Dynamic programming and greedy algorithms

Pierre Aboulker - pierreaboulker@gmail.com
1 - Introduction
Basic Algorithmic strategies

- Divide and Conquer
- Dynamic Programming
- Greedy

Common Theme:
- To solve a large, complicated problem, break it into many smaller subproblems.
- Solve the subproblems.
- Recover a solution of the large problems from the solutions of the small subproblems.
Basic Algorithmic strategies

- Divide and Conquer
- Dynamic Programming
- Greedy

**Common Theme:**

- To solve a large, complicated problem, break it into many smaller sub-problems.
- Solve the subproblems.
- Recover a solution of the large problems from the solutions of the small subproblems.
2 - Dynamic Programming
Divide and conquer VS Dynamic Programming

- **Divide and Conquer** divide the problem into disjoint (unrelated) subproblems, solve them and then merge the solutions (merging is usually the difficult part).

- In contrast, **Dynamic Programming** applies when subproblems overlap, that is, when subproblems share subsubproblems.

  - In this context, a divide and conquer algorithm would solve many subsubproblems many times, big lost of times.

  - Dynamic Programming solve each subsubproblem once, and save the answer in a table. Like that we have access to the solution of the subproblems in constant time when needed.

- **Space Complexity** is of importance for DP.
History of Dynamical Programming

- Bellman pioneered the systematic study of dynamic programming in the 1950s.
- The Secretary of Defense at that time 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" (Bellman, R. E., Eye of the Hurricane, An Autobiography).
Application of Dynamical Programming

- Computational biology: Smith-Waterman algorithm for sequence alignment.
- Operations research: Bellman-Ford algorithm for shortest path routing in networks...
- Combinatorial Optimization
- Graph algorithms: FPT algorithm parametrized by treewidth.
- Algorithm on strings: minimal edit distance
Idea:
Break the problem into many closely related sub-problems, memorize the result of the sub-problems to avoid repeated computation.
Fibonacci numbers
Warm up example: Fibonacci numbers

Fibonacci numbers is defined by the following recurrence relation

\[ F_0 = 0, \quad F_1 = 1, \quad \text{and for } n \geq 2: \quad F_n = F_{n-1} + F_{n-2} \]

First Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144…

**Problem** (Fibonacci Number)

**Input** : An integer \( n \)

**Output** : The value of \( F_n \).
First idea: recursive algorithm

**First idea**: straightforward recursive algorithm from the definition of Fibonacci numbers:

**Algorithm 1** Fibo: recursive algorithm

1. Fibo(n):
2.   if $n \leq 1$ then
3.     return $n$
4.   return Fibo(n-1) + Fibo(n-2)

Solve the recurrence: $T(n) = T(n - 1) + T(n - 2)$

**Complexity**: EXPONENTIAL :(  

**Question**: Why is it so bad??
Why is it so bad??

**Answer:** Because we compute many time the same values
Memoization!

Instead of computing many times the same value, we compute it once and store it.

Your first algo using dynamic programming:

\textbf{Algorithm 2} Fibo: Dynamic Programming

1: Fibo(n):
2: \texttt{Tab} ← zeros(n) \hspace{1cm} \triangleright \text{Array to stock the values of } F_i
3: \texttt{Tab}[0] ← 0
4: \texttt{Tab}[1] ← 1
5: \texttt{for } i ← 2 \text{ to } n \text{ do}
6: \hspace{1cm} \texttt{Tab}[i] = \texttt{Tab}[i-1] + \texttt{Tab}[i-2]
7: \texttt{return } \texttt{Tab}[n]

- \textbf{Time complexity: } \textit{O}(n)
- \textbf{Space complexity: } \textit{O}(n)
Basic Steps in Designing Dynamic Programming Algorithms

- Relate the problem recursively to smaller sub-problems. (Transition function, $f(n) = f(n-1)+f(n-2)$)
- Organize all sub-problems as a dynamic programming table. (A table for all values of $n$)
- Fill in values in the table in an appropriate order ($n = 0, 1, 2, 3 \ldots$)
The Minimum Edit Distance Problem
How similar are two strings?

