Acknowledgement: The course slides are adapted from the slides prepared by K. Wayne of Princeton University.
**Greedy Choice**

- Optimal substructure property exploited by both *Greedy* and DP strategies

- **Greedy Choice Property**: A sequence of locally optimal choices $\Rightarrow$ an optimal solution
  - We make the choice that seems best at the moment
  - Then solve the subproblem arising after the choice is made

**DP**: We also make a choice/decision at each step, but the choice may depend on the optimal solutions to subproblems

**Greedy**: The choice may depend on the choices made so far, but it cannot depend on any future choices or on the solutions to subproblems
**Greedy vs Dynamic Programming**

- DP is a bottom-up strategy
- **Greedy** is a top-down strategy
  - each greedy choice in the sequence iteratively reduces each problem to a similar but smaller problem
Proof of Correctness

- Examine a globally optimal solution

- Apply induction to show that a greedy choice can be used at every step

- Show that this solution can be modified so that
  1. A greedy choice is made as the first step
  2. This choice reduces the problem to a similar but smaller problem

Showing (2) reduces the proof of correctness to proving that the problem exhibits optimal substructure property
Proof of Correctness

Greedy Choice Property: A globally optimal solution can be arrived at by making locally optimal (greedy) choices

- In **DP**, we make a choice at each step but the choice may depend on the solutions to subproblems
- In **Greedy Algorithms**, we make the choice that seems best at that moment then solve the subproblems arising after the choice is made
  - The choice may depend on choices so far, but it cannot depend on any future choice or on the solutions to subproblems

- DP solves the problem bottom-up
- Greedy usually progresses in a top-down fashion by making one greedy choice after another reducing each given problem instance to a smaller one
Proof of Correctness

• We must prove that a greedy choice at each step yields a globally optimal solution

• The proof examines a globally optimal solution

• Shows that the solution can be modified so that a greedy choice made as the first step reduces the problem to a similar but smaller subproblem

• Then induction is applied to show that a greedy choice can be used at each step

• Hence, this induction proof reduces the proof of correctness to demonstrating that an optimal solution must exhibit optimal substructure property
**Optimal Substructure**

- Optimal substructure property is exploited by both *Greedy* and *dynamic programming* strategies.

- Hence one may
  - Try to generate a dynamic programming solution to a problem when a greedy strategy suffices
  - Or, may mistakenly think that a greedy soln works when in fact a DP solution is required

**Example**: Knapsack Problems(S, w)
Knapsack Problems

The 0-1 Knapsack Problem \((S,W)\)
- A thief robbing a store finds \(n\) items \(S = \{i_1, i_2, \ldots, i_n\}\), the \(i\)th item is worth \(v_i\) dollars and weighs \(w_i\) pounds, where \(v_i\) and \(w_i\) are integers
- He wants to take as valuable a load as possible, but he can carry at most \(W\) pounds in his knapsack, where \(W\) is an integer
- The thief cannot take a fractional amount of an item

The Fractional Knapsack Problem \((S, W)\)
- The scenario is the same
- But, the thief can take fractions of items rather than having to make binary (0-1) choice for each item
Knapsack Problems

Although the problems are similar

- the Fractional Knapsack Problem is solvable by Greedy strategy
- whereas, the 0-1 Knapsack Problem is not

<table>
<thead>
<tr>
<th>i</th>
<th>$v_i$</th>
<th>$w_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$1$</td>
<td>$1$ kg</td>
</tr>
<tr>
<td>2</td>
<td>$6$</td>
<td>$2$ kg</td>
</tr>
<tr>
<td>3</td>
<td>$18$</td>
<td>$5$ kg</td>
</tr>
<tr>
<td>4</td>
<td>$22$</td>
<td>$6$ kg</td>
</tr>
<tr>
<td>5</td>
<td>$28$</td>
<td>$7$ kg</td>
</tr>
</tbody>
</table>

knapsack instance (weight limit $W = 11$)

weights and values can be arbitrary positive integers
Greedy Solution to Fractional Knapsack

