Chapter 4

Greedy Algorithms
4.1 Interval Scheduling
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.

- **[Earliest start time]** Consider jobs in ascending order of start time $s_j$.

- **[Earliest finish time]** Consider jobs in ascending order of finish time $f_j$.

- **[Shortest interval]** Consider jobs in ascending order of interval length $f_j - s_j$.

- **[Fewest conflicts]** For each job, count the number of conflicting jobs $c_j$. Schedule in ascending order of conflicts $c_j$. 
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
**Interval Scheduling: Greedy Algorithm**

**Greedy algorithm.** Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

Sort jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).

1. jobs selected
2. \( A \leftarrow \emptyset \)
3. for \( j = 1 \) to \( n \) {
   4. if (job \( j \) compatible with \( A \))
      5. \( A \leftarrow A \cup \{j\} \)
   6. }
7. return \( A \)

**Implementation.** \( O(n \log n) \).

- Remember job \( j^* \) that was added last to \( A \).
- Job \( j \) is compatible with \( A \) if \( s_j \geq f_j^* \).
**Theorem.** Greedy 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 the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of $r$.

---

**Diagram:**

- **Greedy:**
  - $i_1$
  - $i_1$
  - $i_r$
  - $i_{r+1}$

- **OPT:**
  - $j_1$
  - $j_2$
  - $j_r$
  - $j_{r+1}$
  - $\ldots$

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

- Job $i_{r+1}$ finishes before $j_{r+1}$.
Interval Scheduling: Analysis

**Theorem.** Greedy 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 the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of $r$.

Proof by induction that for each $r \geq 1$ the greedy algorithm stays ahead of the optimal algorithm, but contradicts maximality of $r$.

![Diagram showing the greedy and optimal schedules](diagram.png)
4.1 Interval Partitioning
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

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 $= 3 \implies$ schedule below is optimal.

\[\text{a, b, c all contain 9:30}\]

**Q.** Does there always exist a schedule equal to depth of intervals?
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 \( s_1 \leq s_2 \leq \ldots \leq s_n \).

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

\[
\text{for } j = 1 \text{ to } n \{
\quad \text{if (lecture } j \text{ is compatible with some classroom } k) }
\quad \text{schedule lecture } j \text{ in classroom } k
\quad \text{else}
\quad \quad \text{allocate a new classroom } d + 1
\quad \quad \text{schedule lecture } j \text{ in classroom } d + 1
\quad \quad d \leftarrow d + 1
\}
\]

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.

Pf.
- 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.
4.2 Scheduling to Minimize Lateness
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 $d_j$.
- 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 $L = \max \ell_j$.

Ex:

<table>
<thead>
<tr>
<th>$d_j$</th>
<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>
<td>6</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccc}
\hline
 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
 t_j & 3 & 2 & 1 & 4 & 3 & 2 \\
 d_j & 6 & 8 & 9 & 9 & 14 & 15 \\
\hline
\end{array}
\]

\[
\text{lateness} = 2 \quad \text{lateness} = 0 \quad \text{max lateness} = 6
\]
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 $d_j - t_j$. 
**Greedy template.** Consider jobs in some order.

- **[Shortest processing time first]** Consider jobs 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>

- **[Smallest slack]** Consider jobs 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>2</td>
<td>10</td>
</tr>
</tbody>
</table>
Minimizing Lateness: Greedy Algorithm

**Greedy algorithm.** Earliest deadline first.

Sort the 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

<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>8</td>
<td>9</td>
<td>9</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

| 7       | 8       | 9       | 10      | 11      | 12      |
| 13      | 14      | 15      |         |         |         |
**Minimizing Lateness: No Idle Time**

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

**Observation.** The greedy schedule has no idle time.
Def. 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.

Observation. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.
Minimizing Lateness: Inversions

Def. 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.

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:

$$
\ell'_j = f'_j - d_j \quad \text{(definition)}
= f_i - d_j \quad \text{($j$ finishes at time $f_i$)}
\leq f_i - d_i \quad \text{($i < j$)}
\leq \ell_i \quad \text{(definition)}
$$
Minimizing Lateness: Analysis of Greedy Algorithm

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

**Pf.** Define $S^*$ to be an optimal schedule that has the fewest number of inversions, and let's see what happens.

- Can assume $S^*$ has no idle time.
- If $S^*$ has no inversions, then $S = S^*$.
- 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 $S^*$
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.
4.3 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 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.