In many applications one want to know how similar two strings are:

- Spell correction: the user types something like \textit{struing}, what existing word is close of it?
- Biology: to quantify the similarity of DNA sequences, which can be viewed as strings of the letters A, C, G and T.
- Also for Machine Translation, Information Extraction, Speech Recognition...
Minimum edit distance problem

- Different definitions of an edit distance use different sets of string operations.
- The Levenshtein distance operations are:
  ▶ Replacement: replace a letter by another one
  ▶ Delete: delete a letter
  ▶ Insert: insert a letter

**Problem** (Minimum Edit Distance)

**Input**: Two strings $S_1$ and $S_2$

**Output**: Minimum number of basic operations to transform $S_1$ into $S_2$

Example: $S_1 = \textit{vintners}$ (un vigneron) and $S_2 = \textit{winters}$
**Edit transcript**

**Definition**

A string over the alphabet $I$, $D$, $R$, $M$ that describes the transformation $S_1$ into $S_2$ is called an **edit transcript** of $S_1$ into $S_2$.

**Example**: Edit transcript of $S_1$ into $S_2$: $RIMDMDMI$.

- To apply the transcript we need two pointers $next_1$ and $next_2$ into $S_1$ and $S_2$.
- When symbol $I$ is met:
  - character $next_2$ is inserted right before character $next_1$ and
  - $next_2 = next_2 + 1$
- When symbol $D$ is met:
  - character $next_1$ is deleted and
  - $next_1 = next_1 + 1$
- When symbol $R$ or $M$ is met, I let you guess.
Relate the original problem to smaller problems

**Problem (Minimum Edit Distance)**

**Input**: Two strings $S_1$ and $S_2$ with lengths $n$ and $m$

**Output**: Minimum number of basic operation to transform $S_1$ into $S_2$

Denote by $D[i, j]$ the distance between $S_1[1, i]$ and $S_2[1, j]$.

**Goal**: Compute $D[n, m]$

**Strategy**: • Compute all values $D[i, j]$ for $i = 0, \ldots, n$ and $j = 0, \ldots, m$
  • Each value takes constant time to compute thanks to memoization.

**Tasks**: Compute $D(i, j)$ knowing $D(u, v)$ for all $(u, v)$ ”smaller than” $(i, j)$, which means two tasks:
  • find (and prove) a recurrence relation.
  • find an efficient way to compute it: tabular computation
The recurrence relation

- **Base condition:** \( D(i, 0) = i \) and \( D(j, 0) = j \).

- **Recurrence relation:** for \( i, j > 0 \):

\[
D(i, j) = \min \left\{ \begin{array}{ll}
D(i - 1, j) + 1 \\
D(i, j - 1) + 1 \\
D(i - 1, j - 1) + t(i, j)
\end{array} \right. \\
\text{where } t(i, j) = \begin{cases} 
1 & \text{if } S_1(i) \neq S_2(j) \\
0 & \text{if } S_1(i) = S_2(j) 
\end{cases}
\]

Let us prove rigorously that this relation is correct.
Proof of correctness I

Lemma 1: \( D(i, j) \in \{ D(i - 1, j) + 1, \ D(i, j - 1) + 1, \ D(i - 1, j - 1) + t(i, j) \} \)

PROOF Consider an edit transcript for the transformation of \( S_1[1..i] \) to \( S_2[1..j] \) using the minimum number of edit operations, and focus on the last symbol in that transcript. That last symbol must either be I, D, R, or M. If the last symbol is an I then the last edit operation is the insertion of character \( S_2(j) \) onto the end of the (transformed) first string. It follows that the symbols in the transcript before that I must specify the minimum number of edit operations to transform \( S_1[1..i] \) to \( S_2[1..j - 1] \) (if they didn't, then the specified transformation of \( S_1[1..i] \) to \( S_2[1..j] \) would use more than the minimum number of operations). By definition, that latter transformation takes \( D(i, j - 1) \) edit operations. Hence if the last symbol in the transcript is I, then \( D(i, j) = D(i, j - 1) + 1 \).