1. Compute the value per pound $v_i/w_i$ for each item
2. The thief begins by taking, as much as possible, of the item with the greatest value per pound
3. If the supply of that item is exhausted before filling the knapsack he takes, as much as possible, of the item with the next greatest value per pound
4. Repeat (2-3) until his knapsack becomes full

Thus, by sorting the items by value per pound the greedy algorithm runs in $O(n \log n)$ time
Greedy Algorithms

- coin changing
- interval scheduling
- interval partitioning
- scheduling to minimize lateness
- optimal caching
Coin changing

**Goal.** Given U. S. currency denominations \{1, 5, 10, 25, 100\}, devise a method to pay amount to customer using fewest 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.

\[
\text{CASHIERS-ALGORITHM} \ (x, c_1, c_2, \ldots, c_n) \\
\text{SORT} \ n \text{ coin denominations so that } 0 < c_1 < c_2 < \ldots < c_n. \\
S \leftarrow \emptyset. \ \text{multiset of coins selected} \\
\text{WHILE} \ (x > 0) \\
\quad k \leftarrow \text{largest coin denomination } c_k \text{ such that } c_k \leq x. \\
\quad \text{IF} \ (\text{no such } k) \\
\quad \quad \text{RETURN} \ \text{“no solution.”} \\
\quad \text{ELSE} \\
\quad \quad x \leftarrow x - c_k. \\
\quad \quad S \leftarrow S \cup \{ k \}. \\
\quad \text{RETURN } S.
\]
Cashier’s algorithm (for arbitrary coin denominations)

Q. Is cashier’s algorithm optimal 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.
Properties of any optimal solution (for U.S. coin denominations)

Property. Number of pennies ≤ 4.
Pf. Replace 5 pennies with 1 nickel.

Property. Number of nickels ≤ 1.

Property. Number of quarters ≤ 3.

Property. Number of nickels + number of dimes ≤ 2.
Pf.

- Recall: ≤ 1 nickel.
- Replace 3 dimes and 0 nickels with 1 quarter and 1 nickel;
- Replace 2 dimes and 1 nickel with 1 quarter.
Optimality of cashier’s algorithm (for U.S. coin denominations)

Theorem. Cashier’s algorithm is optimal for U.S. coins \( \{ 1, 5, 10, 25, 100 \} \).

Pf. [ by induction on amount to be paid \( x \) ]

- Consider optimal way to change \( c_k \leq x < c_{k+1} \) : greedy takes coin \( k \).
- We claim that any optimal solution must 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. □

<table>
<thead>
<tr>
<th>( k )</th>
<th>( c_k )</th>
<th>all optimal solutions must satisfy</th>
<th>max value of coin denominations ( c_1, c_2, \ldots, c_{k-1} ) in any optimal solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>( P \leq 4 )</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>( N \leq 1 )</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>( N + D \leq 2 )</td>
<td>( 4 + 5 = 9 )</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>( Q \leq 3 )</td>
<td>( 20 + 4 = 24 )</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>no limit</td>
<td>( 75 + 24 = 99 )</td>
</tr>
</tbody>
</table>
Greedy Algorithms

- coin changing
- **interval scheduling**
- interval partitioning
- scheduling to minimize lateness
- optimal caching
Interval scheduling

- Job $j$ starts at $s_j$ and finishes at $f_j$.
- Two jobs are compatible if they don’t overlap.
- Goal: find maximum subset of mutually compatible jobs.
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**
  ![Diagram for earliest start time]

- **counterexample for shortest interval**
  ![Diagram for shortest interval]

- **counterexample for fewest conflicts**
  ![Diagram for fewest conflicts]
### Interval scheduling: earliest-finish-time-first algorithm

**Proposition.** Can implement earliest-finish-time-first in $O(n \log n)$ time.
- Keep track of job $j^*$ that was added last to $S$.
- Job $j$ is compatible with $S$ iff $s_j \geq f_{j^*}$.
- Sorting by finish times takes $O(n \log n)$ time.
Earliest-finish-time-first algorithm demo

