Chapter 4: Greedy Algorithms
Greed is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit.

- Gordon Gecko (Michael Douglas)
Intro: Coin Changing
Coin Changing

**Goal.** Given currency denominations: 1, 5, 10, 25, 100, give change to customer using *fewest* number of coins.

**Ex:** 34¢

Cashier's algorithm. At each step, give the *largest* coin valued ≤ the amount to be paid.

**Ex:** $2.89
Coin-Changing: Does Greedy Always Work?

Observation. Greedy is sub-optimal for US postal denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

Counterexample. 140¢.
- Greedy: 100, 34, 1, 1, 1, 1, 1, 1.
- Optimal: 70, 70.

Algorithm is “Greedy”, but also short-sighted – attractive choice now may lead to dead ends later.

Correctness is key!
“Greedy Algorithms”
what they are

Pros
intuitive
often simple
often fast

Cons
often incorrect!

Proofs are crucial. 3 (of many) techniques:
stay ahead
structural
exchange arguments
4.1 Interval Scheduling

Proof Technique 1: “greedy stays ahead”
Interval Scheduling

Interval scheduling.
- Job $j$ starts at $s_j$ and finishes at $f_j$.
- Two jobs compatible if they don’t overlap.
- Goal: find maximum subset of mutually compatible jobs.
Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

- What order?
- Does that give best answer?
- Why or why not?
- Does it help to be greedy about order?
Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

[Earliest start time] Order jobs by ascending start time $s_j$

[Earliest finish time] Order jobs by ascending finish time $f_j$

[Shortest interval] Order jobs by ascending interval length $f_j - s_j$

[Longest Interval] Reverse of the above

[Fewest conflicts] For each job $j$, let $c_j$ be the count the number of jobs in conflict with $j$. Order jobs by ascending $c_j$
Can You Find Counterexamples?

E.g., Longest Interval:

_________________________________________________________________________

_________________________________________________________________________

Others?:
Interval Scheduling: Greedy Algorithms

**Greedy template.** Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

- Breaks earliest start time
- Breaks shortest interval
- Breaks fewest conflicts
Greedy algorithm. Consider jobs in *increasing order of finish time*. Take each job provided it’s compatible with the ones already taken.

```plaintext
Sort jobs by finish times so that
f_1 \leq f_2 \leq \ldots \leq f_n.

\text{jobs selected}
A \leftarrow \emptyset

for j = 1 to n {
    if (job j compatible with A)
        A \leftarrow A \cup \{j\}
}

return A
```

Implementation. \(O(n \log n)\).
- Remember job \(j^*\) that was added last to \(A\).
- Job \(j\) is compatible with \(A\) if \(s_j \geq f_{j^*}\).
Interval Scheduling

Time

0 1 2 3 4 5 6 7 8 9 10 11

A
B
C
D
E
F
G
H
Interval Scheduling

The diagram illustrates an example of interval scheduling. Different tasks labeled A, B, C, D, E, F, G, and H are scheduled over time. Each interval represents a task with a start and end time. The diagram shows how tasks are sequenced and scheduled without overlapping intervals.
Interval Scheduling
Interval Scheduling

Time

A
B
C
D
E
F
G
H
Interval Scheduling
Interval Scheduling
Interval Scheduling

Time
Interval Scheduling: Correctness

**Theorem.** *Earliest Finish First* Greedy algorithm is optimal.

**Pf.** ("greedy stays ahead")

Let $g_1, ..., g_k$ be greedy’s job picks, $j_1, ..., j_m$ those in some optimal solution.

Show $f(g_r) \leq f(j_r)$ by induction on $r$.

**Basis:** $g_1$ chosen to have min finish time, so $f(g_1) \leq f(j_1)$

**Ind:** $f(g_r) \leq f(j_r) \leq s(j_{r+1})$, so $j_{r+1}$ is among the candidates considered by greedy when it picked $g_{r+1}$, & it picks min finish, so $f(g_{r+1}) \leq f(j_{r+1})$

Similarly, $k \geq m$, else $j_{k+1}$ is among (nonempty) set of candidates for $g_{k+1}$.
4.1 Interval Partitioning

Proof Technique 2: “Structural”
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 4 classrooms to schedule 10 lectures.
Interval Partitioning as Interval Graph Coloring

Vertices = classes; 
edges = conflicting class pairs; 
different colors = different assigned rooms

Note: graph coloring is very hard in general, but graphs corresponding to interval intersections are a much simpler special case.
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:** Same classes, but this schedule uses only 3 rooms.
**Interval Partitioning: A “Structural” 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 is optimal.

