Chapter 4
Greedy Algorithms

Coin Changing

Coin Changing: Cashier's Algorithm

Goal. Given currency denominations: 1, 5, 10, 25, 100, 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: Postal Worker’s Algorithm

Goal. Given postage denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500, dispense amount to customer using fewest number of stamps.

Ex: $1.40.

Postal worker’s algorithm. At each iteration, add stamp of the largest value that does not take us past the amount to be dispensed.

Ex: $1.40.
Coin-Changing

**Observation.** Postal worker’s algorithm is not optimal for U.S. postage.

**Theorem.** Cashier’s algorithm is optimal for U.S. coinage.

**Pf sketch.**

<table>
<thead>
<tr>
<th>optimal solution must satisfy</th>
<th>( P \leq 4 )</th>
<th>( P \leq 4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N \leq 1 )</td>
<td>( P + 5N = 9 )</td>
<td></td>
</tr>
<tr>
<td>( N + D \leq 2 )</td>
<td>( P + 5N = 10D \leq 24 )</td>
<td></td>
</tr>
<tr>
<td>( Q = 3 )</td>
<td>( P + 5N + 10D + 25Q \leq 99 )</td>
<td></td>
</tr>
</tbody>
</table>

\( \Rightarrow \) if amount to change is \( \leq \$k \), optimal solution uses \( \$k \) dollar coin.

---

4.1 Interval Scheduling (CLRS 16.1)

**Interval Scheduling.** 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.

**Theorem.** Greedy algorithm is optimal.

**Pf.** (by contradiction)
- Assume greedy is not optimal, and let’s see what happens.
- Let \( i_1, i_2, \ldots, i_n \) denote set of jobs selected by greedy.
- Let \( J_1, J_2, \ldots, J_m \) denote set of jobs in the optimal solution with \( i_1 = J_1, i_2 = J_2, \ldots, i_n = J_n \) for the largest possible value of \( r \).

Consider jobs in some natural order.

\[
\text{Job } i_r \text{ finishes before } J_{r+1} \]

\[
\text{OPT: } \quad J_1 \quad J_2 \quad \ldots \quad J_{r+1} \]

\[
\text{Greedy: } \quad i_1 \quad i_2 \quad \ldots \quad i_r \quad i_{r+1} \]

why not replace job \( J_{r+1} \) with job \( i_r \)?

Implementation. \( O(n \log n) \).
- Remember job \( j^* \) that was added last to \( A \).
- Job \( j \) is compatible with \( A \) if \( s_j \leq f_j \).

Interval Scheduling: Analysis

**Theorem.** Greedy algorithm is optimal.

**Pf.** (by contradiction)
- Assume greedy is not optimal, and let’s see what happens.
- Let \( i_1, i_2, \ldots, i_n \) denote set of jobs selected by greedy.
- Let \( J_1, J_2, \ldots, J_m \) denote set of jobs in the optimal solution with \( i_1 = J_1, i_2 = J_2, \ldots, i_n = J_n \) for the largest possible value of \( r \).

\[
\text{OPT: } \quad J_1 \quad J_2 \quad \ldots \quad J_{r+1} \]

\[
\text{Greedy: } \quad i_1 \quad i_2 \quad \ldots \quad i_r \quad i_{r+1} \]

solution still feasible and optimal, but contradicts maximality of \( r \).
4.1 Interval Partitioning

**Interval partitioning.**
- Lecture \( j \) starts at \( s_j \) and finishes at \( f_j \).
- **Goal:** find min 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.

**Ex:** This schedule uses only 3.

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

**Key observation.** Number of classrooms needed \( \geq \) depth.
Interval Partitioning: Lower Bound on Optimal Solution

Ex. Depth of schedule below = 3 ⇒ schedule below is optimal.
a, b, c all contain 9:30

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

![Diagram showing depth = 3]

Interval Partitioning: Greedy Algorithm

Greedy algorithm. Consider lectures in increasing order of start time:
assign lecture to any compatible classroom.

\[
\text{Sort intervals by starting time so that } s_1 \leq s_2 \leq \ldots \leq s_n.
\]

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

\[
\text{for } j = 1 \text{ to } n \{
\]

\[
\text{if (lecture } j \text{ is compatible with some classroom } k)
\]

\[
\text{schedule lecture } j \text{ in classroom } k
\]

\[
\text{else}
\]

\[
\text{allocate a new classroom } d + 1
\]

\[
\text{schedule lecture } j \text{ in classroom } d + 1
\]

\[
d \leftarrow d + 1
\]

\[
\}
\]

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.
- These \(d\) jobs each end after \(s_j\).
- 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 \(s_j + e\).
- Key observation ⇒ 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: \( l_j = \max \{ 0, \ f_j - d_j \} \)
- Goal: schedule all jobs to minimize maximum lateness \( L = \max l_j \).