The diagram illustrates the Earliest-finish-time-first algorithm with tasks A through H. Each task is represented by a horizontal bar, indicating the time it takes to complete. The bars are ordered such that tasks finish earliest first, as per the algorithm's principle.
Earliest-finish-time-first algorithm demo

job B is compatible (add to schedule)
Earliest-finish-time-first algorithm demo

The diagram illustrates the Earliest-finish-time-first algorithm with tasks labeled A, B, C, D, E, F, G, and H. The tasks are arranged on a timeline from left to right, indicating their earliest finish time. Each task is represented by a horizontal bar on the timeline, with the time points marked at the bottom.
Earliest-finish-time-first algorithm demo

job C is incompatible (do not add to schedule)
Earliest-finish-time-first algorithm demo
Earliest-finish-time-first algorithm demo

job A is incompatible (do not add to schedule)
Earliest-finish-time-first algorithm demo
Earliest-finish-time-first algorithm demo

job E is compatible (add to schedule)
Earliest-finish-time-first algorithm demo
Earliest-finish-time-first algorithm demo

job D is incompatible (do not add to schedule)
Earliest-finish-time-first algorithm demo
Earliest-finish-time-first algorithm demo

Job F is incompatible (do not add to schedule)
Earliest-finish-time-first algorithm demo
Earliest-finish-time-first algorithm demo

job G is incompatible (do not add to schedule)
job G is incompatible (do not add to schedule)
Earliest-finish-time-first algorithm demo
Earliest-finish-time-first algorithm demo

Job H is compatible (add to schedule)
Earliest-finish-time-first algorithm demo
Earliest-finish-time-first algorithm demo

done (an optimal set of jobs)
Interval scheduling: analysis of earliest-finish-time-first algorithm

Theorem. The earliest-finish-time-first 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 an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of $r$.

why not replace job $j_{r+1}$ with job $i_{r+1}$?

Greedy: $i_1 \rightarrow i_2 \rightarrow i_r \rightarrow i_{r+1} \rightarrow \cdots \rightarrow i_k$

Optimal: $j_1 \rightarrow j_2 \rightarrow j_r \rightarrow j_{r+1} \rightarrow \cdots \rightarrow j_m$

job $i_{r+1}$ exists and finishes no later than $j_{r+1}$
**Interval scheduling: analysis of earliest-finish-time-first algorithm**

**Theorem.** The earliest-finish-time-first 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 an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of $r$.

---

Solution still feasible and optimal (but contradicts maximality of $r$)

---

Greedy:

| $i_1$ | $i_2$ | $i_r$ | $i_{r+1}$ | $\ldots$ | $i_k$ |

Optimal:

| $j_1$ | $j_2$ | $j_r$ | $i_{r+1}$ | $\ldots$ | $j_m$ |
Greedy Algorithms

- coin changing
- interval scheduling
- interval partitioning
- scheduling to minimize lateness
- optimal caching
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 lectures occur at the same time in the same room.

Ex. This schedule uses 4 classrooms to schedule 10 lectures.
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 lectures occur at the same time in the same room.

Ex. This schedule uses 3 classrooms to schedule 10 lectures.
Greedy template. Consider lectures in some natural order. Assign each lecture to an available classroom (which one?); allocate a new classroom if none are available.

- [Earliest start time] Consider lectures in ascending order of $s_j$.
- [Earliest finish time] Consider lectures in ascending order of $f_j$.
- [Shortest interval] Consider lectures in ascending order of $f_j - s_j$.
- [Fewest conflicts] For each lecture $j$, count the number of conflicting lectures $c_j$. Schedule in ascending order of $c_j$. 
**Interval partitioning: greedy algorithms**

**Greedy template.** Consider lectures in some natural order. Assign each lecture to an available classroom (which one?); allocate a new classroom if none are available.

