Dynamic Programming
Dynamic Programming

A way of speeding up certain recursive algorithms

Idea: don’t compute the same thing twice!

Example: Subset Sum

Given: positive integers $a_1, \ldots, a_n$, and a “target” value $t$

Question: is there a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = t$?
We define a Boolean function

\[ P(\langle a_1, \ldots, a_n \rangle, t) \]

Recursive evaluation (intuition): either we include the last index in the solution (and the target shifts) or we don’t

\[ P(\langle a_1, \ldots, a_n \rangle, t): \]

- if \( n = 0 \) then
  - return \( (t = 0) \)
- else
  - \( // \) assume short circuit Boolean evaluation
  - return ( \( a_n \leq t \) and \( P(\langle a_1, \ldots, a_{n-1} \rangle, t - a_n) \) ) or \( P(\langle a_1, \ldots, a_{n-1} \rangle, t) \)
Running time is exponential in $n$

Recursion Tree:

```
                     n
                    /\         /\         /\         /\
                   n-1   n-1   n-1   n-1
                  /\     /\     /\     /\    height = n
                 n-2   n-2   n-2   n-2    size of tree = $2^n$
```

$height = n$
$size of tree = 2^n$
Observe: the number of *unique* problem instances is at most 
\((n + 1)(t + 1)\), which is much smaller (assuming \(t\) is not 
too large)

Idea — “memoization”: maintain a table of results \(T[i, s]\), 
where \(i = 0 \ldots n\) and \(s = 0 \ldots t\)

Initially, \(T[i, s] = \bot\) for all \(i, s\)

As recursion proceeds, \(T[i, s]\) is set to \(P(\langle a_1, \ldots, a_i \rangle, s)\)

If a sub-problem has already been solved, fetch value from 
table, and skip recursion

Total time = \(O(nt)\)
\[ P(\langle a_1, \ldots, a_n \rangle, t): \]

if \( T[n, t] \neq \bot \) then return \( T[n, t] \)

if \( n = 0 \) then

\[ T[n, t] \leftarrow (t = 0) \]

else

\[ T[n, t] \leftarrow (a_n \leq t \text{ and } P(\langle a_1, \ldots, a_{n-1} \rangle, t - a_n)) \text{ or } P(\langle a_1, \ldots, a_{n-1} \rangle, t) \]

return \( T[n, t] \)
The “subproblem graph”

Iterative implementation: evaluate top to bottom / left to right
Iterative algorithm:

\[
T[0, 0] \leftarrow true
\]
for \( s \) in \([1..t]\) do

\[
T[0, s] \leftarrow false
\]

for \( i \) in \([1..n]\) do

for \( s \) in \([0..t]\) do

\[
T[i, s] \leftarrow (a_i \leq s \text{ and } T[i-1, s-a_i]) \text{ or } T[i-1, s]
\]

return \( T[n, t] \)
Iterative algorithm that returns a solution

\[
T[0, 0] \leftarrow true, \ Sol[0, 0] \leftarrow emptyList()
\]

for \(s\) in \([1..t]\) do
    \(T[0, s] \leftarrow false\)

for \(i\) in \([1..n]\) do
    for \(s\) in \([0..t]\) do
        if \(a_i \leq s\) and \(T[i-1, s-a_i]\) then
            \(T[i, s] \leftarrow true\)
            \(Sol[i, s] \leftarrow append(Sol[i-1, s-a_i], i)\)
        else
            \(T[i, s] \leftarrow T[i-1, s]\)
            \(Sol[i, s] \leftarrow Sol[i-1, s]\)

return \((T[n, t], Sol[n, t])\)
Example: Knapsack

An optimization version of Subset Sum

Given: positive integers $a_1, \ldots, a_n$, $t$, and $v_1, \ldots, v_n$

Question: find $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i \leq t$ which maximizes $\sum_{i \in I} v_i$

Intuition:

