1 Greedy Algorithms

- We have previously discussed dynamic programming—a way of improving on inefficient divide-and-conquer algorithm:
  
  - If same subproblem is solved several times, use table to store result of a subproblem the first time it is computed and never compute it again.
  
  - Alternatively, we can think about filling up a table of subproblem solutions from the bottom.

- In divide-and-conquer (and thus dynamic programming) we used the fact that the solution to a problem depends on solutions to smaller subproblems.

- Another, simpler and often less powerful (and less well-defined), technique that uses the same feature is greediness.

- Like in the case of dynamic programming, we will introduce greedy algorithms via an example.

1.1 Activity Selection

- Problem: Given a set \( A = \{ A_1, A_2, \ldots, A_n \} \) of \( n \) activities with start and finish times \((s_i, f_i)\), \( 1 \leq i \leq n \), select maximal set \( S \) of “non-overlapping” activities.

  - One can think of the problem as corresponding to scheduling the maximal number of classes (given their start and finish times) in one classroom.

- Solution:
  
  - Sort activity by finish time (let \( A_1, A_2, \ldots, A_n \) denote sorted sequence)
  
  - Pick first activity \( A_1 \)
  
  - Remove all activities with start time before finish time of \( A_1 \)
  
  - Recursively solve problem on remaining activities.
- Program:

\[
\begin{align*}
\text{Sort } A \text{ by finish time} \\
S &= \{A_1\} \\
j &= 1 \\
\text{FOR } i = 2 \text{ to } n \text{ DO} \\
&\quad \text{IF } s_i \geq s_j \text{ THEN} \\
&\quad \quad S = S \cup \{A_i\} \\
&\quad \quad j = i \\
&\quad \FI \\
\text{OD}
\end{align*}
\]

- Example:

- 11 activities sorted by finish time: (1, 4), (3, 5), (0, 6), (5, 7), (3, 8), (5, 9), (6, 10), (8, 11), (8, 12), (2, 13), (12, 14)

- Running time is obviously \( O(n \log n) \).
• Is algorithm correct?
  – Output is set of non-overlapping activities, but is it the largest possible?

• Proof of correctness:
  – Given activities \( A = \{A_1, A_2, \cdots, A_n\} \) ordered by finish time, there is an optimal solution containing \( A_1 \):
    * Suppose \( S \subseteq A \) is optimal solution
    * If \( A_1 \in S \), we are done
    * If \( A_1 \notin S \):
      - Let first activity in \( S \) be \( A_k \)
      - Make new solution \( S' = S \setminus \{A_k\} \cup \{A_1\} \) by removing \( A_k \) and using \( A_1 \) instead
      - \( S' \) is valid solution (\( f_1 < f_k \)) of maximal size (\( |S'| = |S| \))
  – \( S \) is an optimal solution for \( A \) containing \( A_1 \) \( \Rightarrow \) \( S' = S \setminus \{A_1\} \) optimal solution for \( A' = \{A_i \in A : s_j \geq f_1\} \) (e.g. after choosing \( A_1 \) the problem reduces to finding optimal solution for activities not overlapping with \( A_1 \))
    * Suppose we have solution \( S'' \) to \( A' \) such that \( |S''| > |S'| = |S| - 1 \)
    * \( S''' = S'' \cup \{A_1\} \) would be solution to \( A \)
    * Contradiction since we would have \( |S'''| > |S| \)
  – Correctness follows by induction on size of \( S \)

• Comparison of greedy algorithm technique with dynamic programming (divide-and-conquer):
  – In greedy algorithm we choose what looks like best solution at any given moment and recurse (choice does not depend on solution to subproblems).
  – In dynamic programming, solution depends on solution to subproblems.
  – Both techniques use optimal solution to subproblems (optimal solution “contains optimal solution for subproblems within it”).

• It is often hard to figure out when being greedy works!

Example:

– 0 – 1 KNAPSACK PROBLEM: Given \( n \) items, with item \( i \) being worth \( \$ v_i \) and having weight \( w_i \) pounds, fill knapsack of capacity \( w \) pounds with maximal value.

– FRACTIONAL KNAPSACK PROBLEM: As 0 – 1 KNAPSACK PROBLEM but we can take fractions of items.

– Problems appear very similar, but only FRACTIONAL KNAPSACK PROBLEM can be solved greedily:
  * Compute value per pound \( \frac{v_i}{w_i} \) for each item
  * Sort items by value per pound.
  * Fill knapsack greedily (take objects in order)
  \( O(n \log n) \) time, easy to show that solution is optimal.
Example that 0–1 knapsack problem cannot be solved greedily:

<table>
<thead>
<tr>
<th>Items in value per pound order</th>
<th>Optimal solution for knapsack of size 50</th>
<th>Greedy solution for knapsack of size 50</th>
</tr>
</thead>
<tbody>
<tr>
<td>$60 10$</td>
<td>$120 30$</td>
<td>$20 10$</td>
</tr>
<tr>
<td>$100 20$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$= $220

$= $160

Note: In fractional knapsack problem we can take $\frac{2}{3}$ of $120 object and get $240 solution.

- 0–1 knapsack problem can be solved in time $O(n \cdot w)$ using dynamic-programming (homework).