<table>
<thead>
<tr>
<th>counterexample for earliest finish time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>counterexample for shortest interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>counterexample for fewest conflicts</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
**Early Start Time First** \((n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)\)

**Sort** lectures by start times and renumber so that \(s_1 \leq s_2 \leq \ldots \leq s_n\).

\[ d \leftarrow 0. \quad \text{number of allocated classrooms} \]

**FOR** \(j = 1\) **TO** \(n\)

**IF** (lecture \(j\) is compatible with some classroom)

Schedule lecture \(j\) in any such classroom \(k\).

**ELSE**

Allocate a new classroom \(d + 1\).

Schedule lecture \(j\) in classroom \(d + 1\).

\[ d \leftarrow d + 1. \]

**RETURN** schedule.
Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if one exists).
- Otherwise, open up a new classroom.
Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if one exists).
- Otherwise, open up a new classroom.

no compatible classroom: open up a new classroom and assign lecture to it
Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if one exists).
- Otherwise, open up a new classroom.

*no compatible classroom: open up a new classroom and assign lecture to it*
Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if one exists).
- Otherwise, open up a new classroom.

no compatible classroom: open up a new classroom and assign lecture to it
Earliest-start-time-first algorithm demo

Consider lectures in order of start time:
- Assign next lecture to any compatible classroom (if one exists).
- Otherwise, open up a new classroom.

lecture d is compatible with classrooms 1 and 3
Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if one exists).
- Otherwise, open up a new classroom.

lecture e is compatible with classroom 1
Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

• Assign next lecture to any compatible classroom (if one exists).
• Otherwise, open up a new classroom.

lecture f is compatible with classroom 2 and 3
Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if one exists).
- Otherwise, open up a new classroom.

lecture g is compatible with classroom 2
Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if one exists).
- Otherwise, open up a new classroom.

lecture h is compatible with classroom 1

<table>
<thead>
<tr>
<th>Time</th>
<th>9</th>
<th>9:30</th>
<th>10</th>
<th>10:30</th>
<th>11</th>
<th>11:30</th>
<th>12</th>
<th>12:30</th>
<th>1</th>
<th>1:30</th>
<th>2</th>
<th>2:30</th>
<th>3</th>
<th>3:30</th>
<th>4</th>
<th>4:30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if one exists).
- Otherwise, open up a new classroom.

lecture j is compatible with classrooms 2 and 3
Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if one exists).
- Otherwise, open up a new classroom.

lecture i is compatible with classroom 2
Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if one exists).
- Otherwise, open up a new classroom.

done
**Interval partitioning: earliest-start-time-first algorithm**

**Proposition.** The earliest-start-time-first algorithm can be implemented in $O(n \log n)$ time.

**Pf.**

- Sorting by start times takes $O(n \log n)$ time.
- Store classrooms in a **priority queue** (key = finish time of its last lecture).
  - to allocate a new classroom, **INSERT** classroom onto priority queue.
  - to schedule lecture $j$ in classroom $k$, **INCREASE-KEY** of classroom $k$ to $f_j$.
  - to determine whether lecture $j$ is compatible with any classroom, compare $s_j$ to **FIND-MIN**
- Total # of priority queue operations is $O(n)$; each takes $O(\log n)$ time.

**Remark.** This implementation chooses a classroom $k$ whose finish time of its last lecture is the **earliest**.
Interval partitioning: lower bound on optimal solution

**Def.** The depth of a set of open intervals is the maximum number of intervals that contain any given point.

**Key observation.** Number of classrooms needed \( \geq \) depth.

**Q.** Does minimum number of classrooms needed always equal depth?  
**A.** Yes! Moreover, earliest-start-time-first algorithm finds a schedule whose number of classrooms equals the depth.
**Observation.** The earliest-start-time first algorithm never schedules two incompatible lectures in the same classroom.