Ex:

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

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 \( t_j \).
- [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 algorithm. Earliest deadline first.

Sort \( n \) jobs by deadline so that \( d_1 \leq d_2 \leq \ldots \leq d_n \),
\( t \leftarrow 0 \)
\( \text{for } j = 1 \text{ to } n \)
\( \text{Assign job } j \text{ to interval } [t, t + t_j] \)
\( s_j = t, f_j = t + t_j \)
\( t = t + t_j \)
\( \text{output intervals } [s_j, f_j] \)

Counterexample

<table>
<thead>
<tr>
<th>( t_j )</th>
<th>( d_j )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
</tr>
</tbody>
</table>

Max lateness = 1
Minimizing Lateness: No Idle Time

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

<table>
<thead>
<tr>
<th>d = 4</th>
<th>d = 6</th>
<th>d = 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2</td>
<td>3 4 5 6</td>
<td>7 8 9 10 11</td>
</tr>
</tbody>
</table>

**Observation.** The greedy 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 scheduled before i.

![Diagram]

**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: 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* •

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

**Pf.** Let ℓ be the lateness before the swap, and let ℓ’ be it afterwards.

- ℓ’k = ℓk for all k ≠ i, j
- ℓ’i ≤ ℓi
- If job j is late:

\[
\ell’_j = f_j - d_j \quad \text{(definition)}
\]

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

\[
\leq f_i - d_i \quad (i < j)
\]

\[
\leq \ell_i \quad \text{(definition)}
\]
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. Kruskal, Prim, Dijkstra, Huffman, ...

Algorithmic Paradigms

Greed. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Dynamic Programming History

Bellman. [1950s] Pioneered the systematic study of dynamic programming.

Etymology.
- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.

"it's impossible to use dynamic in a pejorative sense"
"something not even a Congressman could object to"

Dynamic Programming Applications

Areas.
- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ...

Some famous dynamic programming algorithms.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

6.1 Weighted Interval Scheduling

Weighted Interval Scheduling

Weighted interval scheduling problem.
- Job \( j \) starts at \( s_j \), finishes at \( f_j \), and has weight or value \( v_j \).
- Two jobs compatible if they don’t overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

Recall. Greedy algorithm works if all weights are 1.
- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

Unweighted Interval Scheduling Review
Weigthed Interval Scheduling

Notation. Label jobs by finishing time: \( f_1 \leq f_2 \leq \ldots \leq f_n \).

Def. \( p(j) \) = largest index \( i < j \) such that job \( i \) is compatible with \( j \).

Ex: \( p(8) = 5 \), \( p(7) = 3 \), \( p(2) = 0 \).

**Input:** \( n, s_1, \ldots, s_n, e_1, \ldots, e_n, v_1, \ldots, v_n \)

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

**Compute** \( p(1), p(2), \ldots, p(n) \)

**Compute-Opt(\( j \))**

\[
\text{if } (j = 0) \text{ return } 0 \\
\text{else return } \max\{v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1)\}
\]

Dynamic Programming: Binary Choice

Notation. \( \text{OPT}(j) = \text{value of optimal solution to the problem consisting of job requests } 1, 2, \ldots, j \).