Do the same when the last symbol is \( D, R \) and \( M \).
Proof of correctness II

Lemma 2: \( D(i, j) \leq \min\{D(i - 1, j) + 1, \ D(i, j - 1) + 1, \ D(i - 1, j - 1) + t(i, j)\} \)

**Proof** The reasoning is very similar to that used in the previous lemma, but it achieves a somewhat different goal. The objective here is to demonstrate constructively the existence of transformations achieving each of the three values specified in the inequality. Then since all three values are feasible, their minimum is certainly feasible.

First, it is possible to transform \( S_1[1..i] \) into \( S_2[1..j] \) with exactly \( D(i, j - 1) + 1 \) edit operations. Simply transform \( S_1[1..i] \) to \( S_2[1..j - 1] \) with the minimum number of edit operations, and then use one more to insert character \( S_2(j) \) at the end. By definition, the number of edit operations in that particular way to transform \( S_1 \) to \( S_2 \) is exactly \( D(i, j - 1) + 1 \). Second, it is possible to transform \( S_1[1..i] \) to \( S_2[1..j] \) with exactly \( D(i - 1, j) + 1 \) edit operations. Transform \( S_1[1..i - 1] \) to \( S_2[1..j] \) with the fewest operations, and then delete character \( S_1(i) \). The number of edit operations in that particular transformation is exactly \( D(i - 1, j) + 1 \). Third, it is possible to do the transformation with exactly \( D(i - 1, j - 1) + t(i, j) \) edit operations, using the same argument.  \(\square\)
Proof of correctness III

**Theorem:** \( D(i,j) = \min\{D(i-1,j) + 1, \ D(i,j-1) + 1, \ D(i-1,j-1) + t(i,j)\} \)

**Proof:** By Lemma 1 and 2.

**Next Step:** Find an efficient way to compute all the \( D(i,j) \)
Not using DP

\[ D(i, 0) = i, \ D(j, 0) = j \text{ and for } i, j > 0: \]

\[ D(i, j) = \min \begin{cases} 
D(i - 1, j) + 1 \\
D(i, j - 1) + 1 \\
D(i - 1, j - 1) + t(i, j) \end{cases} \]

where \( t(i, j) = \begin{cases} 
1 & \text{if } S_1[i] \neq S_2[j] \\
0 & \text{if } S_1[i] = S_2[j] \end{cases} \)

**First idea:** straightforward recursive algorithm from the recurrence relation:

**Algorithm 3** Edit distance: recursive algorithm

1. \textbf{Edit}(S_1[1, n], S_2[1, m]):
2. \textbf{if} \( n = 0 \) \textbf{then}
3. \hspace{1em} \textbf{return} \( m \)
4. \textbf{if} \( m = 0 \) \textbf{then}
5. \hspace{1em} \textbf{return} \( n \)
6. \textbf{return} \( \min\{D(i - 1, j) + 1, \ D(i, j - 1) + 1, \ D(i - 1, j - 1) + t(i, j)\} \)

Simple to program but extremely inefficient (exponential time).
Memoization I

- Create a $n \times m$ array.
- Case $(i, j)$ will contain the value of $D(i, j)$.
- The array with the base cases filled:

```
<table>
<thead>
<tr>
<th>D(i, j)</th>
<th>w</th>
<th>r</th>
<th>i</th>
<th>t</th>
<th>e</th>
<th>r</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>v</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 11.1: Table to be used to compute the edit distance between *vintner* and *writers*. The values in row zero and column zero are already included. They are given directly by the base conditions.
Memoization II

- Compute $D(i, j)$ for increasing values of $i$ and $j$ (find a smart way to do it)
- Here: fill the array line by line:

```
<table>
<thead>
<tr>
<th>D(i, j)</th>
<th>w</th>
<th>r</th>
<th>i</th>
<th>t</th>
<th>e</th>
<th>r</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>v</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>i</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>n</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>t</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>n</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>e</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- This is called **Bottom-up** computation.
Time analysis

Computing the value of a specific cell is done in constant time.