**Theorem.** Earliest-start-time-first algorithm is optimal.

**Pf.**

- Let $d =$ number of classrooms that the algorithm allocates.
- Classroom $d$ is opened because we needed to schedule a lecture, say $j$, that is incompatible with a lecture in each of $d - 1$ other classrooms.
- Thus, these $d$ lectures each end after $s_j$.
- Since we sorted by start time, each of these incompatible lectures 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.  ■
Greedy Algorithms

- coin changing
- interval scheduling
- interval partitioning
- scheduling to minimize lateness
- optimal caching
Scheduling to minimizing lateness

- 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 maximum lateness $L = \max_j \ell_j$.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$d_j$</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

$d_3 = 9$  $d_2 = 8$  $d_6 = 15$  $d_1 = 6$  $d_5 = 14$  $d_4 = 9$

lateness = 2  lateness = 0  max lateness = 6
Minimizing lateness: greedy algorithms

Greedy template. Schedule jobs according to some natural order.

- [Shortest processing time first] Schedule jobs in ascending order of processing time $t_j$.

- [Earliest deadline first] Schedule jobs in ascending order of deadline $d_j$.

- [Smallest slack] Schedule jobs in ascending order of slack $d_j - t_j$. 
Minimizing lateness: greedy algorithms

Greedy template. Schedule jobs according to some natural order.

- [Shortest processing time first] Schedule jobs in ascending order of processing time $t_j$.

\[
\begin{array}{c|c|c}
1 & 2 \\
\hline
\text{t}_j & 1 & 10 \\
\text{d}_j & 100 & 10 \\
\end{array}
\]

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

\[
\begin{array}{c|c|c}
1 & 2 \\
\hline
\text{t}_j & 1 & 10 \\
\text{d}_j & 2 & 10 \\
\end{array}
\]
### Minimizing lateness: earliest deadline first

**EARLIEST-DEADLINE-FIRST** \((n, t_1, t_2, \ldots, t_n, d_1, d_2, \ldots, d_n)\)

- **SORT** jobs by due times and renumber so that \(d_1 \leq d_2 \leq \ldots \leq d_n\).
- \(t \leftarrow 0\).
- **FOR** \(j = 1\) **TO** \(n\)
  - Assign job \(j\) to interval \([t, t + t_j]\).
  - \(s_j \leftarrow t\); \(f_j \leftarrow t + t_j\).
  - \(t \leftarrow t + t_j\).
- **RETURN** intervals \([s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]\).

---

**max lateness** \(L = 1\)

<table>
<thead>
<tr>
<th>(d_1 = 6)</th>
<th>(d_2 = 8)</th>
<th>(d_3 = 9)</th>
<th>(d_4 = 9)</th>
<th>(d_5 = 14)</th>
<th>(d_6 = 15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Minimizing lateness: no idle time

Observation 1. There exists an optimal schedule with no idle time.

Observation 2. The earliest-deadline-first 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 $j$ is scheduled before $i$.

<table>
<thead>
<tr>
<th>a schedule with an inversion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>$j$</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$i$</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Inversion if $i < j$

Recall: we assume the jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$

**Observation 3.** The earliest-deadline-first schedule is the unique idle-free schedule with no inversions.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
<th>$n$</th>
</tr>
</thead>
</table>

**Minimizing lateness: inversions**

**Def.** Given a schedule $S$, an inversion is a pair of jobs $i$ and $j$ such that:

$i < j$ but $j$ is scheduled before $i$.

![a schedule with an inversion]

- **Observation 4.** If an idle-free schedule has an inversion, then it has an adjacent inversion.

**Pf.**

- Let $i - j$ be a closest inversion.
- Let $k$ be element immediately to the right of $j$.
- **Case 1.** $[j > k]$ Then $j - k$ is an adjacent inversion.
- **Case 2.** $[j < k]$ Then $i - k$ is a closer inversion since $i < j < k$. ※

![two inverted jobs scheduled consecutively]
Minimizing lateness: inversions

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

**Key claim.** Exchanging two adjacent, inverted jobs $i$ and $j$ reduces the number of inversions by 1 and does not increase the max lateness.