- you have a knapsack with a weight capacity of $t$
- you have $n$ items, where the $i$th item has weight $a_i$ and value $v_i$
- you want to pack the knapsack so as to maximize the value of the items in it, without exceeding its capacity
Recursive algorithm:

\[ Opt(\langle a_1, \ldots, a_n \rangle, t, \langle v_1, \ldots, v_n \rangle): \]

if \( n = 0 \) then
  return 0
else
  \[ // \text{val} = \text{value of best solution which does not include } n\text{th item} \]
  \[ \text{val} \leftarrow Opt(\langle a_1, \ldots, a_{n-1} \rangle, t, \langle v_1, \ldots, v_{n-1} \rangle) \]
  if \( a_n \leq t \) then
    \[ // \text{val'} = \text{value of best solution which includes } n\text{th item} \]
    \[ \text{val'} \leftarrow v_n + Opt(\langle a_1, \ldots, a_{n-1} \rangle, t - a_n, \langle v_1, \ldots, v_{n-1} \rangle) \]
    return \( \max(\text{val}, \text{val'}) \)
  else
    return \( \text{val} \)

Same subproblem structure as subset sum
Iterative algorithm with solution:

for $s$ in $[0..t]$ do
    $T[0, s] \leftarrow 0$, $Sol[0, s] \leftarrow EmptyList()$

for $i$ in $[1..n]$ do
    for $s$ in $[0..t]$ do
        if $a_i \leq s$ and $v_i + T[i-1, s-a_i] > T[i-1, s]$ then
            $T[i, s] \leftarrow v_i + T[i-1, s-a_i]$
            $Sol[i, s] \leftarrow append(Sol[i-1, s-a_i], i)$
        else
            $T[i, s] \leftarrow T[i-1, s]$
            $Sol[i, s] \leftarrow Sol[i-1, s]$

return $Sol[n, t]$
Example: Longest Common Subsequence

\[ X = \langle x_1, \ldots, x_m \rangle \]
\[ Z = \langle z_1, \ldots, z_k \rangle \]

\( Z \) is a subsequence of \( X \) if there exist indices \( i_1 < i_2 < \cdots < i_k \) such that \( x_{i_j} = z_j \) for \( j = 1 \ldots k \)

\[ X = \langle a, b, c, b, d, a, b \rangle \]
\[ Z = \langle b, c, d, b \rangle \]

Given sequences \( X \) and \( Y \), we say \( Z \) is a common subsequence of \( X \) and \( Y \) if \( Z \) is a subsequence of \( X \) and \( Z \) is a subsequence of \( Y \)
Example: \( X = \langle a, b, c, b, d, a, b \rangle, Y = \langle b, d, c, a, b, a \rangle \)

\( Z = \langle b, c, b, a \rangle \)

\[
\begin{align*}
X &= \langle a, b, c, b, d, a, b \rangle \\
Y &= \langle b, d, c, a, b, a \rangle
\end{align*}
\]

Problem: given \( X \) and \( Y \), find a Longest Common Subsequence (LCS) of \( X \) and \( Y \)

Notation: for \( X = \langle x_1, \ldots, x_m \rangle \) and \( i = 0 \ldots m \), we define \([X]_i := \langle x_1, \ldots, x_i \rangle\)
Key observation: Let $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$, and let $Z = \langle z_1, \ldots, z_k \rangle$ be an LCS of $X$ and $Y$

1. $x_m = y_n \implies z_k = x_m$ and $[Z]_{k-1}$ is an LCS of $[X]_{m-1}$ and $[Y]_{n-1}$

2. $(x_m \neq y_n) \land (z_k \neq x_m) \implies Z$ is an LCS of $[X]_{m-1}$ and $Y$

3. $(x_m \neq y_n) \land (z_k \neq y_n) \implies Z$ is an LCS of $X$ and $[Y]_{n-1}$

$X = \langle a, b, c, b, d, a, b \rangle$

$Y = \langle b, d, c, a, b, a, b, c, b \rangle$

$X = \langle a, b, c, b, d, a, b \rangle$

$Y = \langle b, d, c, a, b, a \rangle$

$X = \langle a, b, c, b, d, a, b \rangle$

$Y = \langle b, d, c, a, b, a \rangle$
Recursive algorithm LCS($X, Y$):

if $m = 0$ or $n = 0$ then
  return emptyList()
else if $x_m = y_n$ then
  $Z' \leftarrow$ LCS($[X]_{m-1}, [Y]_{n-1}$)
  $Z \leftarrow$ append($Z'$, $x_m$)
  return $Z$
else
  $Z_1 \leftarrow$ LCS($[X]_{m-1}, Y$)
  $Z_2 \leftarrow$ LCS($X, [Y]_{n-1}$)
  if length($Z_1$) > length($Z_2$) then
    return $Z_1$
  else
    return $Z_2$
Correctness: observation and induction on $m + n$