- **Time Complexity:** $O(nm)$
- **Space Complexity:** $O(nm)$ (size of the array)
The traceback

- The algorithm computes the distance between two strings, but how can we **extract the associated optimal transcript?**
- Solution: establish pointers in the table as the table values are computed.
- Where to point depends on the chosen value of

$$\min\{D(i-1,j) + 1, \ D(i,j-1) + 1, \ D(i-1,j-1) + t(i,j)\}$$

<table>
<thead>
<tr>
<th>$D(i,j)$</th>
<th>(w)</th>
<th>(r)</th>
<th>(i)</th>
<th>(t)</th>
<th>(e)</th>
<th>(r)</th>
<th>(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>←1</td>
<td>←2</td>
<td>←3</td>
<td>←4</td>
<td>←5</td>
<td>←6</td>
</tr>
<tr>
<td>(v)</td>
<td>↑1</td>
<td>↓1</td>
<td>↓2</td>
<td>↓3</td>
<td>↓4</td>
<td>↓5</td>
<td>↓6</td>
</tr>
<tr>
<td>(i)</td>
<td>↑2</td>
<td>↑2</td>
<td>↓2</td>
<td>↓2</td>
<td>↓3</td>
<td>↓4</td>
<td>↓5</td>
</tr>
<tr>
<td>(n)</td>
<td>↑3</td>
<td>↑3</td>
<td>↑3</td>
<td>↑3</td>
<td>↑3</td>
<td>↑3</td>
<td>↑4</td>
</tr>
<tr>
<td>(t)</td>
<td>↑4</td>
<td>↑4</td>
<td>↓4</td>
<td>↑4</td>
<td>↑4</td>
<td>↑3</td>
<td>↓4</td>
</tr>
<tr>
<td>(n)</td>
<td>↑5</td>
<td>↑5</td>
<td>↑5</td>
<td>↑5</td>
<td>↑5</td>
<td>↑4</td>
<td>↓4</td>
</tr>
<tr>
<td>(e)</td>
<td>↑6</td>
<td>↑6</td>
<td>↑6</td>
<td>↑6</td>
<td>↑6</td>
<td>↑5</td>
<td>↓4</td>
</tr>
<tr>
<td>(r)</td>
<td>↑7</td>
<td>↑7</td>
<td>↖6</td>
<td>↖7</td>
<td>↑6</td>
<td>↑5</td>
<td>↖4</td>
</tr>
</tbody>
</table>
Dynamic Programming

- More powerful than greedy and divide-and-conquer strategies.

- Implicitly explore space of all possible solutions.

- Solve multiple sub-problems and build up correct solutions to larger and larger sub-problems.

- Careful analysis needed to ensure number of sub-problems solved is polynomial in the size of the input.
The Knapsack Problem
Knapsack Problem

**INPUT:**
- A knapsack that can hold items of total weight at most $W$.
- $n$ items with weights $w_1, w_2, \ldots, w_n$.
- Each item also has a value $v_1, v_2, \ldots, v_n$.

**Goal:** Select some items to put into the knapsack such that:
1. Total weight is at most $W$.
2. Total value is as large as possible.
**Example for the Knapsack Problem**

Capacity of the knapsack: \( W = 10 \)

Three items:

- \( w_1 = 4 \quad v_1 = 2 \)
- \( w_2 = 5 \quad v_1 = 3 \)
- \( w_3 = 7 \quad v_1 = 4 \)

Solution: Put items 1 and 2 in the knapsack:
- Total weight is 9 ≤ 10
- Total value is 3 + 2 = 5
Example for the Knapsack Problem

Capacity of the knapsack: $W = 10$

Three items:
- $w_1 = 4$ \quad $v_1 = 2$
- $w_2 = 5$ \quad $v_1 = 3$
- $w_3 = 7$ \quad $v_1 = 4$

**Solution**: Put items 1 and 2 in the knapsack:
- Total weight is $9 \leq 10$
- Total value is $3 + 2 = 5$
Formal description