**Pf.** Let $\ell$ be the lateness before the swap, and let $\ell'$ be it afterwards.

- $\ell'_k = \ell_k$ for all $k \neq i, j$.
- $\ell'_i \leq \ell_i$.
- If job $j$ is late, $\ell'_j = f'_j - d_j$ (definition)
  
  $= f_i - d_j$ (j now finishes at time $f_i$)
  
  $\leq f_i - d_i$ (since $i < j \implies d_i \leq d_j$)
  
  $\leq \ell_i$ (definition)
Minimizing lateness: analysis of earliest-deadline-first algorithm

**Theorem.** The earliest-deadline-first schedule $S$ is optimal.

**Pf.** [by contradiction]

Define $S^\ast$ to be an optimal schedule with the fewest inversions.

- Can assume $S^\ast$ has no idle time. \[\text{Observation 1}\]

- Case 1. [ $S^\ast$ has no inversions ] Then $S = S^\ast$. \[\text{Observation 3}\]

- Case 2. [ $S^\ast$ has an inversion ]
  - let $i-j$ be an adjacent inversion \[\text{Observation 4}\]
  - exchanging jobs $i$ and $j$ decreases the number of inversions by 1 without increasing the max lateness \[\text{key claim}\]
  - contradicts “fewest inversions” part of the definition of $S^\ast$  \*
Google’s foo.bar challenge

A “secret” web tool that Google uses to recruit developers.

• Triggered by specific searches related to programming.
• Algorithmic coding challenges of increasing difficulty.
Quantum antimatter fuel comes in small pellets, which is convenient since the many moving parts of the LAMBCHOP each need to be fed fuel one pellet at a time. However, minions dump pellets in bulk into the fuel intake. You need to figure out the most efficient way to sort and shift the pellets down to a single pellet at a time.

The fuel control mechanisms have three operations:

- Add 1 fuel pellet
- Remove 1 fuel pellet
- Divide the entire group of fuel pellets by 2 (due to the destructive energy released when a quantum antimatter pellet is cut in half, the safety controls will only allow this to happen if there is an even number of pellets)

Write a function called answer(n) which takes a positive integer n as a string and returns the minimum number of operations needed to transform the number of pellets to 1.

29 → 28 → 14 → 7 → 8 → 4 → 2 → 1
Google's foo.bar challenge

Level 3 complete. You are now on level 4. Challenges to complete level: 2.

Level 1 100% [==================================]
Level 2 100% [==================================]
Level 3 100% [==================================]
Level 4  0% [.................................]
Level 5  0% [.................................]

Excellent! You've destroyed Commander Lambda's doomsday device and saved Bunny Planet! But there's one small problem: the LAMBCHOP was a wool-y important part of her space station, and when you blew it up, you triggered a chain reaction that's tearing the station apart. Can you rescue the imprisoned bunnies and escape before the entire thing explodes?

Type request to request a new challenge now, or come back later.

[#1] The code is strong with this one. Share solutions with a Google recruiter?

[Y]es [N]o [A]sk me later: A

Response: contact postponed.
To share your progress at any time, use the recruitme command.
Greedy Algorithms

- coin changing
- interval scheduling
- interval partitioning
- scheduling to minimize lateness
- 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 in cache when requested.
- Cache miss: item not in cache when requested.
  
  (must evict some item from cache and bring requested item into cache)

Applications. CPU, RAM, hard drive, web, browser, ….

Goal. Eviction schedule that minimizes the number of evictions.

Ex. $k = 2$, initial cache = $ab$, requests: $a, b, c, b, c, a, b$.
Optimal eviction schedule. 2 evictions.
Optimal offline caching: greedy algorithms

**LIFO/FIFO.** Evict item brought in least (most) recently.

**LRU.** Evict item whose most recent access was earliest.

**LFU.** Evict item that was least frequently requested.

---

LIFO/FIFO: eject a
LRU: eject d
LIFO: eject e

cache miss
(which item to eject?)
Optimal offline caching: farthest-in-future (clairvoyant algorithm)

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

Theorem. [Bélaïdy 1966] 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 brings an item \( d \) into the cache in step \( j \) only if there is a request for \( d \) in step \( j \) and \( d \) is not already in the cache.

