Chapter 4

Greedy Algorithms
4.1 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.
**Interval Scheduling: Greedy Algorithms**

*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?
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.

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

A ← ∅
for j = 1 to n {
    if (job j compatible with A)
        A ← A ∪ {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
Interval Scheduling
Interval Scheduling

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

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

The diagram illustrates the scheduling of activities represented by intervals on a timeline. Each activity is represented by a colored bar, with the length of the bar indicating the duration of the activity. The timeline is divided into time slots, with each slot representing a unit of time.

Activities:
- A: [3, 7]
- B: [1, 3]
- C: [4, 6]
- D: [4, 8]
- E: [5, 7]
- F: [6, 10]
- G: [7, 9]
- H: [8, 11]

The goal of interval scheduling is to maximize the number of activities that can be scheduled without overlapping. In this example, all activities can be scheduled without any conflicts.

Time:
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
Interval Scheduling

![Interval Scheduling Diagram]

- Time scale from 0 to 11
- Tasks labeled A to H
- Scheduling intervals for each task
Interval Scheduling
Interval Scheduling

The diagram shows a time-axis from 0 to 11, with intervals for different activities labeled A, B, C, D, E, F, G, and H. The intervals are shaded in different colors to represent their time slots:

- A: Red, from 0 to 5
- B: Yellow, from 1 to 4
- C: Blue, from 3 to 5
- D: Green, from 1 to 7
- E: Orange, from 5 to 7
- F: Pink, from 7 to 9
- G: Blue, from 8 to 10
- H: Grey, from 8 to 11
Interval Scheduling: Correctness

**Theorem.** Greedy algorithm is optimal.

**Pf.** ("greedy stays ahead")
Let \( i_1, i_2, \ldots i_k \) be jobs picked by greedy, \( j_1, j_2, \ldots j_m \) those in some optimal solution
Show \( f(i_r) \leq f(j_r) \) by induction on \( r \).

- **Basis:** \( i_1 \) chosen to have min finish time, so \( f(i_1) \leq f(j_1) \)
- **Ind:** \( f(i_r) \leq f(j_r) \leq s(j_{r+1}) \), so \( j_{r+1} \) is among the candidates considered by greedy when it picked \( i_{r+1} \), & it picks min finish, so \( f(i_{r+1}) \leq f(j_{r+1}) \)

Similarly, \( k \geq m \), else \( j_{k+1} \) is among (nonempty) set of candidates for \( i_{k+1} \)

---

**Diagram:**

- **Greedy:** \( i_1 \) \( i_1 \) \( i_r \) \( i_{r+1} \)
- **OPT:** \( j_1 \) \( j_2 \) \( j_r \) \( j_{r+1} \) \( \ldots \)

**Note:** Job \( j_{r+1} \) starts after \( i_r \) ends, so included in min(...)
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 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: 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 ≥ depth.

**Ex:** Depth of schedule below = 3 ⇒ schedule below is optimal.

**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 $s_1 \leq s_2 \leq \ldots \leq s_n$. Let $d \leftarrow 0$ number of allocated classrooms

for $j = 1$ to $n$
  if (lect $j$ is compatible with some classroom $k$, $1 \leq k \leq d$)
    schedule lecture $j$ in classroom $k$
  else
    allocate a new classroom $d + 1$
    schedule lecture $j$ in classroom $d + 1$
    $d \leftarrow d + 1$

Implementation? Run-time?
Next HW
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 \( d-1 \) previously used 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 \( s_j + \varepsilon \), i.e. depth \( \geq d \).
- “Key observation” \( \Rightarrow \) all schedules use \( \geq \) depth classrooms, so \( d = \) depth and greedy is optimal.
Interval Partitioning: Alt Proof (exchange argument)

When 4th room added, room 1 was free; why not swap it in there?
(A: it conflicts with later stuff in schedule, which dominoes)
But: room 4 schedule after 11:00 is conflict-free; so is room 1 schedule, so could swap both post-11:00 schedules
Why does it help? Delays needing 4th room; repeat.

**Cleaner**: “Let S* be an opt sched with latest use of last room. When that room is added, all others in use, else we could swap, contradicting ‘latest’”
4.2 Scheduling to Minimize Lateness
Scheduling to Minimize 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: \( l_j = \max \{ 0, f_j - d_j \} \).
- Goal: schedule all jobs to minimize maximum lateness \( L = \max l_j \).

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>

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| \( d_3 = 9 \) | \( d_2 = 8 \) | \( d_6 = 15 \) | \( d_1 = 6 \) | \( d_5 = 14 \) | \( d_4 = 9 \) |

lateness = 2  lateness = 0  max lateness = 6
Minimizing Lateness: Greedy Algorithms

Greedy template. 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 $t_j$.

![Table]

<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 jobs in ascending order of slack $d_j - t_j$.

![Table]

<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>

counterexample
Minimizing Lateness: Greedy Algorithm

**Greedy algorithm.** Earliest deadline first.

Sort $n$ jobs by deadline so that $d_1 \leq d_2 \leq ... \leq d_n$

$t \leftarrow 0$

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 $[s_j, f_j]$

<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>

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

max lateness = 1
Minimizing Lateness: No Idle Time

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

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

**Def.** An *inversion* in schedule $S$ is a pair of jobs $i$ and $j$ such that: deadline $i < j$ 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.
(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.)

**Observation.** Swapping *adjacent* inversion reduces # inversions by 1 (exactly)
Def. An inversion in schedule $S$ is a pair of jobs $i$ and $j$ such that: deadline $i < j$ but $j$ scheduled before $i$.

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

\begin{itemize}
\item $\ell'_k = \ell_k$ for all $k \neq i, j$
\item $\ell'_i \leq \ell_i$
\item If job $j$ is now late:
\end{itemize}

\begin{align*}
\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)}
\end{align*}

(j had later deadline, so is less tardy than $i$ was)

only $j$ moves later, but it’s no later than $i$ was, so max not increased
Minimizing Lateness: No Inversions

Claim. All 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., they 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. 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. But then $\text{Lateness}(S) = \text{Lateness}(S^*)$
**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.

*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.
4.3 Optimal Caching

1. cache
   Pronunciation: 'kash
   Function: noun
   Etymology: French, from cacher to press, hide
   a hiding place especially for concealing and preserving provisions or implements

2. cache
   Function: transitive verb
   to place, hide, or store in a cache

- Webster’s Dictionary
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.

- 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.
You’ve seen this in 373, 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 with many applications.
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 $s-2-3-5-t = 9 + 23 + 2 + 16 = 48$. 

cost of path = sum of edge costs in path
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:**

- Graph with nodes $s$, $u$, and $v$.
- Edge weights $\ell_e$.
- Path from $s$ to $v$ via $u$. 

**Note:** 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 $d(v) = \pi(v)$.

shortest path to some $u$ in explored part, followed by a single edge $(u, v)$
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: \( c_1 < c_2 < \ldots < c_n \).

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

**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 \( c_1, ..., c_{k-1} \) 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>
Coin-Changing: Analysis of Greedy Algorithm

**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.