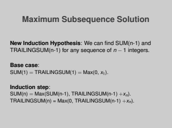
Maximum Subsequence Solution

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

Base case:
SUM(1) $=$ TRAILINGSUM(1) $=$ Max(0, x_1).

Induction step:
SUM(n) $=$ Max(SUM(n-1), TRAILINGSUM(n-1) $+x_n$).
TRAILINGSUM(n) = Max(0, TRAILINGSUM(n-1) $+x_n$).



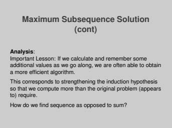
no notes

Maximum Subsequence Solution (cont)

Analysis:
Important Lesson: If we calculate and remember some additional values as we go along, we are often able to obtain a more efficient algorithm.

This corresponds to strengthening the induction hypothesis so that we compute more than the original problem (appears to) require.

How do we find sequence as opposed to sum?



$\mathrm{O}(n)$. $T(n) = T(n-1) + 2$.
Remember position information as well.

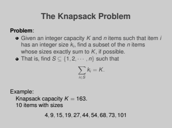
The Knapsack Problem

Problem:
- Given an integer capacity K and n items such that item i has an integer size k_i, find a subset of the n items whose sizes exactly sum to K, if possible.
- That is, find $S \subseteq \{1, 2, \cdots, n\}$ such that

$$\sum_{i \in S} k_i = K.$$

Example:
Knapsack capacity $K = 163$.
10 items with sizes

$$4, 9, 15, 19, 27, 44, 54, 68, 73, 101$$



This version of Knapsack is one of several variations.
Think about solving this for 163. An answer is:

$$S = \{9, 27, 54, 73\}$$

Now, try solving for $K = 164$. An answer is:

$$S = \{19, 44, 101\}.$$

There is no relationship between these solutions!

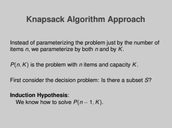
Knapsack Algorithm Approach

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 $P(n-1, K)$.



Is there a subset S such that $\sum S_i = K$?

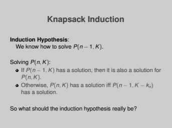
Knapsack Induction

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?

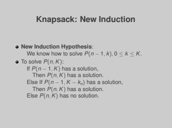


But... I don't know how to solve $P(n-1, K-k_n)$ since it is not in my induction hypothesis! So, we must strengthen the induction hypothesis.

New Induction Hypothesis:
We know how to solve $P(n-1, k), 0 \leq k \leq K$.

Knapsack: New Induction

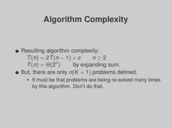
- **New Induction Hypothesis**:
  We know how to solve $P(n-1, k), 0 \leq k \leq K$.
- To solve $P(n, K)$:
  If $P(n-1, K)$ has a solution,
  Then $P(n, K)$ has a solution.
  Else If $P(n-1, K-k_n)$ has a solution,
  Then $P(n, K)$ has a solution.
  Else $P(n, K)$ has no solution.



Need to solve two subproblems: $P(n-1, k)$ and $P(n-1, k-k_n)$.

Algorithm Complexity

- Resulting algorithm complexity:
  $T(n) = 2T(n-1) + c \qquad n \geq 2$
  $T(n) = \Theta(2^n) \qquad$ 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.



Problem: Can't use Theorem 3.4 in this form.

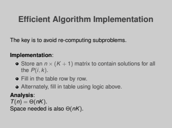
Efficient Algorithm Implementation

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 $\Theta(nK)$.



To solve $P(i, k)$ look at entry in the table.
If it is marked, then OK.
Otherwise solve recursively.
Initially, fill in all $P(i, 0)$.

Example

$K = 10$, with 5 items having size 9, 2, 7, 4, 1.

	0	1	2	3	4	5	6	7	8	9	10
$k_1 = 9$	O	–	–	–	–	–	–	–	–	I	–
$k_2 = 2$	O	–	I	–	–	–	–	–	–	O	–
$k_3 = 7$	O	–	O	–	–	–	–	I	–	I/O	–
$k_4 = 4$	O	–	O	–	I	–	I	O	–	O	–
$k_5 = 1$	O	I	O	I	O	I	O	I/O	I	O	I

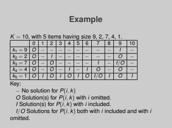
Key:

– 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.

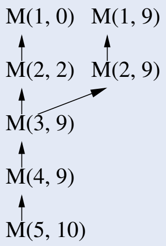
I/O Solutions for $P(i, k)$ both with i included and with i omitted.



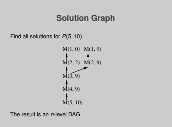
Example: M(3, 9) contains O because $P(2, 9)$ has a solution.
It contains I because $P(2, 2) = P(2, 9 - 7)$ has a solution.
How can we find a solution to $P(5, 10)$ from M?
How can we find **all** solutions for $P(5, 10)$?

Solution Graph

Find all solutions for $P(5, 10)$.



The result is an n-level DAG.



Alternative approach:
Do not precompute matrix. Instead, solve subproblems as necessary, marking in the array during backtracking.
To avoid storing the large array, use hashing for storing (and retrieving) subproblem solutions.

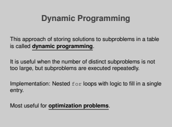
Dynamic Programming

This approach of storing solutions to subproblems in a table is called **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 **optimization problems**.

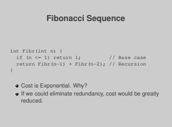


no notes

Fibonacci Sequence

```
int Fibr(int n) {
  if (n <= 1) return 1;          // Base case
  return Fibr(n-1) + Fibr(n-2); // Recursion
}
```

- Cost is Exponential. Why?
- If we could eliminate redundancy, cost would be greatly reduced.



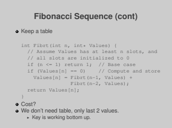
Essentially, we are making as many function calls as the value of the Fibonacci sequence itself. It is roughly (though not quite) two function calls of size $n - 1$ each.

Fibonacci Sequence (cont)

- Keep a table

```
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.



no notes