---

*an unreduced schedule*

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>c</td>
<td>b</td>
</tr>
</tbody>
</table>

*an unreduced schedule*

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>d</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>c</td>
<td>b</td>
</tr>
</tbody>
</table>

\( d \) enters cache without a request

\( d \) enters cache even though already in cache

*an unreduced schedule*

* a reduced schedule
Claim. Given any unreduced schedule $S$, can transform it into a reduced schedule $S'$ with no more evictions.

Pf. [by induction on number of steps $j$]

- Suppose $S$ brings $d$ into the cache in step $j$ without a request.
- Let $c$ be the item $S$ evicts when it brings $d$ into the cache.
- Case 1a: $d$ evicted before next request for $d$.  

\[d\] enters cache without a request.  
\[d\] evicted before next request for $d$.  
might as well leave $c$ in cache until $d$ is evicted.
**Reduced eviction schedules**

**Claim.** Given any unreduced schedule $S$, can transform it into a reduced schedule $S'$ with no more evictions.

**Pf.** [by induction on number of steps $j$]

- Suppose $S$ brings $d$ into the cache in step $j$ without a request.
- Let $c$ be the item $S$ evicts when it brings $d$ into the cache.
- Case 1a: $d$ evicted before next request for $d$.
- Case 1b: next request for $d$ occurs before $d$ is evicted.
Reduced eviction schedules

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

Pf. [by induction on number of steps $j$]

- Suppose $S$ brings $d$ into the cache in step $j$ even though $d$ is in cache.
- Let $c$ be the item $S$ evicts when it brings $d$ into the cache.
- Case 2a: $d$ evicted before it is needed.
Reduced eviction schedules

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

Pf. [by induction on number of steps $j$ ]

- Suppose $S$ brings $d$ into the cache in step $j$ even though $d$ is in cache.
- Let $c$ be the item $S$ evicts when it brings $d$ into the cache.
- Case 2a: $d$ evicted before it is needed.
- Case 2b: $d$ needed before it is evicted.

<table>
<thead>
<tr>
<th>unreduced schedule $S$</th>
<th>$S'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_1$ $a$ $c$</td>
<td>$d_1$ $a$ $c$</td>
</tr>
<tr>
<td>$d_1$ $a$ $c$</td>
<td>$d_1$ $a$ $c$</td>
</tr>
<tr>
<td>$d_1$ $a$ $c$</td>
<td>$d_1$ $a$ $c$</td>
</tr>
<tr>
<td>$d_1$ $a$ $d_3$</td>
<td>$d_1$ $a$ $c$</td>
</tr>
<tr>
<td>$d_1$ $a$ $d_3$</td>
<td>$d_1$ $a$ $c$</td>
</tr>
<tr>
<td>$c$ $c$ $a$ $d_3$</td>
<td>$d$ $d_1$ $a$ $c$</td>
</tr>
<tr>
<td>$c$ $c$ $a$ $d_3$</td>
<td>$d$ $d_1$ $a$ $c$</td>
</tr>
<tr>
<td>$a$ $c$ $a$ $d_3$</td>
<td>$c$ $c$ $a$ $c$</td>
</tr>
<tr>
<td>$a$ $c$ $a$ $d_3$</td>
<td>$c$ $c$ $a$ $c$</td>
</tr>
<tr>
<td>step $j$</td>
<td>step $j'$</td>
</tr>
</tbody>
</table>

$d_3$ enters cache even though $d_1$ is already in cache

$d_3$ not needed

$d_3$ needed

might as well leave $c$ in cache until $d_3$ in needed
**Reduced eviction schedules**

