What is a Greedy Algorithm?

Solves an optimization problem:
the solution is “best” in some sense.

Greedy Strategy:
- At each decision point, do what looks best “locally”
- Choice does not depend on evaluating potential future choices or solving subproblems
- Top-down algorithmic structure
  • With each step, reduce problem to a smaller problem

Optimal Substructure:
- optimal solution contains in it optimal solutions to subproblems
Greedy Choice Property:
- “locally best” = globally best
Greedy Examples:

- Minimum Spanning Tree
- Minimum Spanning Forest
- Dijkstra Shortest Path
- Huffman Codes
- Fractional Knapsack
- Activity Selection

Greedy Minimum Spanning Tree

**Time:** $O(E \log V)$ given fast FIND-SET UNION

- $A = \emptyset$
- for each vertex $v \in G.V$
- $\text{MAKE-SET}(v)$
- sort the edges of $G.E$ into nondecreasing order by weight $w$
- for each edge $(u, v)$ in $G.E$, taken in nondecreasing order by weight $w$
- if $\text{FIND-SET}(u) \neq \text{FIND-SET}(v)$
- $A \leftarrow A \cup \{(u,v)\}$
- $\text{UNION}(u,v)$
- return $A$

**Invariant:** Minimum weight spanning forest

Greedy Huffman Codes

Single Source Shortest Paths: Dijkstra’s Algorithm

**For (nonnegative) weighted, directed graph $G=(V,E)$**

**DIJKSTRA** ($G, w, s$)

1. INITIALIZE-SINGLE-SOURCE$(G, s)$
2. $S = \emptyset$
3. $Q = G.V$
4. while $Q \neq \emptyset$
5. $u = \text{EXTRACT-MIN}(Q)$
6. $S = S \cup \{u\}$
7. for each vertex $v \in G.Adj[u]$
8. $\text{RELAX}(u, v, w)$

Greedy Huffman Codes

Code table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1111</td>
</tr>
<tr>
<td>2</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>1101</td>
</tr>
<tr>
<td>4</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>1011</td>
</tr>
<tr>
<td>6</td>
<td>1010</td>
</tr>
<tr>
<td>7</td>
<td>1001</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
</tbody>
</table>

source: 9.503 textbook Cormen et al.
Greedy: Example

Activity Selection: Problem Instance

- Set \( S = \{1, 2, \ldots, n\} \) of \( n \) activities
- Each activity \( i \) has:
  - start time: \( s_i \)
  - finish time: \( f_i \) \( s_i \leq f_i \)
- Activities \( i, j \) are compatible iff non-overlapping: \( [s_i, f_i] \cap [s_j, f_j] = \emptyset \).

- Objective:
  - select a \textbf{maximum-sized} set of mutually compatible activities

Greedy: Why does this all work?

Why does it provide an optimal (maximal) solution? It is clear that it will provide a solution (a set of non-overlapping activities), but there is no obvious reason to believe that it will provide a maximal set of activities.

We start with a restatement of the problem in such a way that a \textit{dynamic programming solution} can be constructed. This solution will be shown to be maximal. It will then be \textbf{modified into a greedy solution} in such a way that maximality will be preserved.
Consider the set $S$ of activities, sorted in monotonically increasing order of finishing time:

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Where the activity $a_i$ corresponds to the time interval $[s_i, f_i)$.

- The subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities, but is not maximal.
- The subsets $\{a_1, a_4, a_8, a_{11}\}$ and $\{a_2, a_4, a_9, a_{11}\}$ are also compatible and are both maximal.

**We define the suproblems:** find a maximum size subset of mutually compatible activities from $S_{ij}$, $0 \leq i < j \leq n+1$.

**What is the substructure?**

**Substructure:** suppose a solution to $S_{ij}$ contains some activity $a_k$, $f_i \leq s_k < f_k \leq s_j$. $a_k$ generates two subproblems, $S_{ik}$ and $S_{kj}$.

- The solution to $S_{ij}$ is the union of a solution to $S_{ik}$ (the activities that end before the beginning of $a_k$) and a solution to $S_{kj}$ (the activities that begin after the end of $a_k$).
- Any solution to a larger problem can be obtained by patching together solutions for smaller problems.
- Cardinality (=number of activities) is also additive.

**First Step:** find an optimal substructure: an optimal solution to a subproblem that can be extended to a full solution.

**Define an appropriate space of subproblems:**

$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$, the set of all those activities compatible with $a_i$ and $a_j$ and with those that finish no later than $a_i$ and start no earlier than $a_j$. Add two activities:

$a_0 = [\ldots, 0); a_{n+1} = [\infty, \ldots);$

which come before and after, respectively, all activities in $S = S_{0,n+1}$.

Assume the activities are sorted in non-decreasing order of finish: $f_0 \leq f_1 \leq \ldots \leq f_n < f_{n+1}$.

- Then $S_{ij}$ is empty whenever $i \geq j$.

**Optimal Substructure:** assume $A_{ij}$ is an optimal solution to $S_{ij}$ containing an activity $a_k$.

- Then $A_{ij}$ contains a solution to $S_{ik}$ (the activities that end before the beginning of $a_k$) and a solution to $S_{kj}$ (the activities that begin after the end of $a_k$).

- If these solutions were not already maximal, then one could obtain a maximal solution for, say, $S_{ik} \cup \{a_k\}$ and the solution to $S_{kj}$ to obtain a solution for $S_{ij}$ of greater cardinality, thus violating the optimality condition assumed for $A_{ij}$. 


