Maximum Subsequence Solution

New Induction Hypothesis: We can find \(\text{SUM}(n-1)\) and \(\text{TRAILINGSUM}(n-1)\) for any sequence of \(n - 1\) integers.

Base case: \(\text{SUM}(1) = \text{TRAILINGSUM}(1) = \text{Max}(0, x_1)\).

Induction step: 
\[
\text{SUM}(n) = \text{Max}(\text{SUM}(n-1), \text{TRAILINGSUM}(n-1) + x_n).
\]
\[
\text{TRAILINGSUM}(n) = \text{Max}(0, \text{TRAILINGSUM}(n-1) + x_n).
\]

Example:

Instead of parameterizing the problem just by the number of items \(n\), we parameterize by both \(n\) and by \(K\).

First consider the decision problem: Is there a subset \(S\) such that \(\sum S_i = K\)?

**Knapsack Algorithm Approach**

Instead of parameterizing the problem just by the number of items \(n\), we parameterize by both \(n\) and by \(K\).

\(P(n, K)\) is the problem with \(n\) items and capacity \(K\).

First consider the decision problem: Is there a subset \(S\)?

**Induction Hypothesis:**

We know how to solve \(P(n - 1, K)\).
Knapsack Induction

Induction Hypothesis:
We know how to solve $P(n - 1, K)$.

Solving $P(n, K)$:
- If $P(n - 1, K)$ has a solution, then it is also a solution for $P(n, K)$.
- Otherwise, $P(n, K)$ has a solution iff $P(n - 1, K - k_n)$ has a solution.

So what should the induction hypothesis really be?

Knapsack: New Induction

- New Induction Hypothesis:
  We know how to solve $P(n - 1, k), 0 \leq k \leq K$.
- To solve $P(n, K)$:
  - If $P(n - 1, K)$ has a solution, then $P(n, K)$ has a solution.
  - Else if $P(n - 1, K - k_n)$ has a solution, then $P(n, K)$ has a solution.
  - Else $P(n, K)$ has no solution.

Algorithm Complexity

- Resulting algorithm complexity:
  - $T(n) = 2T(n - 1) + c$ for $n \geq 2$.
  - $T(n) = \Theta(2^n)$ by expanding sum.
- But, there are only $n(K + 1)$ problems defined.
  - It must be that problems are being re-solved many times by this algorithm. Don’t do that.

Efficient Algorithm Implementation

The key is to avoid re-computing subproblems.

Implementation:
- Store an $n \times (K + 1)$ matrix to contain solutions for all the $P(i, k)$.
- Fill in the table row by row.
- Alternately, fill in table using logic above.

Analysis:
$T(n) = \Theta(nK)$.
Space needed is also $\Theta(nK)$.
Example

\[ K = 10, \text{ with 5 items having size 9, 2, 7, 4, 1.} \]

<table>
<thead>
<tr>
<th>k</th>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O/O</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>3</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O/O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>4</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>5</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

Key:
- No solution for \(P(i, k)\)
- \(O\) Solution(s) for \(P(i, k)\) with \(i\) omitted.
- \(I\) Solution(s) for \(P(i, k)\) with \(i\) included.
- \(I/O\) Solutions for \(P(i, k)\) both with \(i\) included and with \(i\) omitted.

Solution Graph

Find all solutions for \(P(5, 10)\).  
\[
M(1, 0) \quad M(1, 9) \\
M(2, 2) \quad M(2, 9) \\
M(3, 9) \\
M(4, 9) \\
M(5, 10)
\]

The result is an \(n\)-level DAG.

Dynamic Programming

This approach of storing solutions to subproblems in a table is called \textbf{dynamic programming}.

It is useful when the number of distinct subproblems is not too large, but subproblems are executed repeatedly.

Implementation: Nested for loops with logic to fill in a single entry.

Most useful for \textbf{optimization problems}.

Fibonacci Sequence

\[
\text{int Fibr(int n) } \{
    \text{if } (n \leq 1) \text{ return 1; } \quad \text{// Base case }
    \text{return Fibr(n-1) + Fibr(n-2); } \quad \text{// Recursion }
\}
\]

- Cost is Exponential. Why?
- If we could eliminate redundancy, cost would be greatly reduced.
### Fibonacci Sequence (cont)

- Keep a table

```c
int Fibrt(int n, int* Values) {
    // Assume Values has at least n slots, and
    // all slots are initialized to 0
    if (n <= 1) return 1; // Base case
    if (Values[n] == 0) // Compute and store
        Values[n] = Fibrt(n-1, Values) +
                    Fibrt(n-2, Values);
    return Values[n];
}
```

- **Cost?**
- **We don’t need table, only last 2 values.**
  - Key is working bottom up.