**Claim.** Given any unreduced schedule $S$, can transform it into a reduced schedule $S'$ with no more evictions.

**Pf.** [by induction on number of steps $j$]

- Case 1: $S$ brings $d$ into the cache in step $j$ without a request. ✔
- Case 2: $S$ brings $d$ into the cache in step $j$ even though $d$ is in cache. ✔
- If multiple unreduced items in step $j$, apply each one in turn, dealing with Case 1 before Case 2.

 vrij: resolving Case 1 might trigger Case 2
Farthest-in-future: analysis

**Theorem.** FF is optimal eviction algorithm.

**Pf.** Follows directly from the following invariant.

**Invariant.** There exists an optimal reduced schedule $S$ that has the same eviction schedule as $S_{FF}$ through the first $j$ steps.

**Pf.** [by induction on number of steps $j$]

Base case: $j = 0$.

Let $S$ be reduced schedule that satisfies invariant through $j$ steps.

We produce $S'$ that satisfies invariant after $j + 1$ steps.

- Let $d$ denote the item requested in step $j + 1$.
- Since $S$ and $S_{FF}$ have agreed up until now, they have the same cache contents before step $j + 1$.
- 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 item.
  $S' = S$ 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 $f$

$$
\begin{array}{c|cc|c}
\text{same} & e & f & \text{step j} \\
\hline
S & \text{same} & e & f \\
S' & \text{same} & d & f
\end{array}
$$

- now $S'$ agrees with $S_{FF}$ for first $j+1$ steps; we show that having item $f$ in cache is no worse than having item $e$ in cache

- let $S'$ behave the same as $S$ until $S'$ is forced to take a different action
  (because either $S$ evicts $e$; or because either $e$ or $f$ is requested)
Let \( j' \) be the \textbf{first} step after \( j + 1 \) that \( S' \) must take a different action from \( S \); let \( g \) denote the item requested in step \( j' \).

\[
\begin{array}{|c|c|}
\hline
\text{same} & e \\
\hline
\end{array}
\quad \begin{array}{|c|c|}
\hline
\text{step } j' & \text{same} & f \\
\hline
\end{array}
\]

\( S \quad S' \)

\begin{itemize}
  \item **Case 3a:** \( g = e \).
    Can’t happen with FF since there must be a request for \( f \) before \( e \).

  \item **Case 3b:** \( g = f \).
    Element \( f \) can’t be in cache of \( S \); let \( e' \) be the item that \( S \) evicts.
    \begin{itemize}
      \item if \( e' = e \), \( S' \) accesses \( f \) from cache; now \( S \) and \( S' \) have same cache
      \item if \( e' \neq e \), we make \( S' \) evict \( e' \) and bring \( e \) into the cache;
          now \( S \) and \( S' \) have the same cache
    \end{itemize}
    We let \( S' \) behave exactly like \( S \) for remaining requests.
\end{itemize}

\( S' \) is no longer reduced, but can be transformed into a reduced schedule that agrees with FF through first \( j + 1 \) steps.
Farthest-in-future: analysis

Let $j'$ be the first step after $j + 1$ that $S'$ must take a different action from $S$; let $g$ denote the item requested in step $j'$.

![Diagram showing the analysis of Farthest-in-future]

- Case 3c: $g \neq e, f$. $S$ evicts $e$.
  - make $S'$ evict $f$.
  - now $S$ and $S'$ have the same cache
  - let $S'$ behave exactly like $S$ for the remaining requests
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 item brought in most recently.

LRU. Evict item whose most recent access was earliest.

FF is optimal offline eviction algorithm.

- Provides basis for understanding and analyzing online algorithms.
- LIFO can be arbitrarily bad.
- LRU is $k$-competitive: for any sequence of requests $\sigma$, $LRU(\sigma) \leq k \times FF(\sigma) + k$.
- Randomized marking is $O(\log k)$-competitive.

see Section 13.8
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. Gale–Shapley, Kruskal, Prim, Dijkstra, Huffman, …