INPUT:
Two vectors $w = (w_1, \ldots, w_n)$ (weight vector), $v = (v_1, \ldots, v_n)$ (value vector), and an integer $W > 0$ (capacity)

GOAL: Find $x = (x_1, \ldots, x_n) \in \{0, 1\}^n$ (choose some of the $n$ items)

- subject to $\sum_{i=1}^{n} w_i \cdot x_i \leq W$
- maximizes $\sum_{i=1}^{n} v_i$

It is an Optimization Problem

**BRUTE FORCE**: $2^n$

Can we do better?
Divide and conquer?

Recall divide and conquer strategy:

1. Partition the problem into subproblems
2. Solve the subproblems
3. Combine the solutions to solve the original one.

**Remark**: If the subproblems are not independent, i.e. subproblems share subsubproblems, then a divide-and-conquer algorithm repeatedly solves the common subsubproblems. Thus, it does more work than necessary!

**Question**: Any better solution?
Divide and conquer?

Recall divide and conquer strategy:

1. Partition the problem into subproblems
2. Solve the subproblems
3. Combine the solutions to solve the original one.

**Remark**: If the subproblems are not independent, i.e. subproblems share subsubproblems, then a divide-and-conquer algorithm repeatedly solves the common subsubproblems. Thus, it does more work than necessary!

**Question**: Any better solution?

**Yes**: Dynamic programming (DP)!
Designing a DP algorithm for Knapsack

1. Think of the problem as making a sequence of decisions:
   - For each item, decide whether we put it into the knapsack or not.

2. Focus on last decision, enumerate the options
   - For the last item, we either put it in, or leave it out.

3. Try to relate each option to a smaller subproblem
   - Subproblems:
     - Fill the remaining capacity using the remaining items.
     - leave it out: number of items is now smaller.
     - put it in: both capacity and number of items are smaller.
Relate the original problem to smaller subproblem

You start with item 1, and have two choices:

- Put item 1 in the knapsack and then solve a new knapsack problem with parameters:
  - \((v_2, \ldots, v_n)\)
  - \((w_2, \ldots, w_n)\)
  - \(W := W - w_2\)

- Don’t put item 1 in the knapsack and then solve a new knapsack problem with parameters:
  - \((v_2, \ldots, v_n)\)
  - \((w_2, \ldots, w_n)\)
  - \(W\)
Formalization

Definition of the subproblems:
Let $KP(i, T)$ be the optimal solution when the set of available items is the first $i$ items and the maximum total weight is at most $T$.

Base condition: $KP(0, T) = 0$ and $KP(i, 0) = 0$.

Recurrence relation for $i, W > 0$:

$$KP(i, W) = \min \left\{ \begin{array}{ll}
KP(i - 1, W) & \text{item } i \text{ is not chosen} \\
KP(i - 1, W - w_i) + v_i & \text{item } i \text{ is chosen (need } W \geq w_i) \end{array} \right. $$
Pseudocode

Algorithm 4 Knapsack: Dynamic Programming

1: for \( i \leftarrow 0 \) to \( n \) do
2: \( \text{KP}[i,0]=0 \)
3: for \( i \leftarrow 0 \) to \( W \) do
4: \( \text{KP}[0,W]=0 \)
5: for \( i \leftarrow 1 \) to \( n \) do
6: \( \text{for } j \leftarrow 1 \) to \( W \) do
7: \( \text{if } W < w_i \text{ then} \)
8: \( \text{KP}[i, W] \leftarrow \text{KP}[i - 1, W] \)
9: \( \text{else} \)
10: \( \text{KP}[i, W] = \min \left\{ \text{KP}[i - 1, W], \text{KP}[i - 1, W - w_i] + v_i \right\} \)

return \( \text{KP}[n, W] \)

- \( KP \) is an \( n \cdot W \) array
- We first fill the first line and the first column (base condition)
- Then we fill it lines by lines.
We need to do:

- Time analysis.
- Proof of correctness.
Time and complexity analysis

The computation of each cell of the array KP take constant time, so

- **Time Complexity:** $O(nW)$
- **Space Complexity:** $O(nW)$ (size of the array)

Is this a polynomial-time algorithm?
**Time and complexity analysis**

The computation of each cell of the array KP take constant time, so

- **Time Complexity:** $O(nW)$
- **Space Complexity:** $O(nW)$ (size of the array)

Is this a polynomial-time algorithm?

