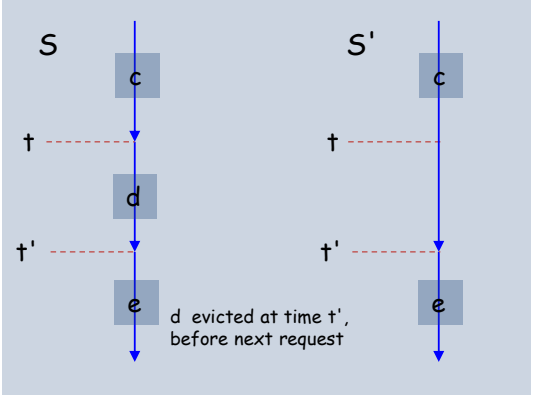


a reduced schedule

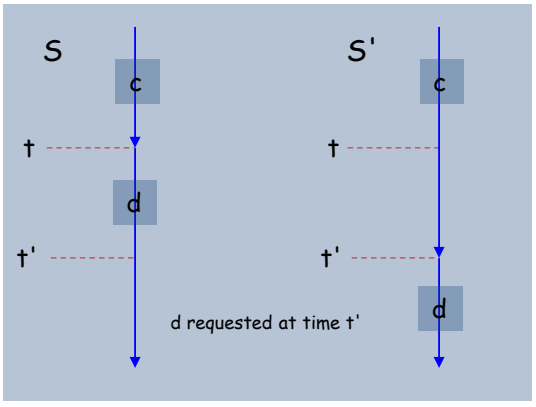
Reduced Eviction Schedules

- Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more cache misses.
- Pf. (by induction on number of unreduced items)
  - 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. ▪

doesn't enter cache at requested time



Case 1



Case 2

Farthest-In-Future: Analysis

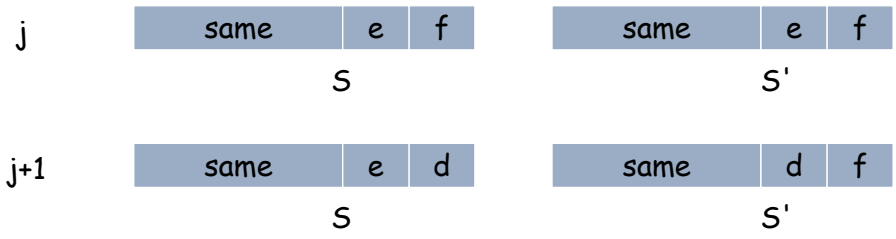
- Theorem. FF is optimal eviction algorithm.
- Pf. (by induction on number or requests j)

> 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 ≠ e).
    - begin construction of S' from S by evicting e instead of f

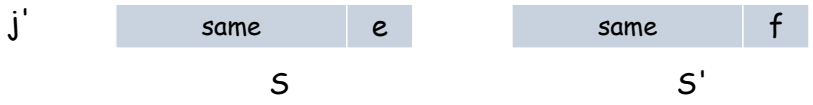


    - now S' agrees with S_{FF} on first j+1 requests; we show that having element f in cache is no worse than having element e

Farthest-In-Future: Analysis

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

must involve e or f (or both)



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

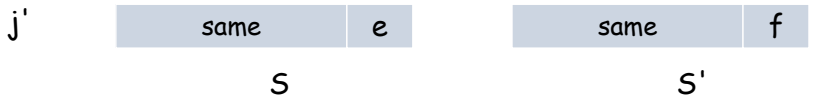
  - Case 3b: g = f. Element f can't be in cache of S, so let e' be the element that S evicts.
    - if e' = e, S' accesses f from cache; now S and S' have same cache
    - if e' ≠ e, S' evicts e' and brings e into the cache; now S and S' have the same cache

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

Farthest-In-Future: Analysis

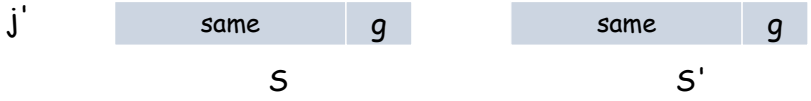
- Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.

must involve e or f (or both)



otherwise, S' would take the same action

- Case 3c: g ≠ 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.
  - LIFO is arbitrarily bad.

49