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

![Diagram of schedules and time intervals](image-url)
Greedy algorithm. Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

Sort intervals by start time so \( s_1 \leq s_2 \leq ... \leq s_n \).

\[
\begin{align*}
d &\leftarrow 0 \quad \text{number of allocated classrooms} \\
\text{for } j = 1 \text{ to } n \{ \\
&\quad \text{if (lect } j \text{ is compatible with some room } k, 1 \leq k \leq d) } \\
&\quad \quad \text{schedule lecture } j \text{ in classroom } k \\
&\quad \text{else} \\
&\quad \quad \text{allocate a new classroom } d + 1 \\
&\quad \quad \text{schedule lecture } j \text{ in classroom } d + 1 \\
&\quad \quad d \leftarrow d + 1
\end{align*}
\]

Implementation? Run-time?

Exercises
Observation. Earliest Start First Greedy algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Earliest Start First Greedy algorithm is optimal. 

Pf (exploit structural property).

- Let \( d \) = number of rooms the greedy algorithm allocates.
- Classroom \( d \) is opened because we needed to schedule a job, say \( j \), that is incompatible with all \( d-1 \) previously used classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than \( s_j \).
- Thus, \( d \) lectures overlap at time \( s_j + \varepsilon \), i.e. depth \( \geq d \)
- “Key observation” on earlier slide \( \Rightarrow \) all schedules use \( \geq \) depth rooms, so \( d = \) depth and greedy is optimal.
4.2 Scheduling to Minimize Lateness

Proof Technique 3: “Exchange” Arguments
Minimizing lateness problem.

- Single resource processes one job at a time.
- Job $j$ requires $t_j$ units of processing time & 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 to minimize $\max$ lateness $L = \max \ell_j$.

Ex:

<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>

lateness = 2 lateness = 0 max lateness = 6

$\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\hline
d_3 = 9 & \hline
d_2 = 8 & \hline
d_6 = 15 & \hline
d_1 = 6 & \hline
d_5 = 14 & \hline
d_4 = 9 & \hline
\end{array}$
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 in ascending order of processing time $t_j$.

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$d_j$</td>
<td>100</td>
<td>10</td>
</tr>
</tbody>
</table>
```

counterexample

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

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$d_j$</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
```

counterexample
Greedy algorithm. Earliest deadline first.

Minimizing Lateness: Greedy Algorithm

Sort $n$ jobs by deadline 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 $[t, t + t_j]$

$s_j \leftarrow t$, $f_j \leftarrow t + t_j$

$t \leftarrow t + t_j$

output intervals $[s_j, f_j]$

max lateness = 1
Proof Strategy

A schedule is an ordered list of jobs

Suppose $S_1$ is any schedule

Let $G$ be the/a schedule produced by the greedy algorithm

To show: $\text{Lateness}(S_1) \geq \text{Lateness}(G)$

Idea: find a series of simple changes that successively transform $S_1$ into other schedules that are more and more like $G$, each better than the last, until we reach $G$. I.e.

$$\text{Lateness}(S_1) \geq \text{Lateness}(S_2) \geq \text{Lateness}(S_3) \geq \ldots \geq \text{Lateness}(G)$$

If it works for any starting $S_1$, it will work for an optimal $S_1$, so $G$ is optimal

HOW?: exchange pairs of jobs
Minimizing Lateness: No Idle Time

Notes:

1. There is an optimal schedule with no idle time.

2. The greedy schedule has no idle time.
Minimizing Lateness: Inversions

Def. An *inversion* in schedule S is a pair of jobs i and j s.t.: deadline i < deadline j but j scheduled before i.

- Greedy schedule has no inversions.
- Claim: If a schedule has an inversion, it has an adjacent inversion, i.e., a pair of inverted jobs scheduled consecutively.
  (Pf: If j & i aren’t consecutive, then look at the job k scheduled right after j. If \( d_k < d_j \), then \((j,k)\) is a consecutive inversion; if not, then \((k,i)\) is an inversion, & nearer to each other - repeat.)
Minimizing Lateness: Inversions

**Def.** An *inversion* in schedule S is a pair of jobs i and j s.t.: deadline i < deadline j but j scheduled before i.

- **Claim:** Swapping an *adjacent* inversion reduces # inversions by 1 (exactly)

**Pf:** Let i,j be an adjacent inversion. For any pair (p,q), inversion status of (p,q) is unchanged by i↔j swap unless \( \{p, q\} = \{i, j\} \), and the i,j inversion is removed by that swap.
Minimizing Lateness: Inversions

**Def.** An *inversion* in schedule $S$ is a pair of jobs $i$ and $j$ s.t.: deadline $i < j$ but $j$ scheduled before $i$.