- **Case 1:** \( \text{OPT} \) selects job \( j \).
  - collect profit \( v_j \)
  - can’t use incompatible jobs \{ \( p(j) + 1, p(j) + 2, \ldots, j - 1 \) \}
  - must include optimal solution to problem consisting of remaining compatible jobs \( 1, 2, \ldots, p(j) \)

- **Case 2:** \( \text{OPT} \) does not select job \( j \).
  - must include optimal solution to problem consisting of remaining compatible jobs \( 1, 2, \ldots, j-1 \)

\[
\text{OPT}(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \{ v_j + \text{OPT}(p(j)), \text{OPT}(j-1) \} & \text{otherwise}
\end{cases}
\]

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems \( \Rightarrow \) exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.
Weighted Interval Scheduling: Memoization

**Memoization.** Store results of each sub-problem in a cache; lookup as needed.

**Input:** \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

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

**Compute** \( p(1), p(2), \ldots, p(n) \)

```
for j = 1 to n
    M[j] = empty
    M[j] = 0  // global array
M-Compute-Opt(j) {
    if \( M[j] \text{ is empty} \)
        M[j] = max(\( w_i + M-Compute-Opt(p[j]) \), M-Compute-Opt(j-1))
    return M[j]
}
```

Weighted Interval Scheduling: Running Time

**Claim.** Memoized version of algorithm takes \( O(n \log n) \) time.

- Sort by finish time: \( O(n \log n) \).
- Computing \( p() \): \( O(n \log n) \) via sorting by start time.

- \( M-Compute-Opt(j) \): each invocation takes \( O(1) \) time and either
  - (i) returns an existing value \( M[j] \)
  - (ii) fills in one new entry \( M[j] \) and makes two recursive calls

- Progress measure \( \Phi = \# \text{ nonempty entries of } M[] \).
  - Initially \( \Phi = 0 \), throughout \( \Phi \leq n \).
  - (ii) increases \( \Phi \) by 1 \( \Rightarrow \) at most \( 2n \) recursive calls.

- Overall running time of \( M-Compute-Opt(n) \) is \( O(n) \).  

**Remark.** \( O(n) \) if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Finding a Solution

**Q.** Dynamic programming algorithms computes optimal value.

What if we want the solution itself?

**A.** Do some post-processing.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
    if \( j = 0 \)
        output nothing
    else if \( (v_j + M[p[j]] > M[j-1]) \)
        print \( j \)
        Find-Solution(p[j])
    else
        Find-Solution(j-1)
}
```

- \# of recursive calls \( \leq n \Rightarrow O(n) \).

Weighted Interval Scheduling: Bottom-Up

**Bottom-up dynamic programming.** Unwind recursion.

**Input:** \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

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

**Compute** \( p(1), p(2), \ldots, p(n) \)

```
Iterative-Compute-Opt {
    M[0] = 0
    for j = 1 to n
        M[j] = max(\( v_j + M[p[j]] \), M[j-1])
}
```
6.4 Knapsack Problem

Knapsack Problem.
- Given \( n \) objects and a "knapsack."
- Item \( i \) weighs \( w_i > 0 \) kilograms and has value \( v_i > 0 \).
- Knapsack has capacity of \( W \) kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: \( \{3, 4\} \) has value 40.

Greedy: repeatedly add item with maximum ratio \( v_i / w_i \).
Ex: \( \{5, 2, 1\} \) achieves only value = 35 \( \Rightarrow \) greedy not optimal.

Dynamic Programming: False Start

Def. \( \text{OPT}(i) = \text{max profit subset of items} 1, ..., i \).

- Case 1: \( \text{OPT} \) does not select item \( i \).
  - \( \text{OPT} \) selects best of \( \{1, 2, ..., i-1\} \)

- Case 2: \( \text{OPT} \) selects item \( i \).
  - accepting item \( i \) does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before \( i \),
    we don’t even know if we have enough room for \( i \)

Conclusion. Need more sub-problems!

Dynamic Programming: Adding a New Variable

Def. \( \text{OPT}(i, w) = \text{max profit subset of items} 1, ..., i \) with weight limit \( w \).

- Case 1: \( \text{OPT} \) does not select item \( i \).
  - \( \text{OPT} \) selects best of \( \{1, 2, ..., i-1\} \) using weight limit \( w \)

- Case 2: \( \text{OPT} \) selects item \( i \).
  - new weight limit = \( w - w_i \)
  - \( \text{OPT} \) selects best of \( \{1, 2, ..., i-1\} \) using this new weight limit

\[
\text{OPT}(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
\text{OPT}(i-1, w) & \text{if } w_i > w \\
\max \{ \text{OPT}(i-1, w), v_i + \text{OPT}(i-1, w-w_i) \} & \text{otherwise}
\end{cases}
\]
Knapsack Problem: Bottom-Up

**Knapsack.** Fill up an n-by-W array.