- Assume algorithm is correct on all inputs of length $< m + n$
- Consider an LCS $Z$ for the input
- Use induction hypothesis and observation to show that the algorithm constructs a solution of the same length as $Z$

There are only $(m + 1)(n + 1)$ distinct subproblems: $([X]_i, [Y]_j)$, for $i = 0 \ldots m$ and $j = 0 \ldots n$

Implement using a table — running time $= O(mn)$
The subproblem graph

Iterative implementation: evaluate top to bottom / left to right
Definitions

Let \( a_1 < a_2 < \cdots < a_n \)

An weight assignment is a tuple

\[(\alpha_1, \ldots, \alpha_n)\]

of real numbers

Interpretation: for \( i \in [1 \ldots n] \), \( \alpha_i \) may be the probability of accessing \( a_i \)
Let $T$ be a binary search tree for $a_1, \ldots, a_n$

- nodes labeled by keys $a_i$
- in-order traversal of tree yields $a_1, \ldots, a_n$

For a node $v$, let $d(v)$ denote its depth in the tree (level #, counting from zero)

Define the cost of $T$:

$$C := \sum_{i=1}^{n} a_i(1 + d(a_i))$$

If the weights are probabilities, $C$ represents the expected number of comparisons to perform a lookup
Example:

\[ C = 2 \cdot \frac{1}{8} + 3 \cdot \frac{1}{8} + 1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} = 1.625 \]

**Goal:** given weights, construct a binary search tree that minimizes \( C \)
Consider a tree $T$ with root $a_k$

The left subtree $T_L$ contains $a_1, \ldots, a_{k-1}$

Let $C_L$ be the cost of $T_L$

The right subtree $T_R$ contains $a_{k+1} \ldots, a_n$

Let $C_R$ be the cost of $T_R$

Let $S := \alpha_1 + \cdots + \alpha_n$

Key observation:

$$C = S + C_L + C_R$$

This means that if $T$ is optimal, then $C_L$ and $C_R$ must be optimal
For $1 \leq k \leq \ell \leq n$, let
\[ S(k, \ell) := \alpha_k + \cdots + \alpha_{\ell} \]
and define $C(k, \ell)$ to be the optimum cost for $\alpha_k, \ldots, \alpha_{\ell}$.

We have
\[ S(k, \ell) = S(k, \ell - 1) + \alpha_{\ell} \]
\[ C(k, \ell) = S(k, \ell) + \min_{k \leq m \leq \ell} \left( C(k, m - 1) + C(m + 1, \ell) \right) \]
where
\[ C(k, k - 1) := C(\ell + 1, \ell) := 0 \]
and
\[ S(k, k - 1) := 0 \]
The subproblem graph:

# nodes = \(O(n^2)\), # edges per node = \(O(n)\)
\[\implies\] running time = \(O(n^3)\)
General dynamic programming strategies:

• Formulate a recursive solution
  
  ○ Optimal substructure property: an optimal solution can be constructed efficiently from optimal solutions of its subproblems

• Identify subproblems and dependencies

• Analyze structure of subproblem graph

• Iterative solution: determine a convenient “topological order” for the nodes in the graph

• Running time analysis: (usually) equals the size of the subproblem graph (# nodes + # edges)
Some subtleties:

- Must choose subproblems so that:
  - the correct/optimal solution can be reconstructed
    - for Subset Sum and Optimum Weighted Trees, this was easy
    - for LCS, some care was necessary to ensure the optimal solution was not missed
  - the number of distinct subproblems does not explode exponentially
Example: variations on “tiling”

We are given “target” $t \in \{0, 1\}^*$, and several “tiles” $s_1, \ldots, s_k \in \{0, 1\}^*$

Can $t$ be obtained by concatenating some of the tiles?

Variations:

• repetitions, given order (easy)
• repetitions, any order (easy)
• no repetitions, given order (easy)
• no repetitions, any order (???)