![Diagram showing inversion before and after swap]

**Claim.** Swapping two adjacent, inverted jobs does not increase the max lateness.

**Pf.** Let $\ell / \ell'$ be the lateness before / after swap, resp.

- $\ell'_k = \ell_k$ for all $k \neq i, j$
- $\ell'_i \leq \ell_i$
- If job $j$ is now late:

\[
\ell'_j = f'_j - d_j \quad \text{(definition)}
\]

\[
= f_i - d_j \quad \text{($j$ finishes at time $f_i$)}
\]

\[\leq f_i - d_i \quad \text{($d_i \leq d_j$)}
\]

\[= \ell_i \quad \text{(definition)}
\]

(j had later deadline, so is less tardy than $i$ was)
Claim. All idle-free, inversion-free schedules $S$ have the same max lateness.

Pf. If $S$ has no inversions, then deadlines of scheduled jobs are monotonically nondecreasing (i.e., increase or stay the same) as we walk through the schedule from left to right. Two such schedules can differ only in the order of jobs with the same deadlines. Within a group of jobs with the same deadline, the max lateness is the lateness of the last job in the group - order within the group doesn’t matter.
Minimizing Lateness: Correctness of Greedy Algorithm

**Theorem.** Greedy schedule $S$ is optimal

**Pf.** Let $S^*$ be an optimal schedule with the fewest number of inversions among all optimal schedules. Can assume $S^*$ has no idle time. 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, $S^*$ has no inversions. Hence $\text{Lateness}(S) = \text{Lateness}(S^*)$
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. (Part of the cleverness is deciding what’s “good.”)

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. (Cleverness here is usually in finding a useful structural characteristic.)

Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality. (Cleverness usually in choosing which pair to swap.)

(In all 3 cases, proving these claims may require cleverness.)
4.3 Optimal Caching

\[1\text{cache}\]

Pronunciation: 'kash
Function: noun
Etymology: French, from cache to press, hide

a hiding place especially for concealing and preserving provisions or implements

\[2\text{cache}\]

Function: transitive verb

to place, hide, or store in a cache

- Webster’s Dictionary
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.

<table>
<thead>
<tr>
<th>current cache:</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
</table>

| future queries: | g | a | b | c | e | d | a | b | b | a | c | d | e | a | f | a | d | e | f | g | h | . . . |

- "current cache:" | a | b | c | d | e | f |
- "future queries:" | g | a | b | c | e | d | a | b | b | a | c | d | e | a | f | a | d | e | f | g | h | . . . |
- cache miss
- eject this one

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

Motivation: “Online” problem is typically what’s needed in practice - decide what to evict without seeing the future. How to evaluate such an alg? Fewer misses is obviously better, but how few? FF is a useful benchmark - best online alg is unknown, but it’s no better than FF, so online performance close to FF’s is the best you can hope for.
You’ve seen this in prerequisite courses, so this section and next two on min spanning tree are review. I won’t lecture on them, but you should review the material. Both, but especially shortest paths, are common problems, having many applications. (And frequent fodder for job interview questions...)
Shortest Path Problem

Shortest path network.
- Directed graph G = (V, E).
- Source s, destination t.
- Length $\ell_e = \text{length of edge } e$.

Shortest path problem: find shortest directed path from s to t.

Cost of path = sum of edge costs in path

Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 48.
Dijkstra's Algorithm

Dijkstra's algorithm.

- Maintain a set of explored nodes $S$ for which we have determined the shortest path distance $d(u)$ from $s$ to $u$.
- Initialize $S = \{s\}$, $d(s) = 0$.
- Repeatedly choose unexplored node $v$ which minimizes

$$
\pi(v) = \min_{e = (u, v): u \in S} \{d(u) + \ell_e\},
$$

add $v$ to $S$, and set $d(v) = \pi(v)$.  

shortest path to some $u$ in explored part, followed by a single edge $(u, v)$
Dijkstra's Algorithm

Dijkstra's algorithm.
- Maintain a set of explored nodes $S$ for which we have determined the shortest path distance $d(u)$ from $s$ to $u$.
- Initialize $S = \{ s \}$, $d(s) = 0$.
- Repeatedly choose unexplored node $v$ which minimizes $\pi(v) = \min_{e = (u, v) : u \in S} d(u) + \ell_e$.

Add $v$ to $S$, and set $d(v) = \pi(v)$.

![Diagram of Dijkstra's Algorithm](image-url)