- $W$ is a number, so it is coded in $\log(W)$ bites.
- Hence the size of the input $n \cdot \log(W)$ and thus our algorithm in exponential.
- We say it is **pseudo-polynomial**
- Knapsack problem is known to be **NP-Hard**
Proof of correctness

- It is enough to prove that recurrence relation is correct.
- We prove correctness of a DP algorithm using induction in the same order as you compute the states.
Proof of correctness

- It is enough to prove that recurrence relation is correct.
- We prove correctness of a DP algorithm using induction in the same order as you compute the states.

- **Induction Hypothesis** we compute $KP[u, v]$ correctly for all pairs of $(u, v) < (i, j)$ (meaning $(u, v)$ is computed before $(i, j)$)
- **Base cases**: the base cases for our algorithm are $a[i, 0] = a[0, j] = 0$. They are trivially correct.
- Assume our Inductive Hypothesis is true, then when computing $KP[i, j]$, we either put item $i$ in the knapsack or leave it out.
- Since we are taking the maximum over the two values, and both $KP[i - 1, j]$ and $KP[i - 1, j - w_i]$ are computed correctly based on our Inductive Hypothesis,
- Our algorithm computes the correct value of $KP[i, j]$. This finishes the induction.
3 - Greedy Algorithm
Greedy Technique

• As with dynamic programming, in order to be solved with the greedy technique, the problem must have the optimal substructure property.

• The problems that can be solved with the greedy method are a subset of those that can be solved with dynamic programming.

• The idea of greedy technique is the following: at every step you have a choice. Instead of evaluating all choices recursively and picking the best one, pick what looks like locally the best choice, and go with that.

• So basically a greedy algorithm picks the locally optimal choice hoping to get the globally optimal solution.

• Coming up with greedy heuristics is easy, but proving that a heuristic gives the optimal solution is tricky (usually).
The Fractional Knapsack Problem

The difference is that now the items are infinitely divisible: can put $\frac{1}{2}$ (or any fraction) of an item into the knapsack.
Formal description

**INPUT:**
Two vectors \( w = (w_1, \ldots, w_n) \) (weight vector), \( v = (v_1, \ldots, v_n) \) (value vector), and an integer \( W > 0 \) (capacity)

**GOAL:** Find \( x = (p_1, \ldots, p_n) \in [0, 1]^n \) (choose some fractions of the \( n \) items)
- subject to \( \sum_{i=1}^{n} w_i \cdot p_i \leq W \)
- maximizes \( \sum_{i=1}^{n} v_i \)
Greedy criterion

We want to make a sequence of decisions on what to put in the knapsack. If you are indeed in the situation, with a knapsack and some items etc, what would be your strategy?
Greedy criterion

We want to make a sequence of decision on what to put in the knapsack. If you are indeed in the situation, with a knapsack and some items etc, what would be your strategy?

**Algorithm:** Iteratively picks the item with the greatest value-per-weight ratio (maximum $\frac{v_i}{w_i}$)

If, at the end, the knapsack cannot fit the entire last item with greatest value-per-weight ratio among the remaining items, we will take a fraction of it to fill the knapsack.
This algorithm takes $O(n \log n)$ time to sort the items by the ratio in decreasing order, and another $O(n)$ time to traverse and pick from the list of items until the knapsack is full.