Greedy

- **Use Optimal Substructure** to construct an optimal solution:
  Any solution to a nonempty problem $S_{ij}$ includes some activity $a_k$, and any optimal solution to $S_{ij}$ must contain optimal solutions to $S_{ik}$ and $S_{kj}$.
  For each $a_k$ in $S_{0,n+1}$, find (recursively) optimal solutions of $S_{0,k}$ and $S_{k,n+1}$, say $A_{0,k}$ and $A_{k,n+1}$. Splice them together, along with $a_k$, to form solutions $A_k$. Take a maximal $A_k$ as an optimal solution $A_{0,n+1}$.

We need to look at the recursion in more detail...

**Greedy**

**Second Step**: the recursive algorithm.
Let $c[i,j]$ denote the maximum number of compatible activities in $S_{ij}$. It is easy to see that $c[i,j] = 0$, whenever $i ≥ j$ - and $S_{ij}$ is empty.
If, in a non-empty set of activities $S_{ij}$, the activity $a_k$ occurs as part of a maximal compatible subset, this generates two subproblems, $S_{ik}$ and $S_{kj}$, and the equation
$c[i,j] = c[i,k] + 1 + c[k,j]$,
which tells us how the cardinalities are related. The problem is that $k$ is not known a priori. Solution:
$c[i,j] = \max_{i < k < j} (c[i,k] + 1 + c[k,j])$, if $S_{ij} ≠ \emptyset$
And one can write an easy “bottom up” (dynamic programming) algorithm to compute a maximal solution.

**Greedy**

**A better mousetrap**. Recall $S_{ij} = \{ a_k \in S : f_i ≤ s_k < f_k ≤ s_j \}$.

**Theorem 16.1**. Consider any nonempty subproblem $S_{ij}$ and let $a_m$ be the activity in $S_{ij}$ with the earliest finish time: $f_m = \min(f_k : a_k \in S_{ij})$. Then:
1. $a_m$ is used in some maximum size subset of mutually compatible activities of $S_{ij}$ (e.g., $A_{ij} = A_{im} ∪ \{ a_m \} ∪ A_{mj}$).
2. The subproblem $S_{im}$ is empty, so that choosing $a_m$ leaves the subproblem $S_{mj}$ as the only one that may be nonempty.

**Proof**: 2) if $S_{im}$ is nonempty then $S_{ij}$ must contain an activity (finishing) prior to $a_m$. Contradiction.
1) Suppose $A_{ij}$ is a maximal subset. Either $a_m$ is in $A_{ij}$ and we are done, or it is not. If not, let $a_i$ be the earliest finishing activity in $A_{ij}$. It can be replaced by $a_m$ (since it finishes no earlier than $a_m$) thus giving a maximal subset containing $a_m$.

**Greedy**

**Why is this a better mousetrap?**

- The dynamic programming solution requires solving $j - i - 1$ subproblems to solve $S_{ij}$. Total Running Time???
- **Theorem 16.1** gives us conditions under which solving $S_{ij}$ requires solving ONE subproblem only, since the other subproblems implied by the dynamic programming recurrence relation are empty, or irrelevant (they might give us other maximal solutions, but we need just one)...
  Lots less computation...

Another benefit comes from the observation that the problem can be solved in a “top-down” fashion: take the earliest finishing activity, $a_m$, and you are left with the problem $S_{m,n+1}$. It is easy to see that each activity needs to be looked at only once: linear time (after sorting).
Greedy

The Algorithm:
R_A_S(s, f, k, n)
1. \( m = k + 1 \)
2. while \( m \leq n \) and \( s[m] < f[k] \) // find first activity in \( S_k \) to finish
3. \( m = m + 1 \)
4. if \( m \leq n \)
5. then return \( \{a_m\} + R_A_S(s, f, m, n) \)
6. else return \( \emptyset \).

The time is - fairly obviously - \( \Theta(n) \).

Greedy

A Slightly Different Perspective.
Rather than start from a dynamic programming approach, moving to a greedy strategy, start with identifying the characteristics of the greedy approach.

1. Make a choice and leave a subproblem to solve.
2. Prove that the greedy choice is always safe - there is an optimal solution starting from a greedy choice.
3. Prove that, having made a greedy choice, the solution of the remaining subproblem can be added to the greedy choice providing a solution to the original problem.

Greedy

Greedy Choice Property.
Choice that looks best in the current problem will work: no need to look ahead to its implications for the solution.

This is important because it reduces the amount of computational complexity we have to deal with.

Can’t expect this to hold all the time: maximization of a function with multiple relative maxima on an interval - the gradient method would lead to a "greedy choice", but may well lead to a relative maximum that is far from the actual maximum.
Greedy

Optimal Substructure.
While the greedy choice property tells us that we should be able to solve the problem with little computation, we need to know that the solution can be properly reconstructed: \textit{an optimal solution contains optimal “sub-solutions” to the subproblems.}

An induction can then be used to turn the construction around (bottom-up).

In the case of a problem possessing only the Optimal Substructure property there is little chance that we will be able to find methods more efficient than dynamic programming (with memoization).

Note:
- If the optimization problem does not have the “greedy choice property”, a greedy algorithm may still be useful in bounding the optimal solution
- Example: minimization problem

Solution Values
- Upper Bound
- Optimal (unknown value)
- Lower Bound