4. **Greedy Algorithms I**

- 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.

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

1. **SORT** \(n\) coin denominations so that \(0 < c_1 < c_2 < \ldots < c_n\).
2. **SET** \(S \leftarrow \emptyset\). **
3. **WHILE** \(x > 0\)
   - **SET** \(k \leftarrow\) largest coin denomination \(c_k\) such that \(c_k \leq x\).
   - **IF** (no such \(k\))
     - **RETURN** “no solution.”
   - **ELSE**
     - **SET** \(x \leftarrow x - c_k\).
     - **SET** \(S \leftarrow S \cup \{k\}\).
4. **RETURN** \(S\).
Is the cashier’s algorithm optimal?

A. Yes, greedy algorithms are always optimal.
B. Yes, for any set of coin denominations $c_1 < c_2 < \ldots < c_k$ provided $c_1 = 1$.
C. Yes, because of special properties of U.S. coin denominations.
D. No.

---

Properties of any optimal solution (for U.S. coin denominations)

**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.**
- Recall: $\leq 1$ 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>
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.

**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.
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 $r$.

Greedy:

```
1 2 3 4 ...
```

Optimal:

```
1 2 3 4 ...
```

---

Suppose that each job also has a positive weight and the goal is to find a maximum weight subset of mutually compatible intervals.

Is the earliest–finish–time–first algorithm still optimal?

- **A.** Yes, because greedy algorithms are always optimal.
- **B.** Yes, because the same proof of correctness is valid.
- **C.** No, because the same proof of correctness is no longer valid.
- **D.** No, because you could assign a huge weight to a job that overlaps the job with the earliest finish time.
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.

Consider lectures in some order, assigning each lecture to first available classroom (opening a new classroom if none is available). Which rule is optimal?

A. **[Earliest start time]** Consider lectures in ascending order of \( s_j \).

B. **[Earliest finish time]** Consider lectures in ascending order of \( f_j \).

C. **[Shortest interval]** Consider lectures in ascending order of \( f_j - s_j \).

D. None of the above.

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

\[
\text{earliest-start-time-first} \ (n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)
\]

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

\( d \leftarrow 0 \). **\text{number of allocated classrooms}**

**FOR** \( j = 1 \text{ TO } n \)

**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.
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.

Interval partitioning: analysis of earliest-start-time-first algorithm

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

![Scheduling Table and Graph](image)

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 \( d_1 \leq d_2 \leq \cdots \leq d_n \).

**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 \([s_1, f_1],[s_2, f_2],\ldots,[s_n, f_n]\).

![Earliest Deadline First Algorithm](image)

Minimizing lateness: no idle time

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

![Optimal Schedule](image)

**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 $i$.

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

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 $t$ be the lateness before the swap, and let $t'$ be it afterwards.

- $t'_k = t_k$ for all $k \neq i, j$.
- $t'_j \leq t'_i$.
- If job $j$ is late, $t'_j = f'_j - d_j = f_i - d_j \leq f_i - d_i \leq t_i$.  
  \[ i < j \implies d_i \leq d_j \]

Minimizing lateness: analysis of earliest-deadline-first algorithm

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

Pf. [by contradiction]

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

- Can assume $S^*$ has no idle time.  
  \[ Y \]
- Case 1. $[S^* \text{ has no inversions }] \implies S = S^*$.  
  \[ W \]
- Case 2. $[S^* \text{ has an inversion }]$
  - let $i\sim j$ be an adjacent inversion
  - 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 $S^*$.  

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 $i$.

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

Pf.

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

---

Quantum antimatter fuel comes in small pellets, which is convenient since the many moving parts of the LAMBCROP 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
```
4. Greedy Algorithms I

- 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.

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. [Bélaðy 1966] FF is optimal eviction schedule.
Pf. Algorithm and theorem are intuitive; proof is subtle.
Greedy algorithms I: quiz 6

Which item will be evicted next using farthest-in-future schedule?

A.

B.

C.

D.

E.

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.

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 \( d \).

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.
- Case 2b: $d$ needed before it is evicted.

“Eproduce $d$, enters cache even though $d$ is already in cache
d, not needed

unreduced schedule $S$

\[
\begin{array}{|c|c|}
\hline
\text{step $j$} & \text{step $j'$} \\
\hline
d & d & d_1 & d_3 \\
\hline
\text{not needed} & \text{evicted} & d_1, d_3, \text{needed} \\
\hline
\end{array}
\]

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 2.

\[
\text{resolving Case 1 might trigger Case 2}
\]

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 $j+1$.
- If contents of $S$ and $S_{FF}$ have agreed up until now, then 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 \( f \)

<table>
<thead>
<tr>
<th>same</th>
<th>( e )</th>
<th>( f )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>same</td>
<td>( e )</td>
<td>( f )</td>
<td>( S' )</td>
</tr>
<tr>
<td>same</td>
<td>( e )</td>
<td>( d )</td>
<td>( \text{step } j+1 )</td>
</tr>
<tr>
<td>same</td>
<td>( d )</td>
<td>( f )</td>
<td>( \text{same } e )</td>
</tr>
</tbody>
</table>

- 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)

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 \( j' \).

<table>
<thead>
<tr>
<th>same</th>
<th>( e )</th>
<th>( \text{step } j' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>same</td>
<td>( f )</td>
<td>( S' )</td>
</tr>
</tbody>
</table>

• Case 3a: \( g = e \).
  \( S' \) agrees with \( S_{FF} \) through first \( j+1 \) steps
  Can’t happen with FF since there must be a request for \( f \) before \( e \).

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

Farthest-in-future: analysis

<table>
<thead>
<tr>
<th>same</th>
<th>( e )</th>
<th>( \text{same } e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S' )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

**Theorem.** 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 \( \alpha \), \( LRU(\alpha) \leq k \cdot FF(\alpha) + k \).
- Randomized marking is \( O(\log k) \)-competitive.

see \textbf{SECTION 13.8}