Hence the total running time is $O(n \log n)$
Correctness Proof

Proof Sketch:

- Assume wlog that $\frac{v_1}{w_1} < \frac{v_2}{w_2} < \ldots, \frac{v_n}{w_n}$
Correctness Proof

Proof Sketch:

1. Assume wlog that \( \frac{v_1}{w_1} < \frac{v_2}{w_2} < \ldots < \frac{v_n}{w_n} \).
2. Let \( ALG = (p_1, \ldots, p_n) \) be the solution output by the algorithm.
Correctness Proof

Proof Sketch:

- Assume wlog that $\frac{v_1}{w_1} < \frac{v_2}{w_2} < \ldots, \frac{v_n}{w_n}$
- Let $ALG = (p_1, \ldots, p_n)$ be the solution output by the algorithm.
- Let $OPT$ be an optimal solution.
Correctness Proof

Proof Sketch:

- Assume wlog that $\frac{v_1}{w_1} < \frac{v_2}{w_2} < \ldots, \frac{v_n}{w_n}$
- Let $ALG = (p_1, \ldots, p_n)$ be the solution output by the algorithm.
- Let $OPT$ be an optimal solution.
- Assume for contradiction that $ALG \neq OPT$.

Let $i$ be the smallest subscript such that $p_i \neq q_i$.

Prove that $p_i > q_i$.

So there exists $j > i$ such that $p_j < q_j$.

Set $q' = (q_1, \ldots, q_{i-1}, q_i - 1, q_i + \epsilon, q_i + 1, \ldots, q_j + \epsilon, \ldots, q_n)$.

Show that $q'$ is a feasible solution, that is $\sum_{i=1}^{n} q'_i \leq W$.

Show that in contradicts the optimality of $OPT$, that is $\sum_{i=1}^{n} q'_i v_i > \sum_{i=1}^{n} q_i v_i$.
Correctness Proof

Proof Sketch:

1. Assume wlog that $\frac{v_1}{w_1} < \frac{v_2}{w_2} < \ldots, \frac{v_n}{w_n}$
2. Let $ALG = (p_1, \ldots, p_n)$ be the solution output by the algorithm.
3. Let $OPT$ be an optimal solution.
4. Assume for contradiction that $ALG \neq OPT$.
5. Let $i$ be the smallest subscript such that $p_i \neq q_i$.

Let $q'$ be the solution obtained by swapping $q_i$ with $q'_i$, where $q'_i = q_i - 1 + \epsilon$. Show that $q'$ is a feasible solution and in contradiction to the optimality of $OPT$. Thus, $ALG = OPT$. 

Pierre Aboulker - pierreaboulker@gmail.com
Dynamic programming and greedy algorithms
Correctness Proof

Proof Sketch:

- Assume wlog that \( \frac{v_1}{w_1} < \frac{v_2}{w_2} < \ldots, \frac{v_n}{w_n} \)
- Let \( ALG = (p_1, \ldots, p_n) \) be the solution output by the algorithm.
- Let \( OPT \) be an optimal solution.
- Assume for contradiction that \( ALG \neq OPT \).
- Let \( i \) be the smallest subscript such that \( p_i \neq q_i \).
- Prove that \( p_i > q_i \).
Correctness Proof

Proof Sketch:

- Assume wlog that $\frac{v_1}{w_1} < \frac{v_2}{w_2} < \ldots < \frac{v_n}{w_n}$
- Let $ALG = (p_1, \ldots, p_n)$ be the solution output by the algorithm.
- Let $OPT$ be an optimal solution.
- Assume for contradiction that $ALG \neq OPT$.
- Let $i$ be the smallest subscript such that $p_i \neq q_i$.
- Prove that $p_i > q_i$.
- So there exists $j > i$ such that $p_j < q_j$. 
Correctness Proof

Proof Sketch:

- Assume wlog that \( \frac{v_1}{w_1} < \frac{v_2}{w_2} < \ldots, \frac{v_n}{w_n} \).
- Let \( ALG = (p_1, \ldots, p_n) \) be the solution output by the algorithm.
- Let \( OPT \) be an optimal solution.
- Assume for contradiction that \( ALG \neq OPT \).
- Let \( i \) be the smallest subscript such that \( p_i \neq q_i \).
- Prove that \( p_i > q_i \).
- So there exists \( j > i \) such that \( p_j < q_j \).
- Set \( q' = (q_1, \ldots, q_{i-1}, q_i + \epsilon, q_{i+1}, \ldots, q_j + \frac{w_i}{w_j}, \ldots, q_n) \).
Correctness Proof