```
Input: n, w₁,...,wₙ, v₁,...,vₙ
for w = 0 to W
    M[0, w] = 0
for i = 1 to n
    for w = 1 to W
        if (wᵢ > w)
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max{M[i-1, w], vᵢ + M[i-1, w-wᵢ]}
return M[n, W]
```

---

Knapsack Problem: Running Time

**Running time.** $\Theta(nW)$.
- Not poly-time in input size!
- "Pseudo-polynomial."
- Decision version of knapsack problem is NP-complete.

**Knapsack approximation algorithm.** There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum.

---

**6.6 Sequence Alignment**
String Similarity

How similar are two strings?
- occurrence
- occurrence

6 mismatches, 1 gap

Def. Alignment of minimum cost.

Sequence Alignment

Goal: Given two strings $X = x_1, x_2, \ldots, x_n$ and $Y = y_1, y_2, \ldots, y_m$ find alignment of minimum cost.

Def. An alignment $M$ is a set of ordered pairs $x_i, y_j$ such that each item occurs in at most one pair and no crossings.

Def. The pair $x_i, y_j$ and $x_i', y_j'$ cross if $i < i'$, but $j > j'$.

$$\text{cost}(M) = \sum_{(x_i, y_j) \in M} \alpha_{x_i, y_j} + \sum_{i: x_i \text{ unmatched}} \delta + \sum_{j: y_j \text{ unmatched}} \delta$$

Ex: CTACCG vs. TACATG.

Sol: $M = x_2, y_1, x_3, y_2, x_4, y_3, x_5, y_4, x_6, y_6$.

Edit Distance

Applications.
- Basis for Unix diff.
- Speech recognition.
- Computational biology.

- Gap penalty $\delta$; mismatch penalty $\alpha_{xy}$
- Cost = sum of gap and mismatch penalties.

Case 2b: $OPT$ leaves $y_j$ unmatched.
- Pay gap for $x_i$ and min cost of aligning $x_1, x_2, \ldots, x_{i-1}$ and $y_1, y_2, \ldots, y_{j-1}$

Case 2b: $OPT$ leaves $y_j$ unmatched.
- Pay gap for $y_j$ and min cost of aligning $x_1, x_2, \ldots, x_i$ and $y_1, y_2, \ldots, y_{j-1}$

Sequence Alignment: Problem Structure

Def. $OPT(i, j) =$ min cost of aligning strings $x_1, x_2, \ldots, x_i$ and $y_1, y_2, \ldots, y_j$.
- Case 1: $OPT$ matches $x_i, y_j$:
  - Pay mismatch for $x_i, y_j$ + min cost of aligning two strings $x_1, x_2, \ldots, x_{i-1}$ and $y_1, y_2, \ldots, y_{j-1}$
- Case 2a: $OPT$ leaves $x_i$ unmatched.
  - Pay gap for $x_i$ and min cost of aligning $x_1, x_2, \ldots, x_{i-1}$ and $y_1, y_2, \ldots, y_{j-1}$
- Case 2b: $OPT$ leaves $y_j$ unmatched.
  - Pay gap for $y_j$ and min cost of aligning $x_1, x_2, \ldots, x_i$ and $y_1, y_2, \ldots, y_{j-1}$

$OPT(i, j) =$

$$\begin{cases} 
  j \delta & \text{if } i = 0 \\
  \min \left( \begin{array}{c}
  \alpha_{x_i, y_j} + OPT(i-1, j-1) \\
  \delta + OPT(i-1, j) \\
  \delta + OPT(i, j-1) \\
  i \delta & \text{if } j = 0 
  \end{array} \right) & \text{otherwise}
\end{cases}$$
Sequence Alignment: Algorithm

Sequence-Alignment(m, n, x_1x_2...x_n, y_1y_2...y_n, δ, α) {
    for i = 0 to m
        M[0, i] = iα
    for j = 0 to n
        M[j, 0] = jδ
    for i = 1 to m
        for j = 1 to n
            M[i, j] = min{α(x_i, y_j) + M[i-1, j-1],
                          δ + M[i-1, j],
                          δ + M[i, j-1]}
    return M[m, n]
}

Analysis. O(mn) time and space.

English words or sentences: m, n ≤ 10.

Computational biology: m = n = 100,000. 10 billions ops OK, but 10GB array?