Algorithm and theorem are intuitive; proof is subtle.

Theorem. [Bellady, 1960s] FF is optimal eviction schedule.
4.4 Shortest Paths in a Graph

shortest path from Princeton CS department to Einstein's house
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 $s$-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 (go through edges)
  \[ \pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e, \]
  add $v$ to $S$, and set $d(v) = \pi(v)$.
- Add only node for which $\pi(v)$ is minimum
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 the shortest path to some \( u \) in explored part, followed by a single edge \((u, v)\).

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

add \( v \) to \( S \), and set \( d(v) = \pi(v) \).
- Running time ?
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 $d(v) = \pi(v)$.
- Running time $O(mn)$ - simple implementation
- Can we do better?

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

**Invariant.** For each node $u \in S$, $d(u)$ is the length of the shortest $s$-$u$ path.

**Pf.** (by induction on $|S|$)

**Base case:** $|S| = 1$ is trivial.

**Inductive hypothesis:** Assume true for $|S| = k \geq 1$.

- Let $v$ be next node added to $S$, and let $u$-$v$ be the chosen edge.
- The shortest $s$-$u$ path plus $(u, v)$ is an $s$-$v$ path of length $\pi(v)$.
- Consider any $s$-$v$ path $P$. We'll see that it's no shorter than $\pi(v)$.
- Let $x$-$y$ be the first edge in $P$ that leaves $S$, and let $P'$ be the subpath to $x$.
- $P$ is already too long as soon as it leaves $S$.

\[
\ell(P) \geq \ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v)
\]

- nonnegative weights
- inductive hypothesis
- defn of $\pi(y)$
- Dijkstra chose $v$ instead of $y$
Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain $\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e$.

- Next node to explore = node with minimum $\pi(v)$.
- When exploring $v$, for each incident edge $e = (v, w)$, update
  \[ \pi(w) = \min \{ \pi(w), \pi(v) + \ell_e \}. \]

**Efficient implementation.** Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.

<table>
<thead>
<tr>
<th>PQ Operation</th>
<th>Dijkstra</th>
<th>Array</th>
<th>Binary heap</th>
<th>d-way Heap</th>
<th>Fib heap †</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$d \log_d n$</td>
<td>$1$</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$d \log_d n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>ChangeKey</td>
<td>$m$</td>
<td>1</td>
<td>$\log n$</td>
<td>$\log_d n$</td>
<td>$1$</td>
</tr>
<tr>
<td>isEmpty</td>
<td>$n$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>$n^2$</td>
<td>$m \log n$</td>
<td>$m \log_{m/n} n$</td>
<td>$m + n \log n$</td>
<td></td>
</tr>
</tbody>
</table>

† Individual ops are amortized bounds
Extra Slides
Coin Changing

Greed is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit.

- Gordon Gecko (Michael Douglas)
**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.
**Coin-Changing: Greedy Algorithm**

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

---

**Sort** coins denominations by value: $c_1 < c_2 < \ldots < c_n$.

\[
\begin{align*}
\text{coins selected} \\
S & \leftarrow \emptyset \\
\text{while } (x \neq 0) \\
& \quad \text{let } k \text{ be largest integer such that } c_k \leq x \\
& \quad \text{if } (k = 0) \\
& \quad \quad \text{return } "\text{no solution found}" \\
& \quad x \leftarrow x - c_k \\
& \quad S \leftarrow S \cup \{k\} \\
& \} \\
\text{return } S
\end{align*}
\]

---

Q. Is cashier's algorithm optimal?
Coin-Changing: Analysis of Greedy Algorithm

Theorem. Greed is optimal for U.S. coinage: 1, 5, 10, 25, 100.

Pf. (by induction on x)

- Consider optimal way to change \( c_k \leq x < c_{k+1} \) : 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 greedy algorithm.

<table>
<thead>
<tr>
<th>( k )</th>
<th>( c_k )</th>
<th>All optimal solutions must satisfy</th>
<th>Max value of coins 1, 2, ..., ( k-1 ) in any OPT</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>
Observation. Greedy algorithm 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.