Proof Sketch:

- Assume wlog that $\frac{v_1}{w_1} < \frac{v_2}{w_2} < \ldots, \frac{v_n}{w_n}$
- Let $ALG = (p_1, \ldots, p_n)$ be the solution output by the algorithm.
- Let $OPT$ be an optimal solution.
- Assume for contradiction that $ALG \neq OPT$.
- Let $i$ be the smallest subscript such that $p_i \neq q_i$.
- Prove that $p_i > q_i$.
- So there exists $j > i$ such that $p_j < q_j$.
- Set $q' = (q_1, \ldots, q_{i-1}, q_i + \epsilon, q_{i+1}, \ldots, q_j + \epsilon \frac{w_i}{w_j}, \ldots, q_n)$.
- Show that $q'$ is a feasible solution, that is
  $$\sum_{i=1}^{n} q'_i w_i \leq W$$
Correctness Proof

Proof Sketch:

- Assume wlog that \( \frac{v_1}{w_1} < \frac{v_2}{w_2} < \ldots, \frac{v_n}{w_n} \).
- Let \( ALG = (p_1, \ldots, p_n) \) be the solution output by the algorithm.
- Let \( OPT \) be an optimal solution.
- Assume for contradiction that \( ALG \neq OPT \).
- Let \( i \) be the smallest subscript such that \( p_i \neq q_i \).
- Prove that \( p_i > q_i \).
- So there exists \( j > i \) such that \( p_j < q_j \).
- Set \( q' = (q_1, \ldots, q_{i-1}, q_i + \epsilon, q_{i+1}, \ldots, q_j + \epsilon \frac{w_i}{w_j}, \ldots, q_n) \). 
- Show that \( q' \) is a feasible solution, that is
  \[
  \sum_{i=1}^{n} q'_i w_i \leq W
  \]
- Show that in contradicts the optimality of \( OPT \), that is
  \[
  \sum_{i=1}^{n} q'_i v_i > \sum_{i=1}^{n} q_i v_i
  \]
Given two strings $s_i$ and $s_j$, we say that $s_i$ is a superstring of $s_j$ if $s_i$ contains consecutive letters that match $s_j$ exactly.

**Problem (Shortest Superstring Problem (SSP))**

**Input**: Strings $s_1, \ldots, s_n$

**Output**: The shortest string $s$ that contains $s_1, \ldots, s_n$ as substrings.
The natural Greedy Heuristic

A very well known, simply implemented, and widely used algorithm for the SSP is the natural greedy algorithm. It is routinely used in DNA sequencing practice. It starts with the string set $S$ and repeatedly merges a pair of distinct strings with the maximum possible overlap until only one string remains in $S$. Next algorithm shows the pseudo-code of the natural greedy for the SSP.

**Algorithm: GREEDY**

**input**: string set $S = \{s_1, s_2, \ldots, s_n\}$

**output**: a superstring of $S$

1. **for** $i = 1$ **to** $n - 1$ **do**
2. \[ L = \{ (s_i, s_j) : s_i, s_j \in S, i \neq j \} \]
3. \[ k = \max\{|o(s_i, s_j)| : (s_i, s_j) \in L\} \]
4. let $(s'_i, s'_j) \in L$ be a pairs such that \(|o(s'_i, s'_j)| = k\)
5. \[ S = (S - \{s'_i, s'_j\}) \cup \{m(s'_i, s'_j)\} \]
6. **end**
7. let $s$ be the only string in $S$
8. **return** $s$
The following Conjecture has been proposed by Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis in 1991:

**Conjecture (The Greedy Conjecture)**

Let $ALG$ be a solution given by the Greedy Algorithm and let $OPT$ be an optimal solution.

Is it true that $ALG \leq 2 \cdot OPT$.

**Easier question:** can you prove that the factor 2 cannot be decreased?